Fudan University

復旦大學

FPGA Embedded System Lab Report

FPGA Muse Dash

By DE2-115 Team D1

Members: Chen Zhiyi 21307130003

Wang Tianchen 21307130031

Dong Yuxuan 21307130081

Chen Yikuan 21307140002

June 2024

Contents

1 Introduction

1.1 Lab Requirements

In our Embedded System Design Lab, students are presented with three exciting project options to choose from. Each option allows students to explore different aspects of embedded system design, combining hardware and software skills.

The first option is to create a game that is displayed on a monitor and operated using a keyboard or mouse. The second option involves building a web server with interactive control capabilities. While the third option allows students to design a custom embedded system with software and hardware co-designs.

For these projects, students should use FPGA boards, specifically the Altera (Intel) DE2-115 or DE1-SoC, or other possible boards.

In our lab, we chose the third option, which is to custom a embedded system with software and hardware co-designs. We use C++ programs to generate ROM verilog files to operate with hardware rhythm game.

1.2 Lab Idea

Muse Dash is a rhythm game developed by PeroPeroGames from China, and published by XD Network in Japan and hasuhasu outside of Japan. It was initially released for iOS and Android in June 2018, later being released on Nintendo Switch, Windows and macOS on June 20, 2019. It combines aspects of action games and music games, using an anime art style in the form of a 2D side-scrolling video game. The icon of the game is Figure 1.

During gameplay, the player will control the playable Muse in the one of two options that they select. Two circle on screen represent the top and bottom areas of attack, depending on the control type is depending on where the player will tap to have the playable Muse enter the top or bottom lines. The Muse will always be in the bottom row and will land whenever sent to the top row. Before a song is played, the player is urged to go into the menus and configure their offset to match with their headphones. The game is played like shown in Figure 2.

Figure 1: Muse Dash Icon

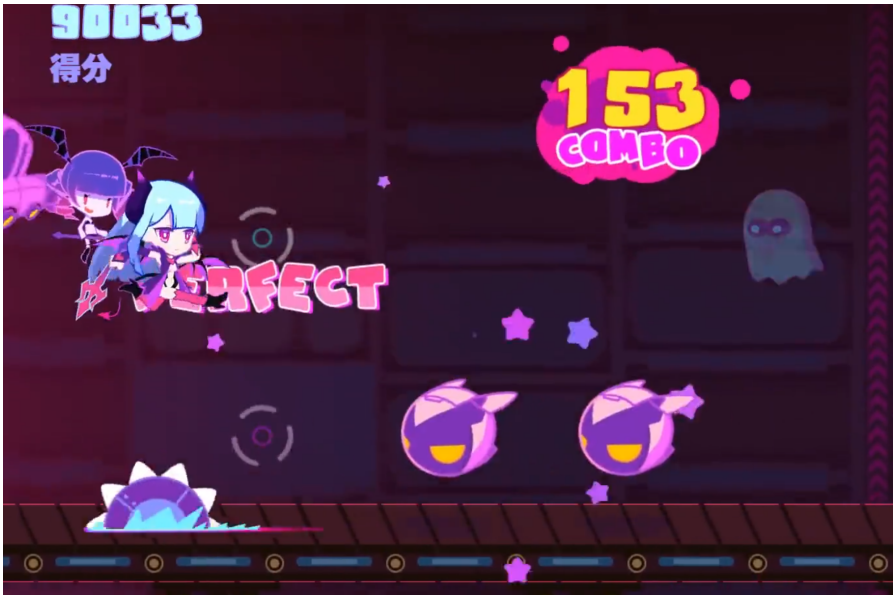


Figure 2: Game Play

The default PC control (shown in Figure 3) is using keys to control the related track.

The core idea of our project is inspired by this particular game. As the music plays (although we haven't implemented music playback yet), notes will continuously scroll from right to left on the two tracks displayed on the TextLCD screen. Our goal is to press the corresponding track button precisely when the notes reach the leftmost end. The more accurate the timing, the higher the score. Each game's score accumulates until the music ends, allowing players to see their scores and choose to retry if they wish.

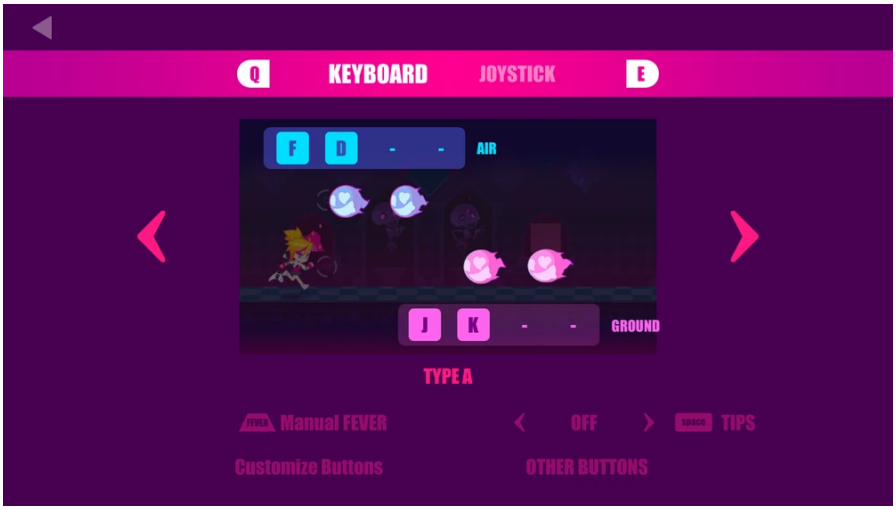


Figure 3: Default PC Control

1.3 Work Introduction

Our project is named FPGA Muse Dash, which is a rhythm game inspired by Muse Dash and implemented on an FPGA board.

The gameplay is straightforward: as notes scroll from right to left on the TextLCD, you need to react accordingly when the corresponding note reaches the far-left column to score points. An O represents a *TAP* note, which you score by pressing once. A < represents a *HOLD_START* note, which marks the beginning of a note that needs to be held down, judged similarly to a *TAP* note. A □ represents a *HOLD_MIDDLE* note, which is the body or end of a hold note that needs to be held down continuously. We will revisit these note types in section 2.1.

The first button on the left controls the upper track, the second button from the left controls the lower track, the first button from the right is the restart button, and the first switch from the right is the `rst_n` switch.

The judgement results are displayed on seven-segment displays: the two leftmost segments correspond to the judgements for the upper track, the third and fourth segments from the left correspond to the judgements for the lower track, and the four rightmost segments show the cumulative score. The judgement results include **PERFECT**, **GOOD**, and **MISS**, which are displayed as PF, GO, and FL on the seven-segment displays, respectively. PF scores 4 points, GO scores 2 points, and FL scores no points. The judgement results are displayed in

real time, and the scores are accumulated in real time as well.

The exact interfaces on board are shown in Figure 4.

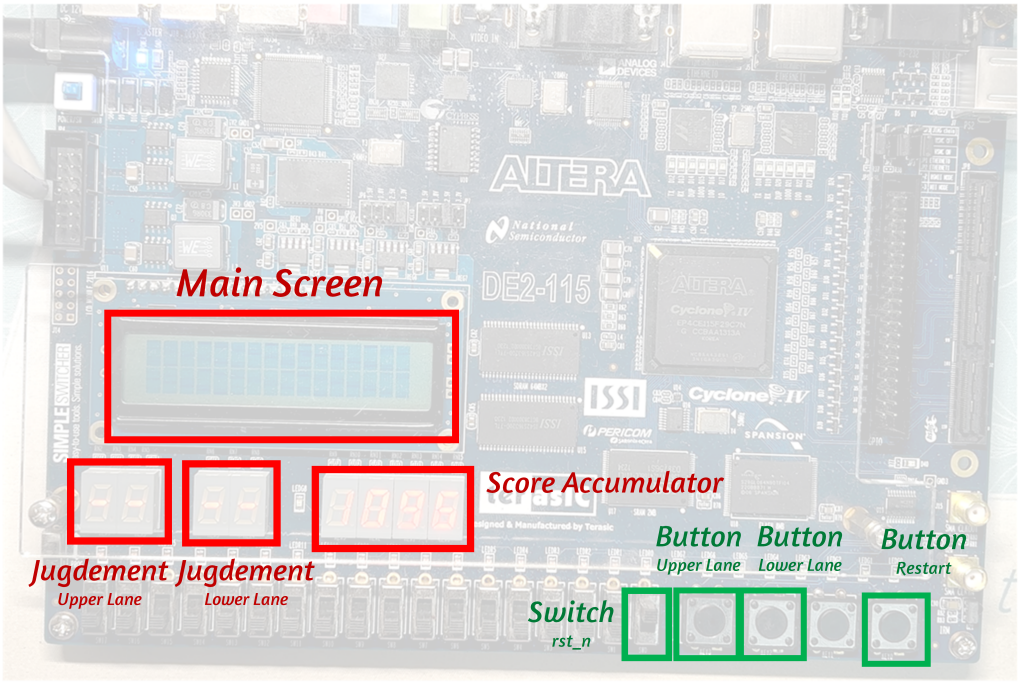


Figure 4: Interfaces on board

1.4 ToolChain

The toochain we have used are as follows:

1. Version Control: gitee
2. Verilog Coding: VSCode + Vivado
3. C++ Coding: VSCode + Mac/Linux environment
4. Behavioral Simulation: Vivado
5. Hardware Verification: DE2-115 development board with Quartus Platform

1.5 Division of Labor

- Verilog Coding: Chen Zhiyi, Wang Tianchen, Dong Yuxuan

- C++ Coding: Chen Yikuan, Dong Yuxuan
- Verification: Wang Tianchen, Chen Zhiyi, Dong Yuxuan
- Report Writing: Dong Yuxuan, Chen Zhiyi, Chen Yikuan, Wang Tianchen
- Diagram Drawing: Chen Yikuan, Chen Zhiyi
- Extensive Exploration: Chen Yikuan
- PPT and Presentation: Chen Zhiyi

2 Software Design

2.1 Input Chart Format

In the rhythm game, every music has its corresponding music chart that includes all the notes in the music. One would expect that a rhythm game has a standardized chart format, and our Muse Dash game uses a `txt` file as the chart, which begins with the speed of the music (BPM), and then shows the information of each note.

Our chart uses 3 parameters to describe a note in the music: time, type of note, and track of note.

- Time represents the number of clock period the note is supposed to be judged.
- A note is classified into the following 3 types, each with different judgements. Details of judgements are explained in section 3.9.
  ①*TAP* note means the player should click the corresponding button, and an offset either too early or too late will affect the judgement.
  ②*HOLD_START* note indicates the start of a hold note that requires the player to keep the note down until it ends. It is judged the same way as the tap note.
  ③*HOLD_MIDDLE* note makes up the rest of hold note.
- The track of note indicates where the note will be. In our game, a note will only appear on the lower or upper track.

Regarding code representation, we use a 2-bit number for note type, where 0 is for *NO_NOTE*, 1 is for *TAP*, 2 is for *HOLD_START*, and 3 is for *HOLD_MIDDLE*. We use a single bit for track representation, where 0 is for the lower track, and 1 is for the upper track. An example of a chart is as below.

```
bpm=150
(16,tap,0)
(18,tap,1)
(32,hold_start,0)
(33,hold_mid,0)
(34,hold_mid,0)
(35,hold_mid,0)
```

In the display video, we manually transplanted a chart from the original rhythm game Muse Dash, with a reference of the original chart. It is proved that the designed chart format is able to achieve similar visual effects compared to real game.

2.2 Chart To ROM

With the standardized chart, we use a `cpp` file to write our chart information into a ROM verilog file, which is used for the following hardware design.

The `cpp` file `handle_input.cpp` first extracts the bpm and notes information, and process the notes information into vector `rom`, a double vector of `bitset` type. Then the processed data is used to generate the ROM file. The key function for ROM generation is shown in Figure 5.

The function takes the processed data `rom` as input, and output the generated ROM file to the given file name. Besides writing the data from `rom`, the function completes other hardware declaration so that the output file is correct in verilog grammar and functions normally as a hardware design.

By the way, the bpm information is used to modify the parameter in the top module `MuseDash`

```
52  void outputROM(const vector<vector<int>>& rom, const string& outputFilename) {
53      ofstream outfile(outputFilename);
54      outfile << "module ROM (\n";
55      outfile << "    input [11:0] addr,\n";
56      outfile << "    output reg [1:0] noteup,\n";
57      outfile << "    output reg [1:0] notedown\n";
58      outfile << ");\n\n";
59
60      outfile << "reg [3:0] ROM [0:4095];\n\n";
61      outfile << "initial begin\n";
62      for (int i = 0; i < 4096; ++i) {
63          outfile << "\tROM[" << i << "] = 4'b" << rom[i][3] << rom[i][2] << rom[i][1] << rom[i][0] << ";" << endl;
64      }
65      outfile << "end\n\n";
66      outfile << "always @(*) begin\n";
67      outfile << "    {noteup, notedown} = ROM[addr];\n";
68      outfile << "end\n\n";
69      outfile << "endmodule\n";
70  }
```

Figure 5: ROM output function

in a way similar as above.

3 Hardware Design

3.1 Top Module

The top-level module is responsible for interconnecting various submodules and performing some basic logic operations. For instance, it integrates the reset signals (`rst_n` and `restart`), and manages the slicing and concatenation of wire arrays. This modular setup ensures that each submodule functions cohesively within the overall design. The detailed module diagram is illustrated below.

By providing a centralized point for signal routing and basic logic, the top-level module simplifies the overall system architecture. It handles the interfacing between submodules, ensuring that they communicate effectively and perform their designated tasks. This design approach promotes modularity and scalability, making the system easier to manage and debug.

The diagram provides a visual representation of the connections and interactions between the submodules, highlighting how they are orchestrated within the top-level module to achieve the desired functionality, see Figure 6.

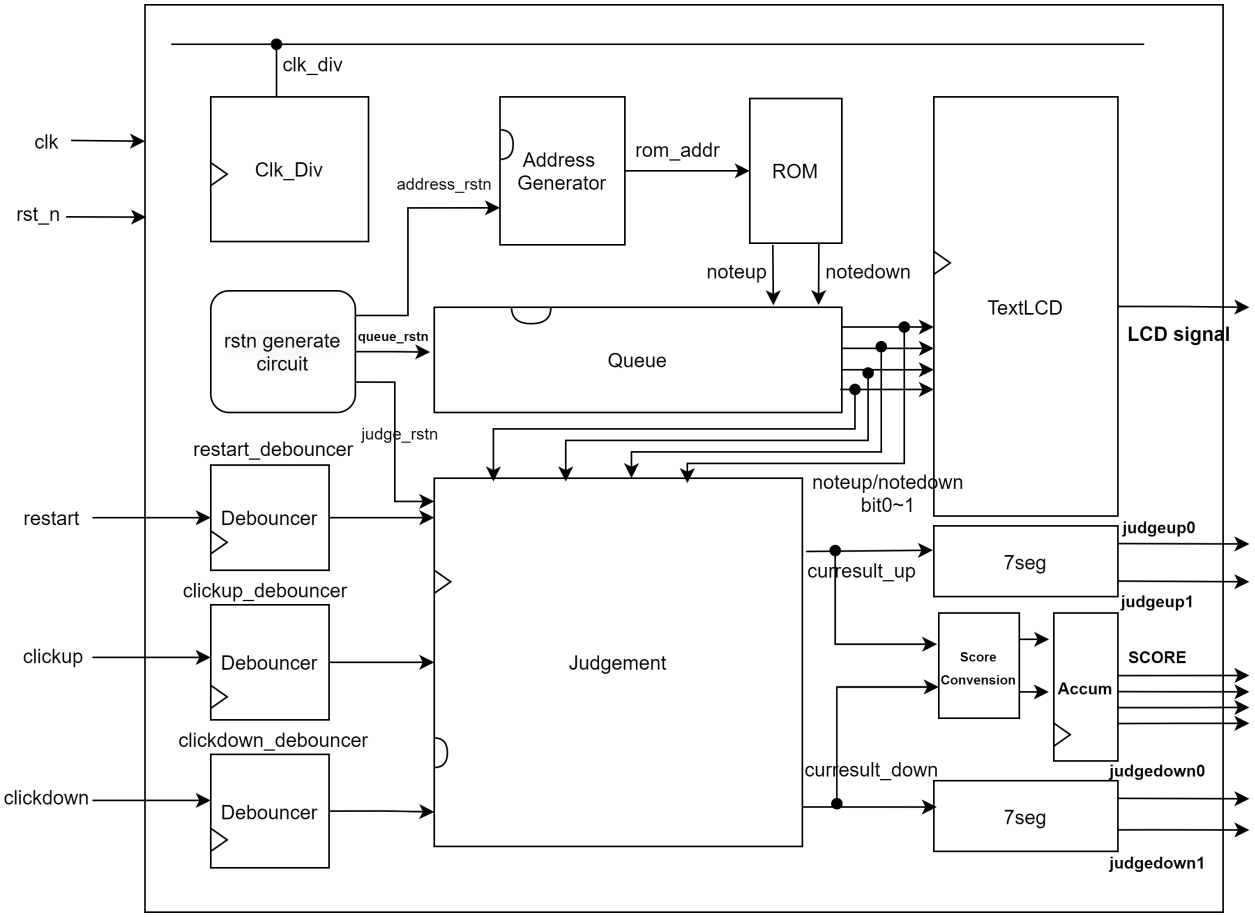


Figure 6: Top Module

3.2 Distributed Architecture

In our hardware design, we adopt a distributed architecture, distributing different functionalities across various modules. This approach effectively makes each module lightweight, facilitating easier debugging and simulation. Additionally, it allows for convenient task allocation within a team, making collaborative design more efficient. As illustrated in the Figure 7, we divide the overall design into 14 modules.

The top-level design is the MuseDash module. The `clk_div` module is responsible for generating variable frequency clocks. The random number generation is handled by the LFSR module. The Debouncer module manages button debouncing. For seven-segment displays, the CurrentJudge7Seg module handles the logic for current judgement display, while the TotalScore7Seg module is responsible for displaying the total score. The TextLCD module is related to TextLCD display. Chart data is stored in the ROM module and retrieved by the

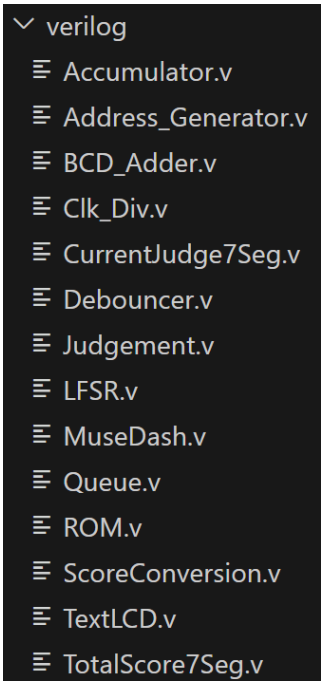


Figure 7: Verilog File Catalog

Address_Generator module, which then passes the data to the Queue module for processing. The Queue module feeds the data to the Judgement module for decision-making and also sends it to the TextLCD module for display.

The core logic module is the Judgement module. The Accumulator and BCD_Adder modules handle score accumulation and addition. The ScoreConversion module converts judgement signals into BCD score values for the accumulator to sum up.

This distributed hardware module design significantly reduces the difficulty of debugging. During the final testing phase, the primary module requiring simulation is the core Judgement module. Other modules either use existing code or are straightforward to verify for correctness. This modular approach streamlines the design process and enhances the efficiency of the development cycle.

3.3 Hierarchical Design

Our design also employs hierarchical structure, which is mainly reflected in the design of the adder and accumulator. For the score accumulation adder, we use a 4-bit BCD_Adder as the basic unit, first constructing a 16-bit BCD adder, and then transforming the adder into an

accumulator. See Figure 8.

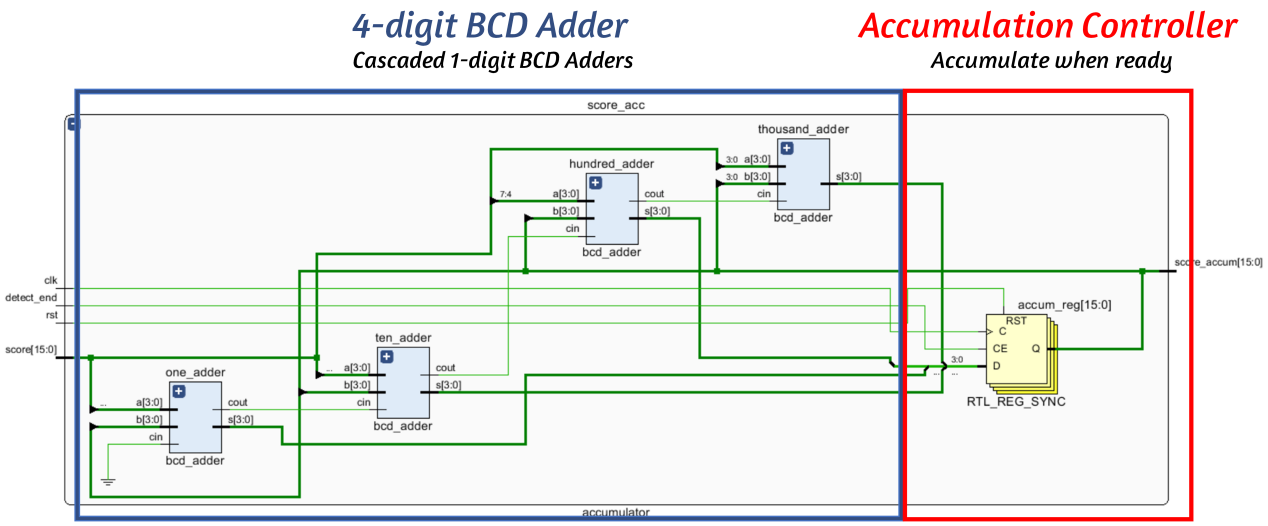


Figure 8: Score Accumulator Schematic

Since this accumulation needs to sum the scores of the judgements from both the upper and lower tracks at once, we use a method where Pre-Addition is performed in the ScoreConversion module, and the resulting score is then passed to the Accumulator for overall accumulation.

This hierarchical design approach minimizes the impact of parameter changes on the overall design and also makes debugging more convenient. By structuring the design in layers, we can isolate and manage different parts of the system more effectively, ensuring that changes or errors in one part do not adversely affect the entire system. This method enhances the robustness and maintainability of the design, contributing to a more streamlined and efficient development process.

3.4 Adjustable Clock Divider

The `clk_div` module is designed to generate a divided clock signal (`clk_div`) from a given input clock (clk). This module incorporates a parameter called div_cnt, which defines the division factor. By default, this parameter is set to 2,500,000, converting a 50 MHz input clock to a 10 Hz output clock, equating to a period of 100 milliseconds. The division counter (cnt) counts the clock cycles of the input clock. When the counter reaches the value specified by div_cnt, it toggles the `clk_div` signal and resets the counter to zero, effectively creating a

slower clock signal.

One of the notable features of this module is that the div_cnt parameter is adjustable. This flexibility allows the module to accommodate various clock division requirements, making it adaptable for different timing needs. For instance, the parameter can be set to other values to achieve different output frequencies, which is particularly useful for applications that require specific timing intervals based on beats per minute (BPM) or other timing constraints. As you can see, we have extended this parameter to the top module so that the C++ program can easily change the required div_cnt value.

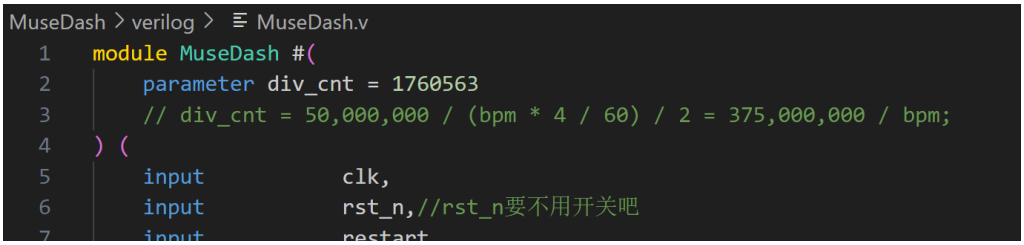


Figure 9: Top Module Parameter div_cnt

In operation, the module continuously monitors the rising edge of the input clock. Upon detecting a reset signal (`rst_n`) being inactive, it resets the counter and the divided clock signal. When the counter reaches the predefined division count, the divided clock signal is toggled, and the counter is reset, ensuring a consistent divided clock output. This mechanism allows the module to provide a reliable clock signal that is essential for synchronizing various components in a digital system.

3.5 Debouncement

The Debouncer module is designed to eliminate noise from a mechanical switch or button input (click_in) to ensure a stable output signal (click_out). When a button is pressed, it can generate multiple transient signals due to mechanical bouncing. This module helps filter out these unwanted transients by employing a debouncing mechanism. The ports of Debouncer is shown in Figure 10.

The module operates by using a counter and a state flag to monitor changes in the input signal. When a potential key press is detected (i.e. the key state `click_in` changes, and it is

```
module Debouncer(
    input clk,
    input rst_n,
    input click_in,

    output reg click_out
);
```

Figure 10: Ports of Debouncer

not currently in a counting state), the signal `start` goes high, indicating the start of counting. After counting starts, the counter ‘counter‘ decrements by 1 each clock cycle. When the counter reaches zero, it stops counting and outputs the current key value `click_in`. If it was due to jitter or glitch, the key state should be the same as initially, and therefore the output remains unchanged; if a key press event has indeed occurred, it will be correctly outputted. Thus, the debouncing effect is achieved.

At each rising edge of the clock signal (`clk`), the module checks whether a reset signal (`rst_n`) is active. If a reset is detected, it initializes the output signal and the counting flag. When a change in the input signal is detected and the module is not currently counting, it starts the counting process by setting the counting flag. During this period, the counter decrements with each clock cycle until it reaches zero. Once the counter reaches zero, the input signal is considered stable, and the output signal is updated to reflect the input state.

The debouncing process ensures that only a stable and consistent input signal is propagated to the output. This is particularly important in digital systems where precise and accurate input signals are crucial for proper operation. The module’s ability to filter out noise and provide a clean signal makes it an essential component in applications involving mechanical switches and buttons.

3.6 Random Generation

LFSR (Linear Feedback Shift Register) is used to generate repeatable pseudo-random sequences (PRBS). This circuit consists of n flip-flops and several XOR gates. In each clock cycle, new input values are fed back into the inputs of the flip-flops within the LFSR. Part of the input values come from the outputs of the LFSR, while another part is obtained by

performing XOR operations on the outputs of the LFSR.

The characteristics of this circuit are as follows:

1. If the initial state is the same, the final output sequence will be the same (i.e., the output sequence is deterministic).
2. The output sequence tends to be random (pseudo-random).
3. After a certain number of iterations, the state values will return to the initial state. The maximum repetition interval is $(2^n - 1)$, where n is the number of shift registers.

Due to these characteristics, LFSR is mainly used to generate PN sequences (pseudo-noise sequences).

In the specific implementation of the LFSR, to ensure the maximum number of iterations for a given bit length while reducing circuit complexity, a method is typically used where a new bit (next_bit) is generated based on the current code, and the register is shifted to insert the new bit. The next_bit is generally obtained according to the LFSR polynomial table algorithm, as shown in Figure 11.

The project utilizes a 13-bit LFSR, primarily used for random input testing when the overall code is not yet completed. The algorithm used is Figure 12.

3.7 TextLCD Display

3.7.1 Queue Design

As mentioned in section 3.2, the data from ROM first goes through a queue. The processed data from the queue is then fed to TextLCD module. So we design a queue module to process and save temporary data for subsequent textLCD display.

The Queue module receives the divided clock `clk_div` to update inner data, which also serve as output of the module. The inner data are managed into four parts: notes on the upper track and on the lower track, each further separated into two bits. Every divided clock period, the inner data shift right a bit and the lowest bits are replaced by the input data, and therefore completes a move in the queue. The application diagram of this queue design is shown in

Bits (n)	Feedback polynomial	Taps	Taps (hex)	Period ($2^n - 1$)
2	$x^2 + x + 1$	11	0x3	3
3	$x^3 + x^2 + 1$	110	0x6	7
4	$x^4 + x^3 + 1$	1100	0xC	15
5	$x^5 + x^3 + 1$	10100	0x14	31
6	$x^6 + x^5 + 1$	110000	0x30	63
7	$x^7 + x^6 + 1$	1100000	0x60	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	10111000	0xB8	255
9	$x^9 + x^5 + 1$	100010000	0x110	511
10	$x^{10} + x^7 + 1$	1001000000	0x240	1, 023
11	$x^{11} + x^9 + 1$	10100000000	0x500	2, 047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	111000001000	0xE08	4, 095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	1110010000000	0x1C80	8, 191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	11100000000010	0x3802	16, 383
15	$x^{15} + x^{14} + 1$	110000000000000	0x6000	32, 767
16	$x^{16} + x^{15} + x^{13} + x^4 + 1$	1101000000001000	0xD008	65, 535
17	$x^{17} + x^{14} + 1$	10010000000000000	0x12000	131, 071
18	$x^{18} + x^{11} + 1$	100000010000000000	0x20400	262, 143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	1110010000000000000	0x72000	524, 287
20	$x^{20} + x^{17} + 1$	10010000000000000000	0x90000	1, 048, 575
21	$x^{21} + x^{19} + 1$	10100000000000000000	0x140000	2, 097, 151
22	$x^{22} + x^{21} + 1$	11000000000000000000	0x300000	4, 194, 303
23	$x^{23} + x^{18} + 1$	10000100000000000000000	0x420000	8, 388, 607
24	$x^{24} + x^{23} + x^{22} + x^{17} + 1$	111000010000000000000000	0xE10000	16, 777, 215

Figure 11: LFSR Polynomial Table

```
// LFSR Algorithm
assign next_bit     = lfsr_reg[12]
                    ^ lfsr_reg[11]
                    ^ lfsr_reg[10]
                    ^ lfsr_reg[7]
                    ^ lfsr_reg[0];
```

Figure 12: LFSR Algorithm Verilog Code

Figure 13.

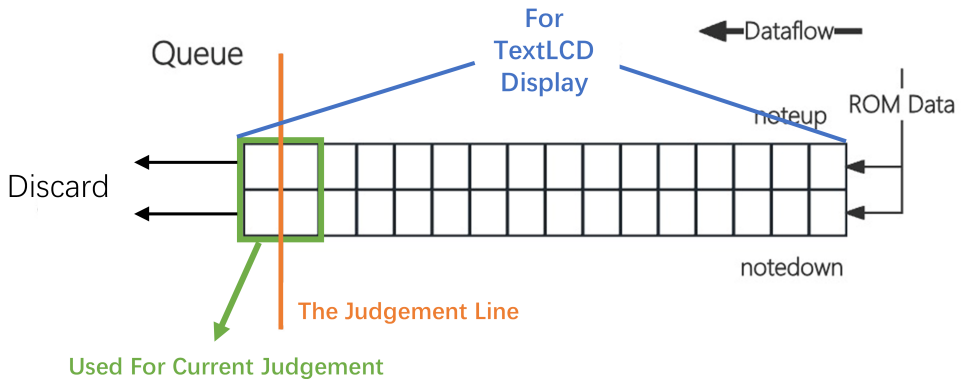


Figure 13: Queue Design

3.7.2 TextLCD Design

The TextLCD display module receices the output of Queue module and generates necessary signals for textLCD part, including `LCD_DATA`, `LCD_EN`, `LCD_RS`, *etc.*

The HD44780 textLCD provides various control instructions. Basic settings must be configured in command mode before any basic operations can begin. The basic operation modes are:

- Read status (RS=0, RW=1)
- Write instructions (RS=0, RW=0)
- Read data (RS=1, RW=1)
- Write data (RS=1, RW=0)

As HD44780 device requires complete setup before displaying, writing instructions is necessary. Based on this, our design only involves two modes, writing instructions and writing data. Specifically, the input data are sequentially written to `LCD_DATA` to display a complete frame of notes.

For data conversion, we combines the inputs from Queue Module and map different types of note to corresponding ASCII codes, which will later be written into the Display Data RAM (DDRAM) in the HD44780 device.

In the display part, we set up a simple loop state machine to switch between the above

two modes. The state transition is as follows: IDLE → CLEAR → SET_FUNCTION → SWITCH_MODE → SHIFT → SET_MODE → SET_DDRAM1 → WRITE_RAM1 → SET_DDRAM2 → WRITE_RAM2 → IDLE.

IDLE refers to idle state. CLEAR empties the screen. The following 4 states beginning with SET_FUNCTION set the type of input data, display mode, and screen cursor. Then for each of the DDRAM, we set the write address and write corresponding data. Up to now, a frame of notes is successfully displayed, and the state machine repeats the cycle.

In the above states, WRITE_RAM1 and WRITE_RAM2 are in write data mode (RS=1, RW=0), and other states are in write instruction mode (RS=0, RW=0). For HD44780 device, there are fixed instructions to complete the above operations, including instructions to set the write address for DDRAM. Figure 14 is a Chinese manual that explains the relation between DDRAM address and the position displayed on the screen.

	显示位置	1	2	3	4	5	6	7	……	40
DDRAM 地 址	第一行	00H	01H	02H	03H	04H	05H	06H	……	27H
	第二行	40H	41H	42H	43H	44H	45H	46H	……	67H

Figure 14: DDRAM write address specification

3.7.3 Dual Frequency Design

Notice that in the TextLCD module, we include another clock divider that generates `clk_buf` signal, a divided clock signal with different frequency compared to `clk_div`. In the design, the frequency of `clk_div` is controlled by the bpm of the music, whereas the frequency of `clk_buf` is controlled by the frame rate of the screen.

Introducing an independent clock signal for screen frame rate will leave optimization space for visual effects, considering that frame rate could evidently affect the experience of a rhythm game player. We've tried different frame rates with regard to `clk_div`, and have found different visual effects. When frame rates are set too high, there will not be enough time for the state machine to complete the DDRAM writing, causing the display patterns to flicker and hard to recognize. On the contrary, a frame rate too low will cause a mismatch between the patterns of the upper track and the lower track, when the latency to write the second DDRAM becomes observable.

The above discovery leads to another conclusion: `clk_div` is too slow for display. This proves again that our dual frequency design is meaningful.

3.8 7-Segment Display

In a four-digit seven-segment display, each digit position has seven individual LED segments: A, B, C, D, E, F, and G. Additionally, there is an eighth segment, often referred to as the decimal point.

To display a number, the corresponding segments are illuminated while the others remain dark. For example, to display the numeral 0, segments A, B, C, D, E, and F are illuminated, while segment G remains dark. Similarly, to display the numeral 1, segments B and C are illuminated, and so forth for the other numerals, see Figure 15.

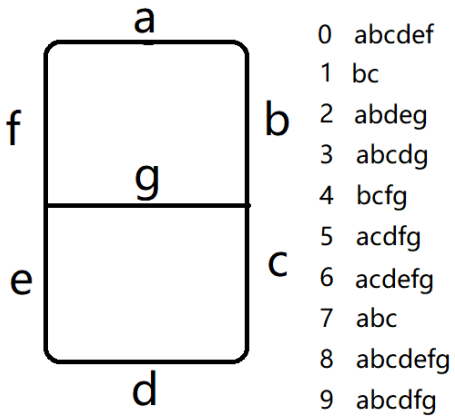


Figure 15: 7-Seg Principle

To display multiple digits, a multiplexing technique is often employed. This involves rapidly cycling through each digit position and illuminating the corresponding segments for that digit position while the others remain dark. By cycling through the digits quickly enough, the human eye perceives all digits as being simultaneously illuminated, creating the appearance of a multi-digit display.

However, on the current DE2-115 board, all the seven-segment displays are independently controlled, eliminating the need for refreshing and persistence of vision strategies for multi-digit display. This simplifies the design significantly and allows the output circuitry to be entirely combinational logic.

In the TotalScore7Seg module, the localparam keyword is used to define the seven-segment display codes for digits 0-9. These localparam variables are then assigned to control the illumination of the seven-segment display that shows the final score. In the CurrentJudge7Seg module, different judgements are displayed as follows: perfect is shown as PF, good is shown as GO, miss is shown as FL, and no_note is shown as –. In practice, the CurrentJudge7Seg module is invoked twice to display the judgements for both the upper and lower tracks.

3.9 Judgement

The main task of the Judgement module is to receive the notes to be judged and remain active during the corresponding judgement time window. It attempts to receive button inputs, evaluates the accuracy of the button inputs, outputs the judgement results, and sends a signal to the accumulator for accumulation.

We use define keyword to simplify coding, as shown in Figure 16.

3.9.1 Hardware Judgement Criteria

The hardware judgement criteria are as follows:

- For *TAP* and *HOLD_START* notes, the same judgement criteria are adopted. Within the judgement window (the time between two beats of `clk_div`), if the click's posedge falls within $\pm 1/2$ of the judgement window, it is judged as perfect. Outside the perfect window but still within the judgement window, it is judged as good. Beyond the

```
`define PERFECT 2'b00
`define GOOD 2'b01
`define MISS 2'b10
`define NO_NOTE 2'b11

`define NOTHING 2'b00
`define TAP 2'b01
`define HOLD_START 2'b10
`define HOLD_MIDDLE 2'b11

`define PUSHED 1'b0
`define NOT_PUSHED 1'b1
```

Figure 16: Macro Definitions

judgement window, it is judged as a miss. See Figure 17.

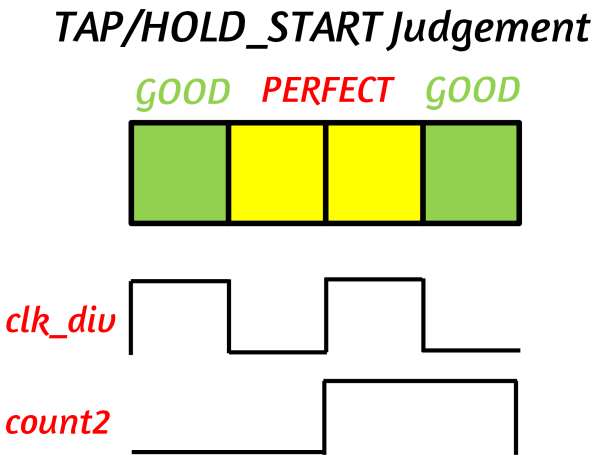


Figure 17: *TAP* and *HOLD_START* Judgement Criteria

- For *HOLD_MIDDLE* notes, button signal must be continuously received during the first half of the note's duration, while the button signal in the second half is disregarded. This design is motivated by the absence of HOLD_END notes, thus necessitating consideration of the possibility of a complete hold ending with the *HOLD_MIDDLE* note. Moreover, in actual rhythm games, holds often do not require button signals to be received throughout the entire judgement window. See Figure 18.

The specific implementation of the first judgement criterion involves determining whether the actual button click rising edge occurs when `clk_div` is 0 or 1 during two `clk_div` clock cycles.

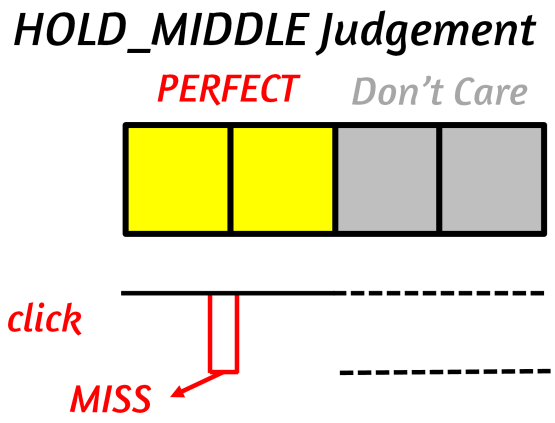


Figure 18: *HOLD_MIDDLE* Judgement Criteria

Combining this with a counter named count2, which continuously toggles between 0 and 1, we can determine which interval the click falls into for judgement. This method leverages the inherent characteristics of clk, thereby avoiding the need for complex designs based on crystal clock.

The second judgement criterion only focuses on the `clk_div` period before the note and disregards the subsequent `clk_div` period, maintaining the judgement result from the previous period in the following period.

3.9.2 Signal Judged

The `judged` signal determines whether a judgement has been completed: 1 indicates completed, and 0 indicates not completed. The signal transitions follow these conditions:

1. If the rising edge of the judgement period occurs (i.e., rising edge of `clk_div` and falling edge of count2), the signal changes to 0.

2. If the current note is *NOTHING* for three consecutive oscillator cycles (`clk` cycles), the signal changes to 0.

3. If the current note is *TAP* or *HOLD_START*, and a button rising edge is detected, the signal changes to 1.

4. If the current note is *HOLD_MIDDLE* and $count2 == 1$ (indicating the second half

of the judgement period, where *HOLD_MIDDLE* no longer needs to be monitored and only needs to be maintained), the signal changes to 1.

5. If the current note is *HOLD_MIDDLE* and $count2 == 0$ (indicating it has not yet entered the second half of the judgement period), and the button is released while the note has not been judged yet, the signal changes to 1.

The exact code is shown in Figure 19.

```
if(!rst_n) begin
    noteup_judged <= 1'b0;
end else begin
    if(cur_noteup == `NOTHING && cur_noteup_delay == `NOTHING && cur_noteup_delay2 == `NOTHING) begin
        noteup_judged <= 1'b0;
    end else if(clk_div_posedge && count2_negedge) begin
        noteup_judged <= 1'b0;
    end else if(clickup_posedge && (cur_noteup == `TAP || cur_noteup == `HOLD_START)) begin
        noteup_judged <= 1'b1;
    end else if(count2 == 1'b1 && cur_noteup == `HOLD_MIDDLE) begin
        noteup_judged <= 1'b1;
    end else if(!noteup_judged && cur_noteup == `HOLD_MIDDLE && clickup == `NOT_PUSHED && count2 == 1'b0) begin
        noteup_judged <= 1'b1;
    end else begin
        noteup_judged <= noteup_judged;
    end
end
```

Figure 19: Verilog Code of Signal Judged

Therefore, it can be seen that the `judged` signal transitions independently of the state machine (FSM) we will discuss later. The benefit of this design is that it allows the `judged` signal to be promptly communicated to other logic within the module. However, the drawback is that it increases debugging difficulty. If the FSM does not transition correctly, the incorrect result will be locked into the wrong state, leading to a cascade of subsequent errors.

3.9.3 FSM Design

In our hardware design, we efficiently integrated all judgement conditions changes using a Finite State Machine (FSM). We sampled the judgement results for each judgement interval using a flip-flop, while the FSM operated under the board's crystal clock (`clk`) to update the state.

The FSM state transition diagram is shown in the Figure 20.

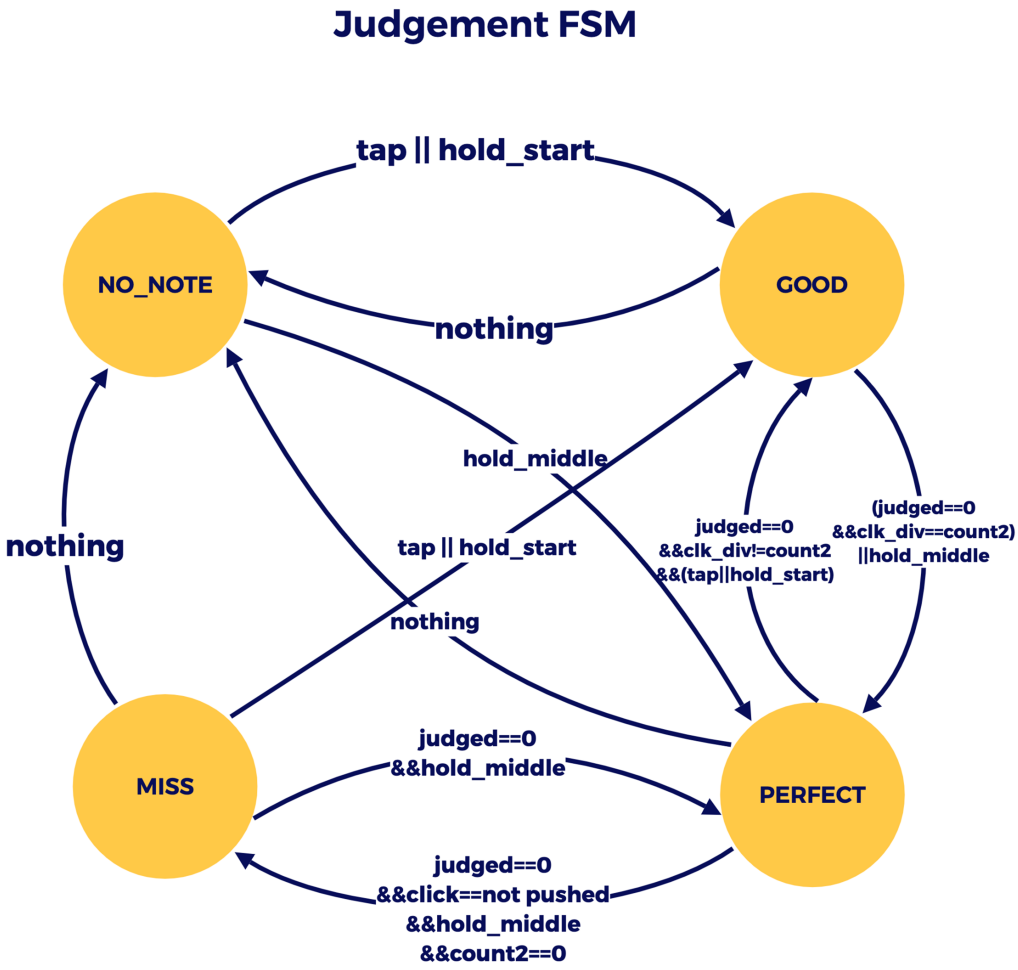


Figure 20: FSM State Transition Diagram

By default, the FSM is in the **NO_NOTE** state. For the condition of **NO_NOTE**:

1. When it detects that the current note is *TAP* or *HOLD_START*, the next state changes to **GOOD**. This is because the judgement process for these types of notes involves detecting whether the rising edge of the button input occurs when $count2 == clk\_div$ or not, to differentiate between **PERFECT** and **GOOD** judgements. Therefore, one `clk_div` period before the note arrives, the FSM must start in the **GOOD** state to listen for button input.

2. When the current note is detected as *HOLD_MIDDLE*, the next state changes to **PERFECT**. This is because the judgement for this note requires the button to be held down during the previous `clk_div` period. Thus, when monitoring starts, it is always in the **PERFECT** state.

In the **GOOD** state, the FSM continues to monitor the button input and changes the next state accordingly. For the condition of **GOOD**:

1. When the current note is *NOTHING*, it indicates that the judgement of the previous note has ended, and the next note is *NOTHING*, so the next state transitions to **NO_NOTE**.

2. If the current note is not judged, and `clk_div` equals count2, it indicates that the FSM is at the leading edge of the $\pm 1/2$ judgement interval, and it transitions to the **PERFECT** state.

3. If the current note is *HOLD_MIDDLE*, it also indicates the transition to the next interval, requiring a return to the initial **PERFECT** state for *HOLD_MIDDLE* judgement. This state transition occurs only in the judgement of *TAP* or *HOLD_MIDDLE*, so there is no need to specifically add a condition for the current note type.

In the **PERFECT** state, the FSM continues to monitor the button input and changes the next state accordingly. For the condition of **PERFECT**:

1. When the current note is *NOTHING*, it indicates that the judgement of the previous note has ended, and the next note is *NOTHING*, so the next state transitions to **NO_NOTE**.

2. If the current note hasn't been judged, the current note is *TAP* or *HOLD_START*, and `clk_div` does not equal count2, it indicates that the FSM is at the trailing edge of the $\pm 1/2$ judgement interval, transitioning to the **GOOD** state.

3. If the current note hasn't been judged, the current note is *HOLD_MIDDLE*, the button is not pressed, and count2 equals 0, it indicates that the *HOLD_MIDDLE* note was not correctly held, and the next state is set to **MISS**.

In the **MISS** state, the FSM ends the judgement of the note as **MISS**. For the condition of **MISS**:

1. When the current note is *NOTHING*, it indicates that the judgement of the previous note has ended, and the next note is *NOTHING*, so the next state transitions to **NO_NOTE**.

2. When the current note is *TAP* or *HOLD_START*, it indicates that the judgement of the previous note has ended, and the next note is *TAP* or *HOLD_START*, so the next

state transitions to the initial state of these notes, which is the **GOOD** state.

3. When the current note is *HOLD_MIDDLE*, it indicates that the judgement of the previous note has ended, and the next note is *HOLD_MIDDLE*, so the next state transitions to the initial state of this note, which is **PERFECT**.

However, during Simulation, we encountered many peculiar issues. Therefore, in the final FSM implementation, we added an extra transition rule. This rule requires that at the `clk_div` rising edge in the middle of a note's judgement interval, the FSM must return to the **NO_NOTE** state before transitioning back to its original state. This seemingly redundant operation is mainly to prevent the issue of combinational logic propagation, which we will discuss in the Discussion section.

Additionally, since this FSM cannot accurately reflect the **MISS** state when a *TAP* or *HOLD_START* judgement exceeds the allowed time, we included an extra condition at the end of each judgement interval. This condition treats any **GOOD** state with judged == 0 as a **MISS**.

3.9.4 Vivado Simulation

As the Judgement module is complex in design, we carry out behavioral simulation on Vivado to test the function and give an explicit result of our judgement implementation, as is shown in Figure 21.

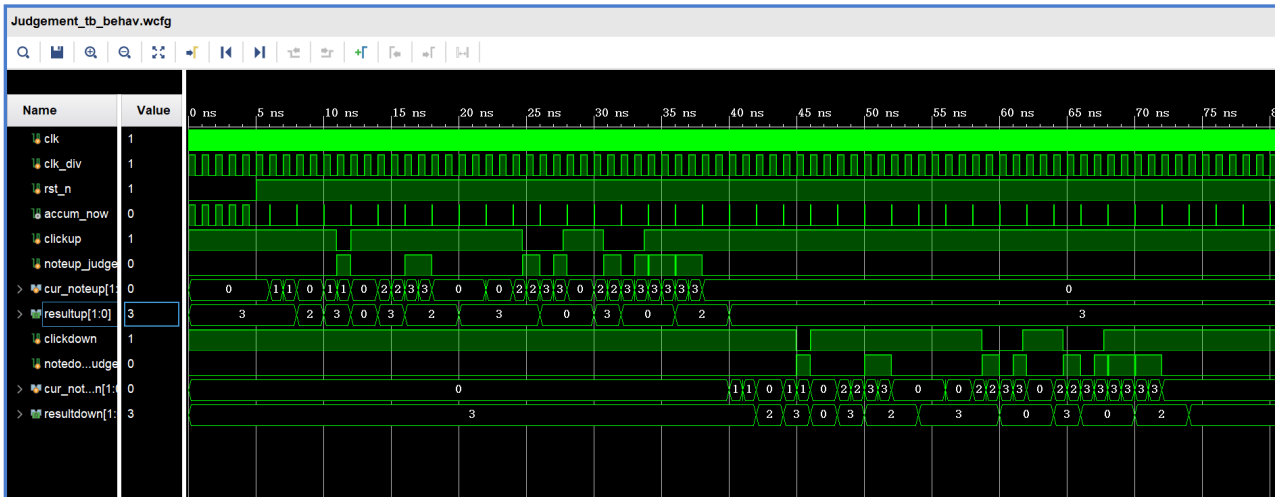


Figure 21: Vivado Simulation of Judgement Module

As we have tried the same pattern of stimulation on both of the track, including all types of notes and timings of click, we only analyze the first half of input stimulation, corresponding to the upper track. The variable `cur_noteup[1:0]` shows the current note to be judged, and can be used as a reference of the inputs. The code for each type of note can be found in section 2.1.

Specifically, we test the following 5 conditions in the testbench, and observe the results in the waveform.

1. Tap without click. Result is **MISS** (2).
2. Tap with a **PERFECT** click, which is close in timing. Result is **PERFECT** (0).
3. Hold (2 judgement intervals) without click. Result is **MISS** repeated twice.
4. Hold (2 judgement intervals) with a click kept long enough (2 judgement intervals). Result is **PERFECT** repeated twice.
5. Hold (4 judgement intervals) with a click kept not long enough (2 judgement intervals). Result is **PERFECT** repeated twice followed by **MISS** repeated twice.

We missed out **GOOD** judgement in this waveform, but it has been proved later on board. On-board results are shown in the display video.

3.10 Score Accumulation

The core function of the Score Accumulator is to perform the accumulation of 4-digit BCD codes. In terms of algorithm implementation, we made several attempts. Finally, we achieved the goal by constructing an adder specifically for BCD codes, with the 4-digit BCD adder composed of four 1-digit BCD adders cascaded together, as illustrated in Figure 22.

3.10.1 BCD Adder

The principle behind BCD code addition is that when two BCD codes are added, if the sum is equal to or less than 1001 (i.e., decimal 9), no correction is required. If the sum falls between 1010 and 1111 (i.e., hexadecimal 0AH to 0FH), a correction of 6 is required. If there is a carry generated during addition, another correction of 6 is needed. This is because the

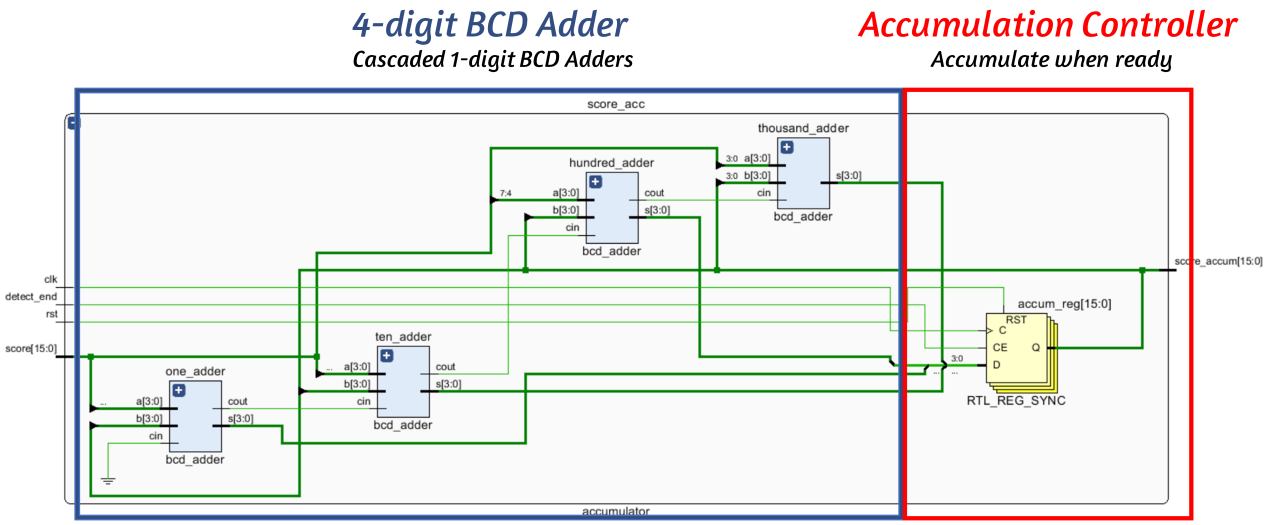


Figure 22: Score Accumulator Schematic

machine operates on binary addition, so when adding two 4-bit binary numbers, the operation is essentially following the principle of hexadecimal addition instead of decimal addition. Since 16 differs from 10 by 6, a correction of 6 is required whenever the sum exceeds 9 or a carry is generated. BCD Adder verilog code see Figure 23.

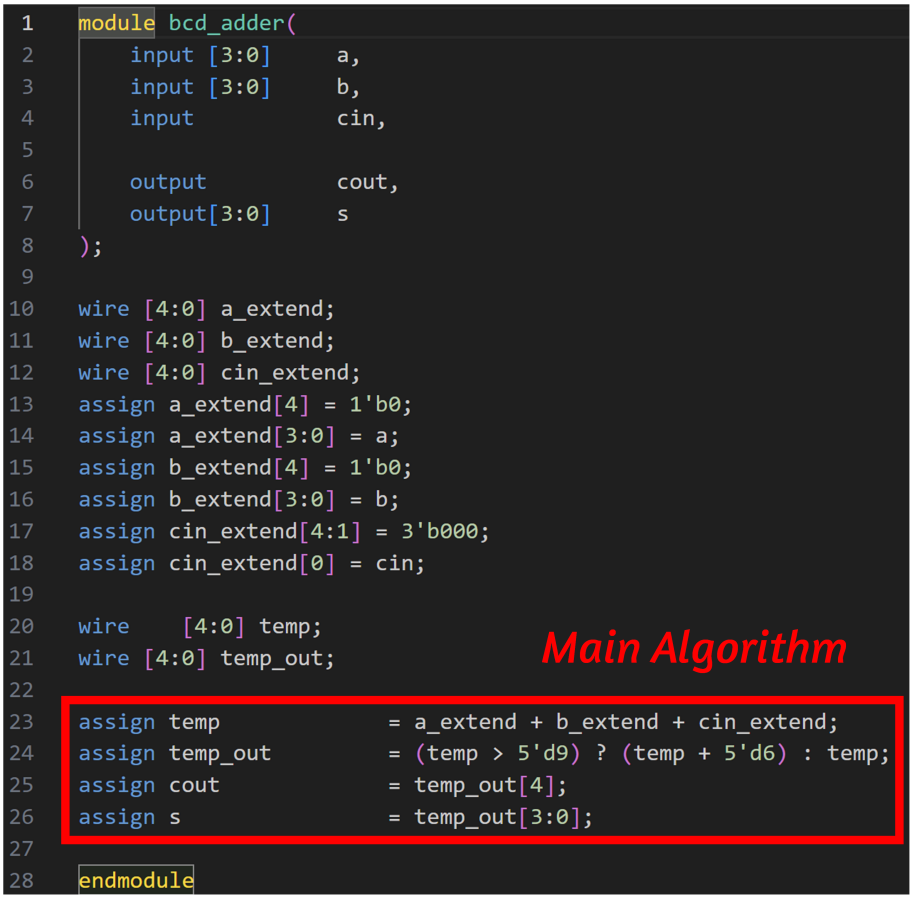
By designing 1-digit BCD adders according to the above rules and connecting the carry output and carry input of BCD addition for units, tens, hundreds, and thousands digits successively, the final calculation result is obtained. This method is also the most conducive to the synthesis of the accumulator during accumulation.

In terms of accumulation controlling, whenever receiving the `accumulate_now` signal, the accumulator performs accumulation by adding the current accumulated sum with the score or time provided by the Judgement module at that moment.

3.10.2 Pre-Addition

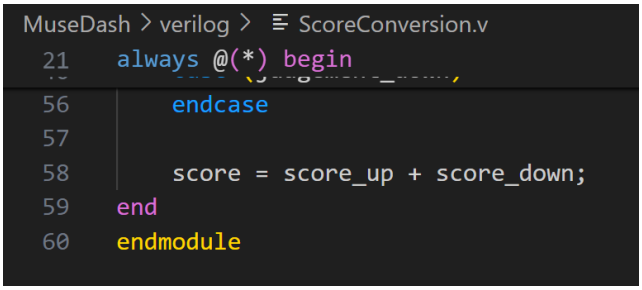
For the final score accumulation, we do not directly input the scores from the upper and lower tracks into the accumulator. Instead, we perform a preliminary addition in the ScoreConversion module before feeding the result into the accumulator for further processing, shown in Figure 24.

This approach has several advantages.




```verilog
module bcd_adder(
    input [3:0]     a,
    input [3:0]     b,
    input           cin,

    output          cout,
    output[3:0]     s
);

wire [4:0] a_extend;
wire [4:0] b_extend;
wire [4:0] cin_extend;
assign a_extend[4] = 1'b0;
assign a_extend[3:0] = a;
assign b_extend[4] = 1'b0;
assign b_extend[3:0] = b;
assign cin_extend[4:1] = 3'b000;
assign cin_extend[0] = cin;

wire    [4:0] temp;
wire [4:0] temp_out;

assign temp             = a_extend + b_extend + cin_extend;
assign temp_out         = (temp > 5'd9) ? (temp + 5'd6) : temp;
assign cout             = temp_out[4];
assign s                = temp_out[3:0];

endmodule
```

Figure 23: BCD Adder Verilog Code



MuseDash > verilog > ScoreConversion.v

```verilog
21    always @(*) begin
56        endcase
57
58        score = score_up + score_down;
59    end
60    endmodule
```

Figure 24: Pre-Addition

1. It avoids the need to design and use a three-input adder, which can complicate the design. By offloading the addition process to the ScoreConversion module, we keep the accumulator module lightweight and focused solely on accumulation tasks.

2. This modular approach facilitates easier modifications and debugging. For instance, if we need to change the scoring rules or adjust the points awarded for different judgements, these changes can be made within the ScoreConversion module without affecting

the accumulator's design. This separation of concerns simplifies the overall design and enhances the system's flexibility and maintainability.

4 Discussion

4.1 Design Techniques

Hardware issues are not as easy to diagnose and resolve as software issues, so it is crucial to read the hardware manual or development guide carefully before attempting any prototyping.

When designing combinational logic, one should pay particular attention to avoiding the generation of latches. When using if-else statements, ensure that all possible conditions are covered; for case statements, a default branch should be added. This prevents uncertain outputs and thus the generation of latches. Latches are complex for static timing analysis and are usually not intentionally part of the design, so they should be avoided whenever possible.

In certain cases, the use of assign statements can replace if-else or case statements. if-else and case statements are turned into priority logic circuits during synthesis, which cannot propagate indeterminate signals (X-state). In contrast, assign statements are used for continuous assignments, avoiding the issue of priority logic, benefiting both the synthesis and simulation of the code.

Naming conventions are crucial for the readability and maintainability of code. For instance, when naming reset signals, `rst` could indicate an active-high reset, whereas `rst_n` might represent an active-low reset. When writing code, one should consider corresponding port names as part of naming variables and ports. With multiple people collaborating, a unified naming convention is especially helpful for communication and teamwork, reducing misunderstandings.

Good annotation is another important means of enhancing code readability. Annotations should clearly and concisely explain the functionality and design intentions of the code, which is beneficial not only for team collaboration but also for individual understanding and maintenance in the future. Annotations should be present throughout all parts of the code, particularly in complex or unintuitive logic sections.

In sequential logic parts, it is good practice to use non-blocking assignments (<=) rather than blocking assignments (=). Non-blocking assignments ensure that all assignments occur simultaneously within one clock cycle, preventing race and hazard conditions and indeterminate behavior due to the order of assignments. Blocking assignments are primarily used in combinational logic and should be utilized cautiously to avoid introducing timing issues.

4.2 Judgement Design

The design of the Judgement FSM has undergone several revisions. Initially, the design featured two separate FSMs: one dedicated to monitoring and another to generating results. This approach proved to be overly complex and posed significant challenges in verifying its functionality. Consequently, the design was simplified to use a single FSM to handle the judgement process. Discarded FSM design see Figure 25 and 26.

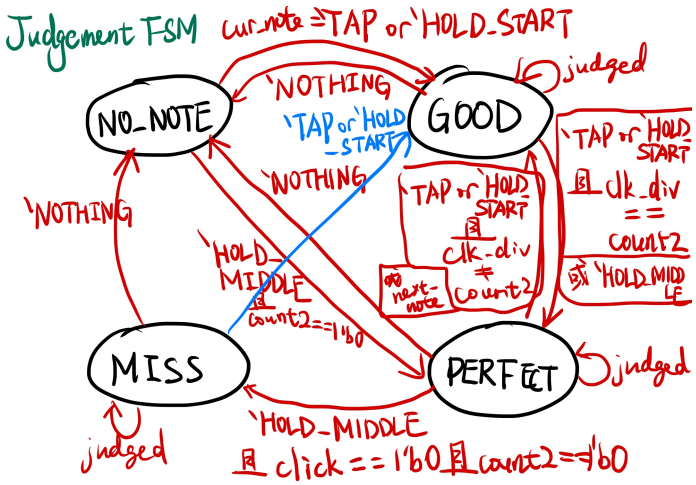


Figure 25: Discarded FSM Design 1

Despite this simplification, initial attempts to directly test the FSM on the hardware without prior simulation yielded unsatisfactory results. Realizing the necessity of thorough validation, we proceeded to perform functional verification of the Judgement module using simulation before testing it on the board. However, even with this step, Vivado's functional verification could not entirely cover all potential error scenarios. Some issues only became apparent during the on-board testing, despite the simulations indicating correct functionality.

In such cases, the process of fixing issues often involved applying patches to address specific

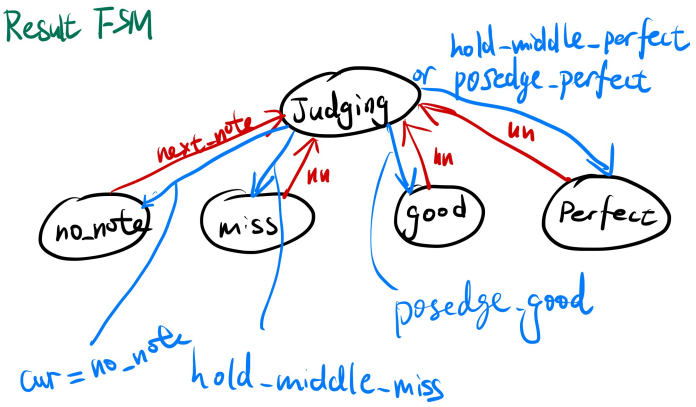


Figure 26: Discarded FSM Design 2

problems, rather than identifying and resolving the root cause of the errors. This method, while sometimes necessary, highlights the limitations of simulation and the complexity of ensuring reliable functionality in hardware design.

4.3 Edge Capture

In the process of designing sequential logic, we encountered a perplexing issue related to capturing the rising edge of signals. Since it's impossible for a single signal to be triggered by two different clock signals simultaneously in hardware, we needed to define a primary clock and use combinational logic to generate a rising edge pulse signal from another clock to facilitate detection.

The first problem pertains to the generation of the rising edge pulse signal. Strangely, using standard sequential logic to delay a signal by one clock cycle often failed to work as expected. This issue persisted in both Simulation and Verification, where we couldn't achieve the desired results. Consequently, in this project, some rising edge pulse signals are delayed by one cycle, while others are delayed by two cycles. This determination is based purely on whether the Simulation and Verification results are correct. It's an empirical, trial-and-error method with little theoretical basis. See Figure 27.

The second problem involves the integration of sequential and combinational logic. This integration also required thorough Simulation and Verification to determine the correct approach.

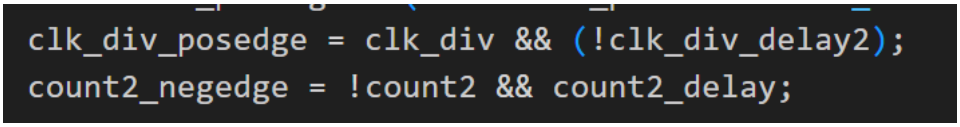


Figure 27: Delay 1 clk or Delay 2 clks ?

Only through iterative testing and modification were we able to select the appropriate method to combine these logic types effectively. See Figure 28.

Figure 28: Use edge or not ?

4.4 Signal Penetration

During the design verification process, we encountered a peculiar phenomenon known as signal penetration. Initially, our `cur_note` signal used combinational logic for input selection. Specifically, when $count2 == 0$, `cur_note` would select `right_note`, and when $count2 == 1$, it would select `left_note`. However, in Simulation, we noticed that this switching process led to a glitch in the `cur_note` input.

This glitch occurred because `count2` and the external inputs `left_note` and `right_note` were all controlled by the `clk_div` clock. Due to the timing differences and race conditions between these signals, a transient glitch appeared in the `cur_note` input. The issue was exacerbated because `cur_note` was sampled using the main clock `clk`, which arrived earlier than `clk_div`. As a result, the `cur_note` signal would erroneously use the `right_note` value for one clock cycle ahead of the intended time.

If `right_note` was simply **NO_NOTE**, this would not have been problematic, as **NO_NOTE** has an inherent ability to detect the current note type and transition to the appropriate state. However, for hold notes, where *HOLD_START* is immediately followed by *HOLD_MIDDLE*, this glitch caused *HOLD_MIDDLE* to prematurely propagate through, leading to irreversible state changes. This propagation resulted in incorrect state transitions that could not be corrected, making it essential to resolve this issue.

To address this issue, I deliberately enforced a state transition to **NO_NOTE** in the FSM at the midpoint of the judgment interval during the `clk_div` rising edge. This intentional mistake allowed the system to reset briefly. Since **NO_NOTE** is designed to transition correctly back to the appropriate state, this redundant operation effectively masked the propagation problem, ensuring the correct logic was implemented. By introducing this intermediate **NO_NOTE** state, we managed to circumvent the glitch caused by signal propagation and maintain proper functionality of the FSM. As shown in Figure 29 .

```
if(count2 == 1'b1 || (count2_negedge && !clk_div_posedge)) begin
    cur_noteup <= left_noteup;
    cur_notedown <= left_notedown;
end else if(clk_div_posedge) begin
    cur_noteup <= `NOTHING;
    cur_notedown <= `NOTHING;
end else begin
    cur_noteup <= right_noteup;
    cur_notedown <= right_notedown;
```

Figure 29: Addressing Signal Penetration

4.5 Hardware-Software Inconsistency

The inconsistency between software simulations and hardware implementation also puzzled us. During the Post-Implementation Timing Simulation in Vivado, everything functioned correctly. However, when we proceeded to on-board testing, various issues emerged. These discrepancies highlighted the challenges of ensuring that a design works as intended in a real hardware environment, despite appearing flawless in a simulated one.

1. The problem of detecting rising edges led to further complications, where the FSM would get stuck in the **NO_NOTE** state and fail to return to the appropriate state in the hardware implementation.

2. The button only responds correctly in *HOLD_MIDDLE*, but it fails to respond for TAP or *HOLD_START*.

Since I couldn't identify the root cause of the problem, I had to forcefully add patches, such as making the FSM transition to the **NO_NOTE** state if it detects three consecutive cycles

of the note being *NOTHING*. In retrospect, Quartus likely has simulation methods as well, and using Vivado might have been a mistake. See Figure 30.

```
if(cur_noteup == `NOTHING && cur_noteup_delay == `NOTHING && cur_noteup_delay2 == `NOTHING) begin
    next_state_up = `NO_NOTE;
```

Figure 30: Forcing **NO_NOTE**

4.6 Quartus Debugging

When assigning pins in Quartus, it's important to ensure that each pin should only correspond to one signal. We started by the pin assigner to manually assign pins, but later turned to `csv` files to import pin assignments. Specific pin details can be found in the DE2-115 manual.

After pin assignment, we can go through several debugging steps to get a wavefile showing the real time signals on board.

First, in the Signal Configuration, select the clock as the sampling basis and set a large sampling depth (Figure 31). For instance, a depth of 2K indicates a total sampling duration of 2K. The number of samples (segments) can then be decided based on the sampling depth.

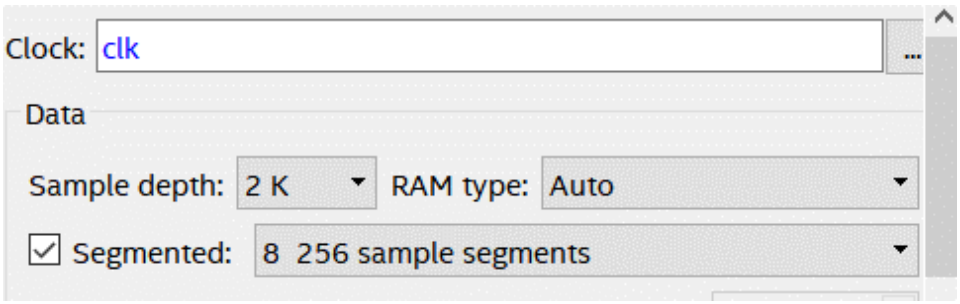


Figure 31: Clock Configuration Settings

Then, in the Setup, select the filter as Post-Compilation, and choose the signals to be monitored within the main module or sub-module (Figure 32). Then select the trigger conditions according to your debugging needs. Choose BASIC AND if sampling should occur only when all trigger conditions are met, or BASIC OR if sampling should occur when any one of the conditions is met. Each signal can have different trigger conditions, such as rising edge, falling

edge, or both edges. Once configured, you can use the Group option to group a set of signals together to view their overall values, like in Vivado.

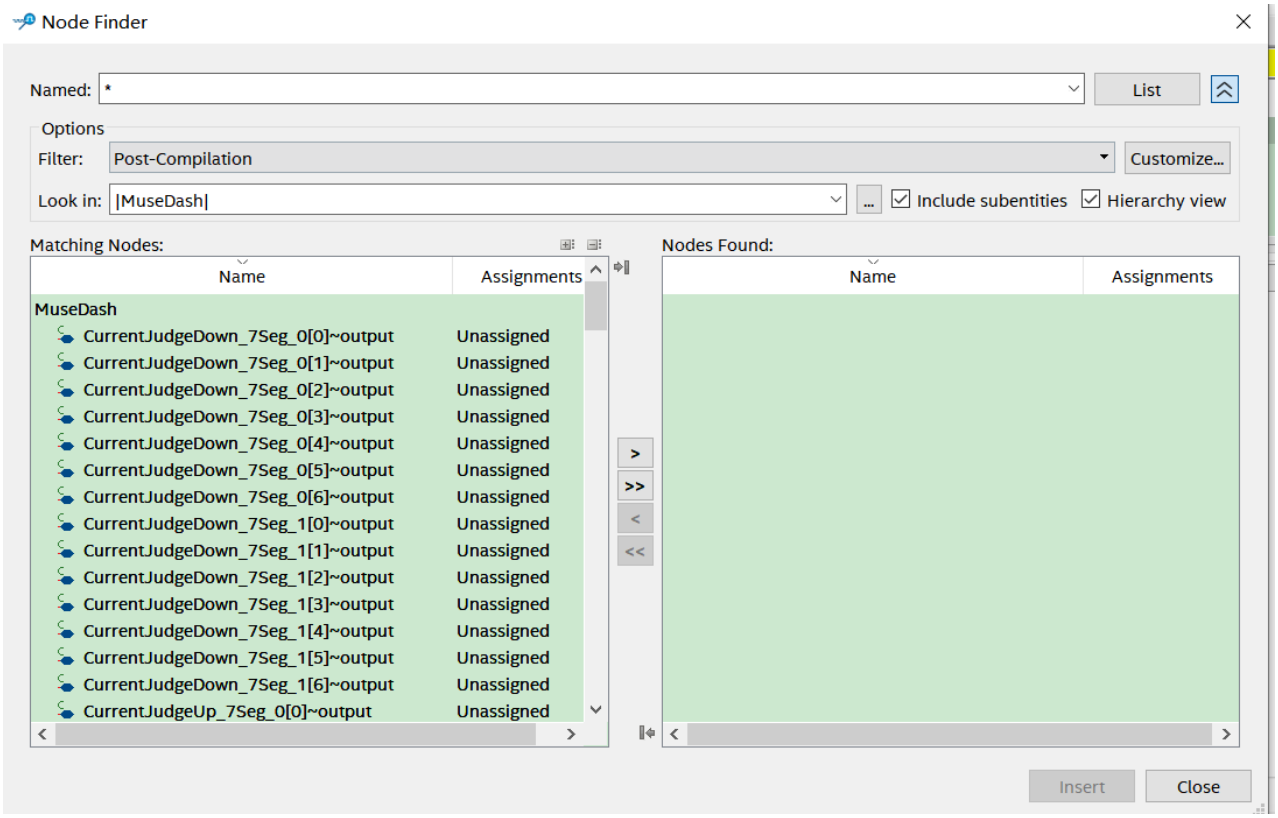


Figure 32: Signal Settings

After connecting to the board, you can choose single detection or continuous detection (Figure 33). When useful signal changes are identified, use the save button next to Data Log to save the waveform and signal configuration of a single instance. This allows you to reopen and compare the data directly next time.

4.7 HLS Trial

High-Level Synthesis (HLS) is the process of converting an algorithm described in a high-level programming language (such as C++) into a hardware description language (like Verilog). HLS automates the mapping of software code to hardware circuits, making hardware design more efficient and accessible. The primary steps in HLS are:

1. Behavioral Description: The algorithm is written in a high-level language (e.g., C++)

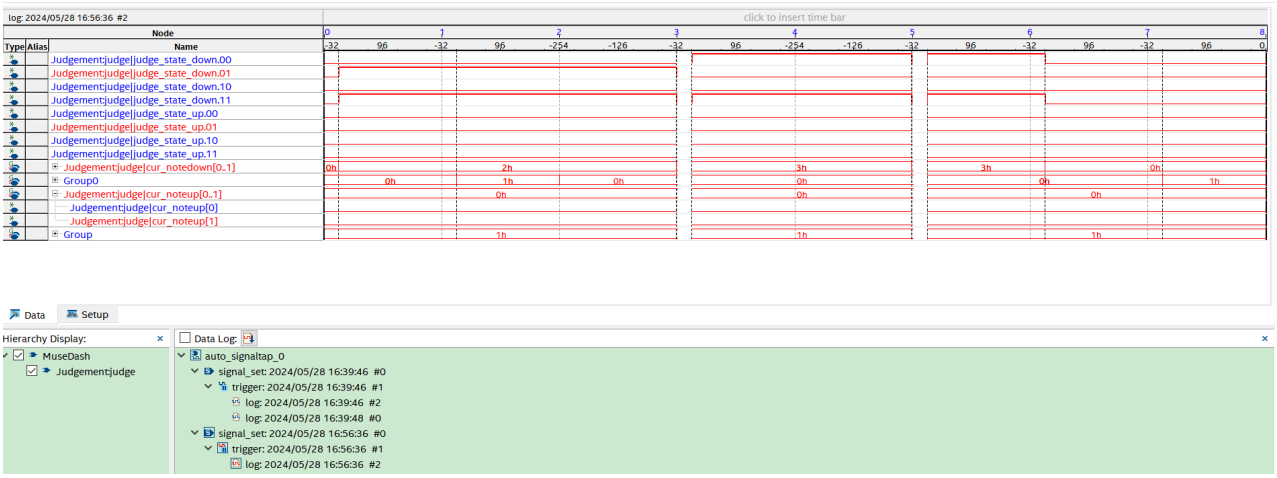


Figure 33: Waveform Save

to describe the system's functionality and behavior.

2. Synthesis: HLS tools convert the high-level code into an intermediate representation (IR). The IR is then optimized using various techniques (e.g., loop unrolling, pipelining).

3. Hardware Mapping: The optimized IR is mapped to hardware resources (e.g., adders, multipliers), generating Register-Transfer Level (RTL) hardware description code (such as Verilog).

4. Verification and Optimization: Simulation tools verify the functionality and performance of the generated RTL code. Further optimizations are performed to meet specific design constraints (such as area, power, latency).

By using HLS, designers can focus on algorithm development in high-level languages while leveraging automated tools to handle the intricate details of hardware implementation, thus bridging the gap between software and hardware design.

We have now completed the transformation from LLVM-like Intermediate Representation (IR) to Verilog code. During this process, we have constructed data flow graphs and control flow graphs based on the LLVM IR. Subsequent stages involved register allocation and logic synthesis, all aimed at optimizing resource usage and ensuring timing requirements are met. This series of steps successfully translated the initial IR into the intended Verilog code. In the future, our efforts will be dedicated to achieving the transformation from C++ code

to LLVM-like IR and also towards further refining and enhancing our High-Level Synthesis (HLS) tool. Our goal is to elevate the tool's capabilities for higher levels of automated design sophistication.

5 Conclusion

In the embedded system lab, we successfully implement a rhythm game on the Altera (Intel) DE2-115 FPGA board. Inspired by Muse Dash, our game design implements most of the original game's note and mechanisms through software and hardware co-designs. On the hardware side, we design a system of judgment mechanisms, along with other hardware computation and display functions. On the software side, we support customized music chart design and automated ROM hardware generation. The game uses buttons as input and textLCD display as output, while judgement and score results are displayed on 7-segment display, effectively recreating the original game's gameplay experience.

6 Future Works

The overall design is successful, but a lot could still be done to improve the gameplay of the design. We list a series of potential improvements for future works.

1. The original Muse Dash game integrates a background music for each of the chart, and uses keyboard as standard input, which we haven't achieved in the current project. We will continue to explore the way for the FPGA board to play music, and support keyboard input and VGA output in the future.

2. The original Muse Dash game allows players to select from hundreds of music and charts. We will continue to support multiple music and charts in the future, and complete the interface for players to select music, and see the brief information for each of the music and charts.

3. In the current project, our software design is mainly used to generate a custom ROM file that is later simply integrated into the hardware design. While it works for small project with few charts, it becomes unfeasible when it comes for further expansion. We

will try to upgrade our software to write data into on-board memory devices instead of custom memory, in order to make better use of on-board resources and release the potential of embedded system design.

4. In the current project, our hardware design uses a debouncement module to eliminate noise from input buttons. However, it is known that the filtered input is postponed for a fixed delay, compared to the ideal button input. The delay results from the counter in the debouncement module and is usually set to be more than 20ms. This delay is considerable enough for a rhythm game, which usually sets the range of PERFECT judgement no more than $\pm$ 50ms. We will consider this delay in the hardware judgement part for future game optimization.