Digital Electronics Lab 5 Report

Ming Gong (mg4264), Xuanyi Wu (xw3036)

Design

User Interface

The user interface has 8 switches (`sw[7:0]`) that are used to record notes, accompanied with `pb[3:0]` . This is encoded to `note_sel` in the "User Interface" section of `piano.vhd`
At the rising edge of `CLK` , the hardware uses a lookup table to translate the values from `switch` to the note encoding `note_sel`

- If multiple, or no switches are active, it's translated as `0000` (no note)

```
case switch is
    when "10000000" => note_sel <= "0001"; -- C
    when "01000000" => note_sel <= "0011"; -- D
    when "00100000" => note_sel <= "0101"; -- E
    when "00010000" => note_sel <= "0110"; -- F
    when "00001000" => note_sel <= "1000"; -- G
    when "00000100" => note_sel <= "1010"; -- A
    when "00000010" => note_sel <= "1100"; -- B
    when others     => note_sel <= "0000";
end case;
```

When the modifier switches are active, the `note_sel` encoding is incremented or decremented accordingly.

```
-- Sharp -- Add one.  PB(3) is the octave key.
if (PB(2) = '1') then
    note_next <= PB(3) & note_sel + 1;
-- Flat --  Minus one.
elsif (PB(1) = '1') then
    note_next <= PB(3) & note_sel - 1;
else
    note_next <= PB(3) & note_sel;
end if;
```

- If `PB(2)` or `PB(1)` is active, `note_sel` is incremented or decremented, respectively
- `PB(3)` is concatenated with the original `note_sel`.

The final `note_sel` will be 5 bits.

- The MSB encodes the octave number. If the MSB is 0, it's in 3; if the MSB is 1, it's in 4
- The LSBs encode the note name and accidental sequentially. `0001` maps to "C3", and `1100` maps to "B3".
  Below is a table that summarizes the pattern:

`PB(3)`	`PB(2)`	`PB(1)`	`switch`	`note_sel`	**Note**
0	0	0	10000000	00001	C3
0	1	0	10000000	00010	C3#
0	0	0	01000000	00011	D3
0	1	0	01000000	00100	D3#
1	0	0	10000000	10001	C4

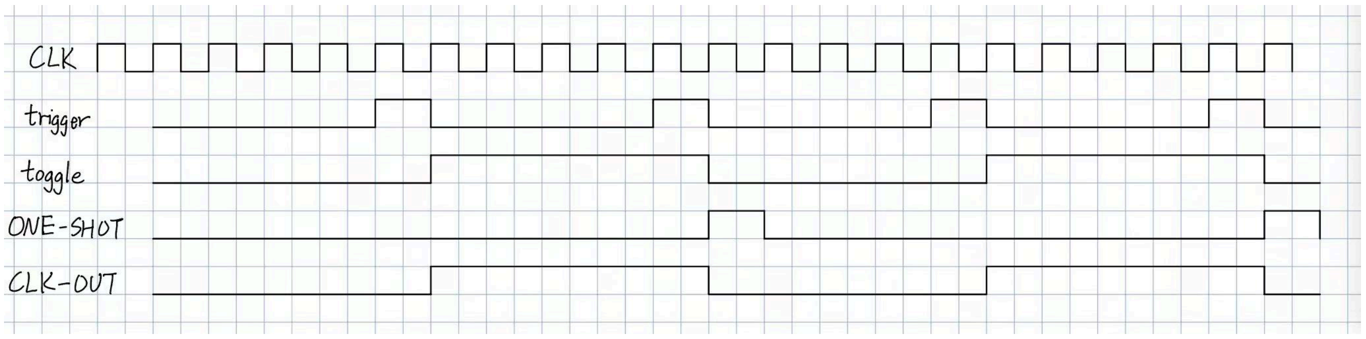
Clock Divider

In the `counter` section, `trigger` is only set to 1 at the next cycle when `count == DIV`.

An active `trigger` controls more signals in the `toggle_trigger` section in the **next** cycle

- `toggle` flips its value
- When `toggle = 1`, `ONE_SHOT` is set to 1 for a cycle
  In the `output` section, `CLK_OUT` also flips its value at the next cycle when `trigger = 1`

The diagram below shows the signal timing diagram for `DIV = 5` (dec). In reality, there will be 500 cycles between each change of all internal and output signals.



The clock frequency in the diagram is drawn 10 times slower for clarity
The diagram assumes that each variable is 0-initialized. The waves may look different if initialized differently. Regardless of the nuances of cycle delay, the results are:

- `CLK_OUT` toggles every 50 cycles of `CLK` between 1 and 0
- `ONE_SHOT` goes high for one `CLK` cycle, and then stays low for 99 cycles.
  Both have a frequency of 1 MHz (divided by 50 * 2 = 100).

Note divider

`note_gen.vhd` further divides the 1 MHz clock from `clk_dvd.vhd`. Here, the division cycles are determined from a lookup table for a specific note.

$$f_{out} = \frac{1\ \text{MHz}}{2\texttt{DIV}}$$

```
case NOTE_IN is
    when "00000" => next_div <= x"0000";
    when "00001" => next_div <= x"0EEE"; -- C3
    when "00010" => next_div <= x"0E18"; -- C3#
    when "00011" => next_div <= x"0D4E"; -- D3
    -- ...
```

Below are the resulting frequencies of each note in the lookup table. They are very close to the actual frequency.

Note	`DIV` (hex)	`DIV` (dec)	f_{out} (Hz)	Actual frequency (Hz)
C3	0EEE	3822	130.82	130.813
C3#	0E18	3608	138.58	138.591
D3	0D4E	3406	146.80	146.832
D3#	0C8E	3214	155.57	155.563
E3	0BDA	3034	164.80	164.814
F3	0B30	2864	174.58	174.614
F3#	0A8E	2702	185.05	184.997
G3	09F7	2551	196.00	195.998
G3#	0968	2408	207.64	207.652
A3	08E1	2273	219.97	220.000
A3#	0861	2145	233.10	233.082
B3	07E9	2025	246.91	246.942
B3#	0777	1911	261.64	261.626
C4b	07E9	2025	246.91	246.942
C4	0777	1911	261.64	261.626

Note	`DIV` (hex)	`DIV` (dec)	f_{out} (Hz)	Actual frequency (Hz)
C4#	070C	1804	277.16	277.183
D4	06A7	1703	293.60	293.665
D4#	0647	1607	311.14	311.127
E4	05ED	1517	329.60	329.628
F4	0598	1432	349.16	349.228
F4#	0547	1351	370.10	369.994
G4	04FB	1275	392.16	391.995
G4#	04B4	1204	415.28	415.305
A4	0470	1136	440.14	440.000
A4#	0431	1073	465.98	466.164
B4	03F4	1012	494.07	493.883

7-Segment Display State Machine

`seven-seg.vhd` takes an encoded note and displays it on the off-chip 4-digit 7-segment display.

1. Convert encoded notes to hex digits

```
case NOTE_IN is
    when "00000" => seg_buf <= "1010" & "1010" & '0'; -- AA
    when "00001" => seg_buf <= "1100" & "0011" & '0'; -- C3
    when "00010" => seg_buf <= "1100" & "0011" & '1'; -- C3#
    -- ...
```

An encoded note `NOTE_IN` is converted to hex digits for 7-seg to display. Each digit and the DP (for accidental) are concatenated and stored in `seg_buf`.

- For example, note `00001` maps to "C3#". The output in `seg_buf` is:
  - `1100` for "C"
  - `0011` for "3"
  - `1` for "#" (DP)
- If an encoding is invalid, `seg_buf` will be set to "AA" or "AA."

2. Scan through the digits

The 7-segment has 4 digits. They are updated by taking turns.

```
    if (SCAN_EN = '1') then
         cur_dig <= cur_dig + 1;
    end if;
```

When `SCAN_EN` is active, `cnt_cur_dig` cycles through `cur_dig` from `00` (LS) to `11` (MS)

3. Load a specific digit

```
case cur_dig is
    when "00" =>
        DIGIT <= "1110";
        digit_en <= '1';
        digit_now <= seg_buf(4 downto 1);
        point_now <= not(seg_buf(0));
    when "01" =>
        DIGIT <= "1101";
        digit_en <= '1';
        digit_now <= seg_buf(8 downto 5);
        point_now <= '1';
    -- ...
```

When a digit is scanned, buffers `digit_now` and `point_now` will be set to the corresponding sections of `seg_buf`. Note that

- The first two digits (`10` and `11`) are unused, with `digit_en = 0`
- `DIGIT`, the output that directly enables the 7-segment digit's anode, is set here
- `DIGIT` and `point_now` are active low.
- `point_now` can only be active after the last digit

4. Decode hex digit to 7-segment

```
if (digit_en = '1') then
    case digit_now is
        when "0000" => seg <= "0000001" & point_now ; -- 0
        when "0001" => seg <= "1001111" & point_now ; -- 1
        when "0010" => seg <= "0010010" & point_now ; -- 2
        -- ...
```

This is the standard hex-to-7-seg decoder lookup table. The output `seg` (active low) is concatenated from the digit segments and DP. It then directly goes to the cathodes, writing to the active digit of the 7-segment display.

This process repeats for all digits (enabled or disabled)

Implementation

Test cases

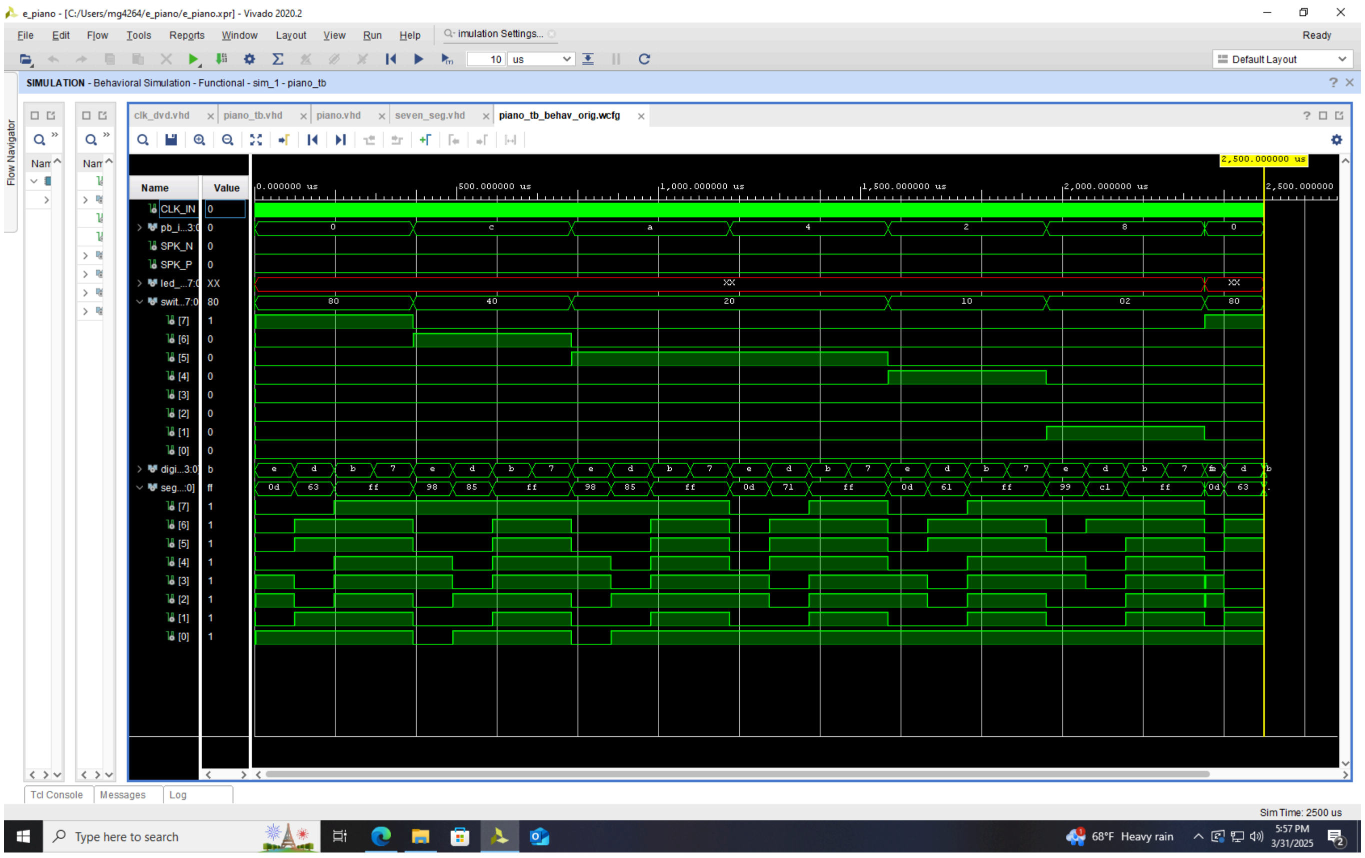
We wrote a simulation test for various inputs in the Test Bench section.
Cases tested:

- Different values for `switch_in`
- `pb[0]` : reset
- `pb[1]` : sharp
- `pb[2]` : flat
- `pb[3]` : higher octave

```
-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
    -- System Reset
    pb_in(0) <= '1';
    wait for 400 ns;

    -- input C3 output C3
    pb_in <= "0000";
    switch_in <= "10000000";
    wait for 392 us;

    -- input D3 output D4#
    -- pb_in raise key and one octave
    pb_in <= "1100";
    switch_in <= "01000000";
    wait for 392 us;

    -- input E3 outputs D4#
    -- pb_in raise key,one octave lower
    pb_in <= "1010";
    switch_in <= "00100000";
    wait for 392 us;

    -- input E3 outputs F3=E3#
    -- pb_in stay key, one octave higher
    pb_in <= "0100";
    switch_in <= "00100000";
    wait for 392 us;

    -- input F3 outputs E3
    -- pb_in stay key, one octave lower
    pb_in <= "0010";
    switch_in <= "00010000";
    wait for 392 us;

    -- input B3 outputs B4
    -- pb_in raise key and octave unchanged
    pb_in <= "1000";
    switch_in <= "00000010";
    wait for 392 us;
END PROCESS;
-- *** End Test Bench - User Defined Section ***
```

Simulation

Below is the simulation result. We'll analyze the first test case (C3) in detail. The other cases follow similar ideas



```
-- input C3 output C3
pb_in <= "0000";
switch_in <= "10000000";
wait for 392 us;
```

The inputs are: `pb_in` = 0 (0000), `switch_in` = 80 (10000000)

- Specifically, `switch_in[7]` is 1, and the rest are 0
  The module calculates the output at `digit_out` and `seg_out` and updates them sequentially.
- `led_out = XX`, since we have not defined them
- `digit_out` cycles between e (1110, Digit 0), d (1101), b (1011), 7 (0111, Digit 3)

`digit_out`	`digit_out` (bin)	Digit place	`seg_out`	`seg_out` (bin)	Digit
e	1110	Digit 0	0d	00001101	3
d	1101	Digit 1	63	01100011	C
b	1011	Digit 2	ff	11111111	
7	0111	Digit 3	ff	11111111	

The next few inputs are for D4#, D4#, F3, E3, B4, using the same analysis.

- When `seg_out[0]` = 0, `DP` is displayed (for #)