# Digital Circuit Lab — Final Project

## 自動音符辨識播放器

## (Automatic Musical Note Player)

Team: 06

Member: 劉容均 b05901084、吳映葦 b05901100、鄧宇凡 b05901183

- **Introduction**

When playing instrument, we often play the wrong note due to incorrect recognition of the music score from time to time. Annoyed by the inharmonic sounds, we came up with the idea “What if we let machines do the recognition for us?” As a result, in this project we will integrate image and audio processing to implement an automatic music player which can identify different musical notes.

- **Hardware**: Altera DE2-115 FPGA board
- **Software**: Quartus II
- **HDL (Hardware Description Language)**: System Verilog
- **Equipments**:


TRDB-D5M
(5 Mega Pixel Digital Camera)

40-Pin GPIO Cable

Speaker

Screen

VGA to VGA Cable

## Part I: User Manual

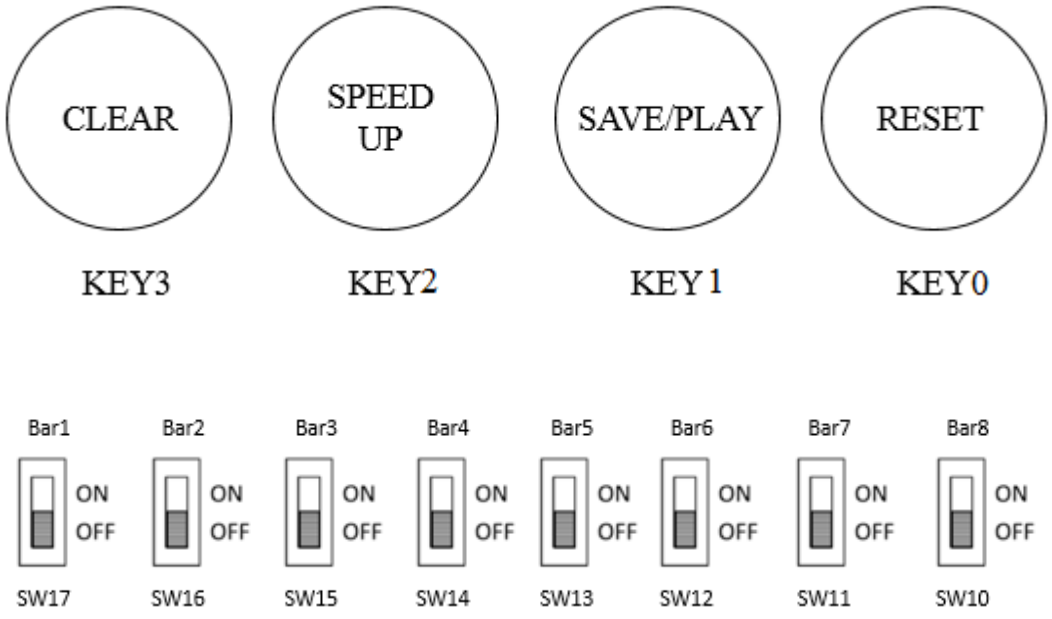
- **Usage**
  - Connect your camera and speaker (or earphone) to the FPGA board.
  - Press KEY3 to clear SRAM content.
  - Save 8 Notes as a Musical Bar

1. Press KEY0 to reset.
2. Switch SW0 downward to the RECORD mode.
3. Place magnets at the position of wanted notes.
4. Check if the played piano sounds are as expected.
5. Press KEY2 to change the playing speed.
6. If the bar is correctly played, switch the SW of the corresponding bar upward.
7. Press KEY1 to save the musical bar.

  - Play the Whole Melody

1. Save at most 8 musical bars on the FPGA board.
2. Switch the SW of the corresponding bar you want to play upward.
3. The corresponding LED will reveal which bars are activated.
4. Switch SW0 upward to the PLAY mode.
5. Press KEY1 to start playing the melody you stored on the FPGA board.

- **Illustration**

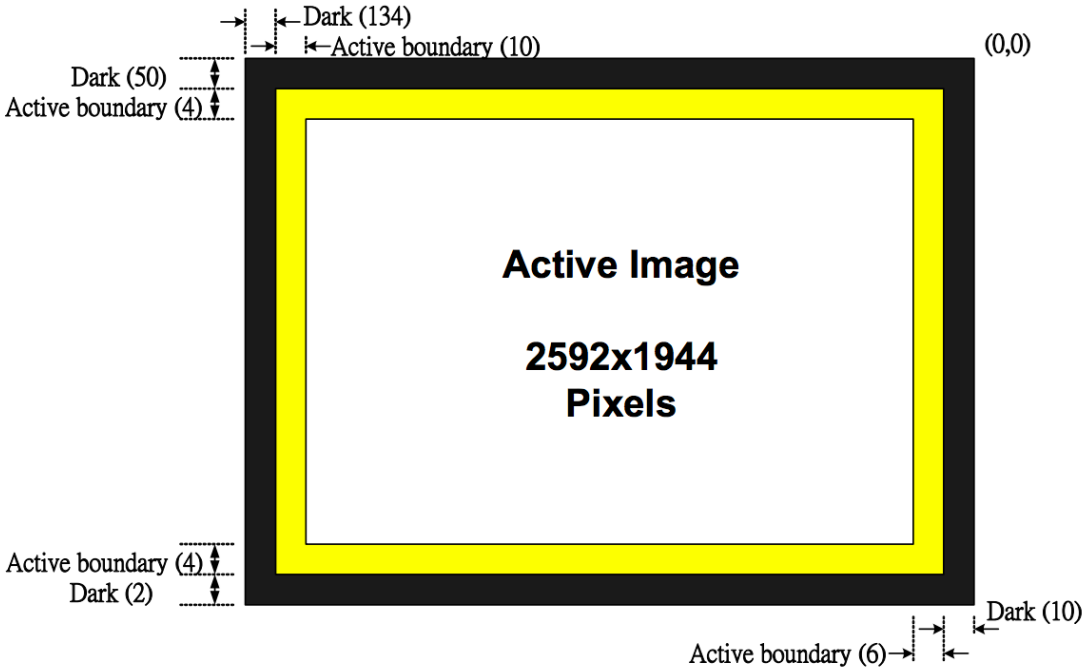


# Part II: Tutorial

- **Introduction of Protocols**

  **- TRDB-D5M Camera**

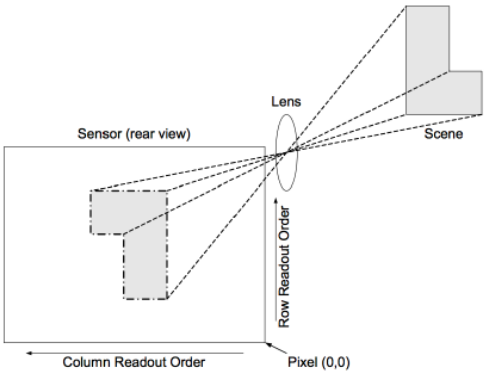
  TRDB-D5M is a camera module provided by Terasic for FPGA board and it can be connected to FPGA via GPIO ports. D5M pixel array consists of a matrix with 2004-row by 2752-column and is addressed by row and column. The address (row 0, column 0) represents the upper-right corner of the entire array as shown in the figure down below.

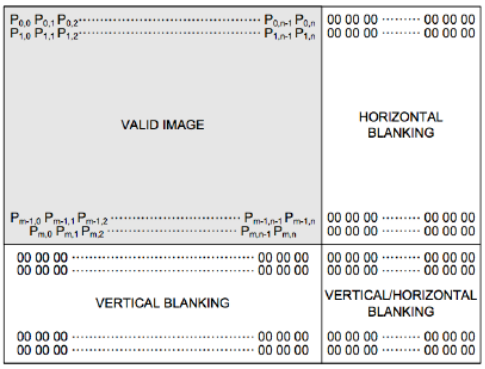



Boundary regions can be used to avoid edge effects when performing color processing to achieve a image with 2592 × 1944 resolution. Optically black columns and rows can be used to monitor the black level.

| Column | Pixel Type |
|---|---|
| 0–9 | Dark (10) |
| 10–15 | Active boundary (6) |
| 16–2,607 | Active image (2592) |
| 2,608–2,617 | Active boundary (10) |
| 2,618–2,751 | Dark (134) |

Readout order as well as output data format is described in the figures below:




Some other useful information about TRDB-D5M is listed down below just for references:

| Resolution | Frame Rate | Sub-sampling Mode | Column_Size (R0x04) | Row_Size (R0x03) | Shutter_Width_Lower (R0x09) | Row_Bin (R0x22 [5:4]) | Row_Skip (R0x22 [2:0]) | Column_Bin (R0x23 [5:4]) | Column_Skip (R0x23 [2:0]) |
|---|---|---|---|---|---|---|---|---|---|
| 2592 x 1944 (Full Resolution) | 15.15 | N/A | 2591 | 1943 | <1943 | 0 | 0 | 0 | 0 |
| 2,048 x 1,536 QXGA | 23 | N/A | 2047 | 1535 | <1535 | 0 | 0 | 0 | 0 |
| 1,600 x 1,200 UXGA | 35.2 | N/A | 1599 | 1199 | <1199 | 0 | 0 | 0 | 0 |
| 1,280 x 1,024 SXGA | 48 | N/A | 1279 | 1023 | <1023 | 0 | 0 | 0 | 0 |
| | 48 | skipping | 2559 | 2047 | | 0 | 1 | 0 | 1 |
| | 40.1 | binning | 2559 | 2047 | | 1 | 1 | 1 | 1 |
| 1,024 x 768 XGA | 73.4 | N/A | 1023 | 767 | <767 | 0 | 0 | 0 | 0 |
| | 73.4 | skipping | 2047 | 1535 | | 0 | 1 | 0 | 1 |
| | 59.7 | binning | 2047 | 1535 | | 1 | 1 | 1 | 1 |
| 800 x 600 VGA | 107.7 | N/A | 799 | 599 | <599 | 0 | 0 | 0 | 0 |
| | 107.7 | skipping | 1599 | 1199 | | 0 | 1 | 0 | 1 |
| | 85.2 | binning | 1599 | 1199 | | 1 | 1 | 1 | 1 |
| 640 x 480 VGA | 150 | N/A | 639 | 479 | <479 | 0 | 0 | 0 | 0 |
| | 150 | skipping | 2559 | 1919 | | 0 | 3 | 0 | 3 |
| | 77.4 | binning | 2559 | 1919 | 3 | 3 | 3 | 3 | 3 |

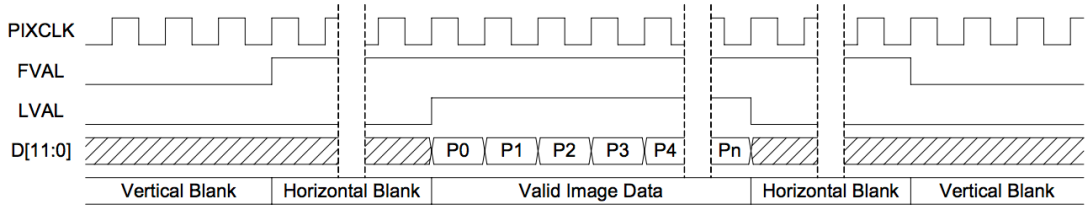
| Parameter | Name | Equation | Default Timing at EXTCLK = 96 MHz |
|---|---|---|---|
| fps | Frame Rate | 1/tFRAME | 15 |
| $^{t}$FRAME | Frame Time | (H + max(VB, VBMIN)) × tROW | 66ms |
| $^{t}$ROW | Row Time | 2 × tPIXCLK x max(((W/2) + max(HB, HBMIN)), (41 + 208 x (Row_Bin+1) + 99)) | 33.5µs |
| W | Output Image Width | 2 × ceil((Column_Size + 1) / (2 × (Column_Skip + 1))) | 2592 PIXCLK |
| H | Output Image Height | 2 × ceil((Row_Size + 1) / (2 × (Row_Skip + 1))) | 1944 rows |
| SW | Shutter Width | max(1, (2*16 × Shutter_Width_Upper) + Shutter_Width_Lower) | 1943 rows |
| HB | Horizontal Blanking | Horizontal_Blank + 1 | 1 PIXCLK |
| VB | Vertical Blanking | Vertical_Blank + 1 | 26 rows |
| HBMIN | Minimum Horizontal Blanking | 208 × (Row_Bin + 1) + 64 + (WDC/2) | 312 PIXCLK |
| VBMIN | Minimum Vertical Blanking | max(8, SW - H) + 1 | 9 rows |
| $^{t}$PIXCLK | Pixclk Period | 1/fPIXCLK | 10.42ns |

There are many registers which can be set according to user preference, such as that for Row Start, Column Start, Horizontal and Vertical Blanking, Exposure and so on. Beginners may refer to the settings in the examples provided on Terasic website and revise it to meet their own needs.

```
begin
    case(LUT_INDEX)
    0   :   LUT_DATA    <=  24'h000000;
    1   :   LUT_DATA    <=  24'h20c000;             //  Mirror Row and Columns
    2   :   LUT_DATA    <=  {8'h09,senosr_exposure};//  Exposure
    3   :   LUT_DATA    <=  24'h050000;             //  H_Blanking
    4   :   LUT_DATA    <=  24'h060019;             //  V_Blanking
    5   :   LUT_DATA    <=  24'h0A8000;             //  change latch
    6   :   LUT_DATA    <=  24'h2B0013;             //  Green 1 Gain
    7   :   LUT_DATA    <=  24'h2C009A;             //  Blue Gain
    8   :   LUT_DATA    <=  24'h2D019C;             //  Red Gain
    9   :   LUT_DATA    <=  24'h2E0013;             //  Green 2 Gain
    10  :   LUT_DATA    <=  24'h100051;             //  set up PLL power on
`ifdef VGA_640x480p60
    11  :   LUT_DATA    <=  24'h111f04;             //  PLL_m_Factor<<8+PLL_n_Divider
    12  :   LUT_DATA    <=  24'h120001;             //  PLL_p1_Divider
`else
    11  :   LUT_DATA    <=  24'h111805;             //  PLL_m_Factor<<8+PLL_n_Divider
    12  :   LUT_DATA    <=  24'h120001;             //  PLL_p1_Divider
`endif
    13  :   LUT_DATA    <=  24'h100053;             //  set USE PLL
    14  :   LUT_DATA    <=  24'h980000;             //  disble calibration
    15  :   LUT_DATA    <=  24'hA00000;             //  Test pattern control
    16  :   LUT_DATA    <=  24'hA10000;             //  Test green pattern value
    17  :   LUT_DATA    <=  24'hA20FFF;             //  Test red pattern value
    18  :   LUT_DATA    <=  sensor_start_row    ;   //  set start row
    19  :   LUT_DATA    <=  sensor_start_column ;   //  set start column
    20  :   LUT_DATA    <=  sensor_row_size;        //  set row size
    21  :   LUT_DATA    <=  sensor_column_size;     //  set column size
    22  :   LUT_DATA    <=  sensor_row_mode;        //  set row mode in bin mode
    23  :   LUT_DATA    <=  sensor_column_mode;     //  set column mode  in bin mode
    24  :   LUT_DATA    <=  24'h4901A8;             //  row black target
    default:LUT_DATA    <=  24'h000000;
    endcase
end
```

After initializing the camera module with $I^2C$ protocol, we can adjust exposure and switch to zoom-in mode by changing the settings of sensor_exposure, location of the start row and start column, and the size of the rows and columns with input signals from key buttons or switches on FPGA board.

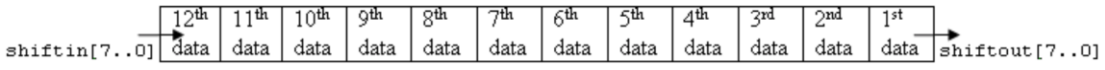
In order to convert from the output raw data of camera module to a RGB format for image processing, we first check out the output data format of the camera module. There is an example of the output data timing diagram below, with P0, P1, P2… represent pixels of a valid image. Here we can see that output data D[11:0] is valid only if signals FVAL and LVAL are both high. When the condition is met, the value of D will be a valid pixel value and will be updated for each PIXCLK clock for transmissions of different pixels.




Since the output data of camera module (i.e. value of D) is not a complete RGB information of a pixel, we could use a line buffer to store previous values. Once the buffer collects complete RGB data information, we are able to output RGB values of one pixel simultaneously. The main function of the line buffer module is shifting, executed by a module generated by Quartus II.

```
altshift_taps   altshift_taps_component (
            .clken (clken),
            .clock (clock),
            .shiftin (shiftin),
            .taps (sub_wire0),
            .shiftout (sub_wire5)
            // synopsys translate_off
            ,
            .aclr ()
            // synopsys translate_on
            );
defparam
    altshift_taps_component.lpm_hint = "RAM_BLOCK_TYPE=M9K",
    altshift_taps_component.lpm_type = "altshift_taps",
    altshift_taps_component.number_of_taps = 3,
    altshift_taps_component.tap_distance = 800,
    altshift_taps_component.width = 12;
```

altshift_taps module is generated by RAM-based shift register (ALTSHIFT_TAPS) megafunction IP core provided by Altera. The ALTSHIFT_TAPS IP core implements a shift register with taps and offers several additional features, including selectable RAM block type, a wide range of widths for shift-in and shift-out ports, selectable distance between taps, and so on. The figure below shows a traditional 12-word-depth shift register.



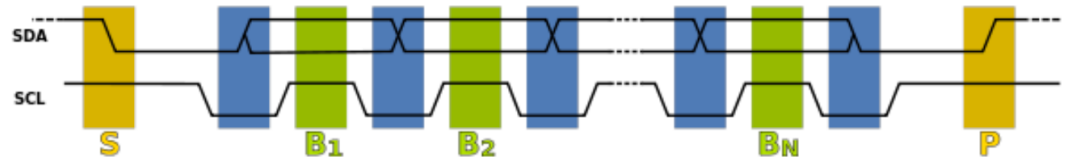

**- SDRAM**

The SDRAM of DE2-115 board has roughly 128MB memory capacity and is achieved by concatenating two 64MB SDRAM chips, which are partitioned into 4 banks. It uses row address and column address to access words, with each being 32-bit long. Since SDRAM is volatile, the memory must be periodically refreshed every once in a while. The read / write operation takes multiple steps to complete,

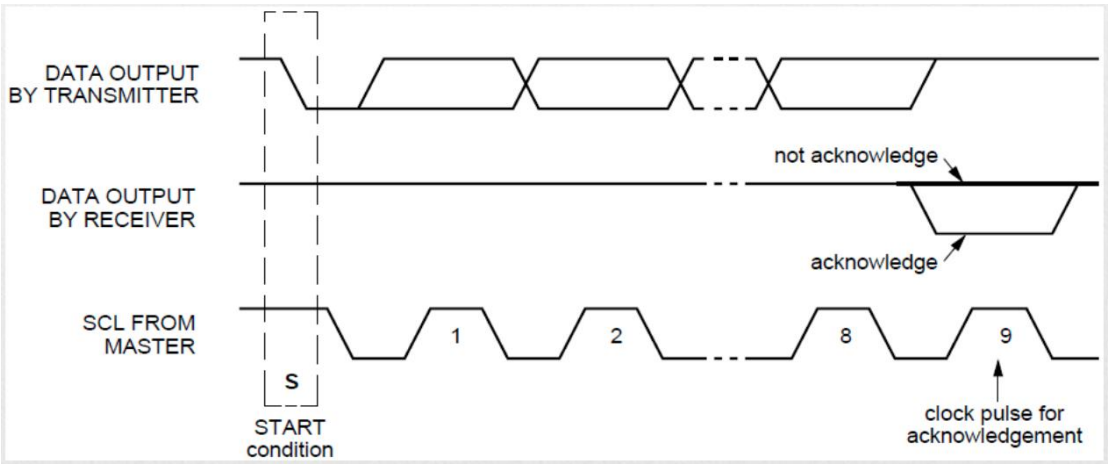
making access to SDRAM generally slower than SRAM.

**- $I^2C$ Protocol**

The picture below is a typical waveform under $I^2C$ protocol, and we shall use it as an example to explain how $I^2C$ protocol works.

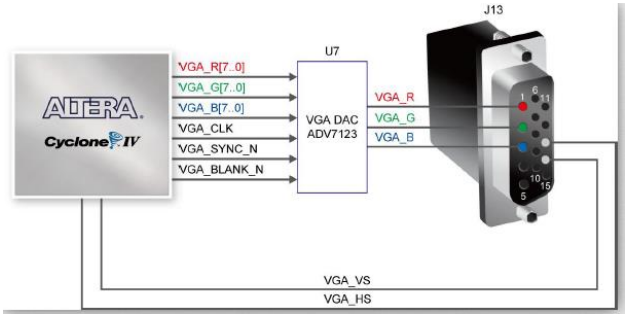



1. The two signals of $I^2C$ protocol, namely SDA and SCL, are preset to high.
2. To start transmission, SDA is pulled low while SCL is high. (refer to the yellow region S)
3. To stop transmission, SDA is pulled high while SCL is high. (refer to the yellow region P)
4. During transmission, SDA sets the data bit to be transferred when SCL is low. (refer to the blue region) While SCL is high, SDA remains unchanged and the data bit is transmitted. (refer to the green region)
5. After transmitting every 8 bits is one direction, an "acknowledgement" bit ACK is transmitted in the opposite direction. If ACK is high, it indicates that the transmission failed; on the other hand, if ACK is low, the transmission is successful. (refer to the figure below)



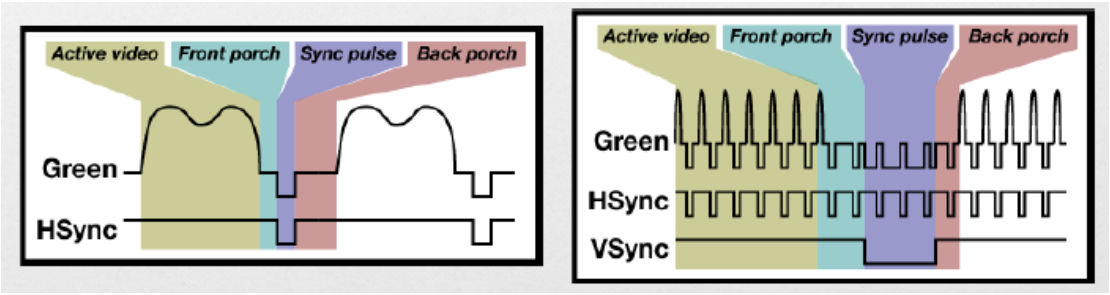

**- VGA**

The DE2-115 board includes a 15-pin D-SUB connector for VGA (Video Graphics Array) output and the synchronization signals are provided directly from the FPGA. The Analog Devices ADV7123 triple 10-bit high-speed video DAC (only the higher 8-bit are used) is used to produce the analog data signals (red, green, and blue).




During active video interval the RGB data drives each pixel in turn across the row being displayed. The data output to the monitor must be off (driven to 0 V) for a time period called the front porch before HSync pulse can occur. Then an active-low sync pulse of specific duration is applied to the horizontal synchronization input of the monitor, which signifies the end of one row of data and the start of the next. Finally, there is a time period called the back porch after the HSync pulse occurs, which is followed by the next active video interval.

The figure below illustrates the basic timing requirements for each row (horizontal) and frame (vertical) that is displayed on a VGA monitor.




Below are the specifications of VGA with 640 × 480 resolution used in this project just for reference:

General timing

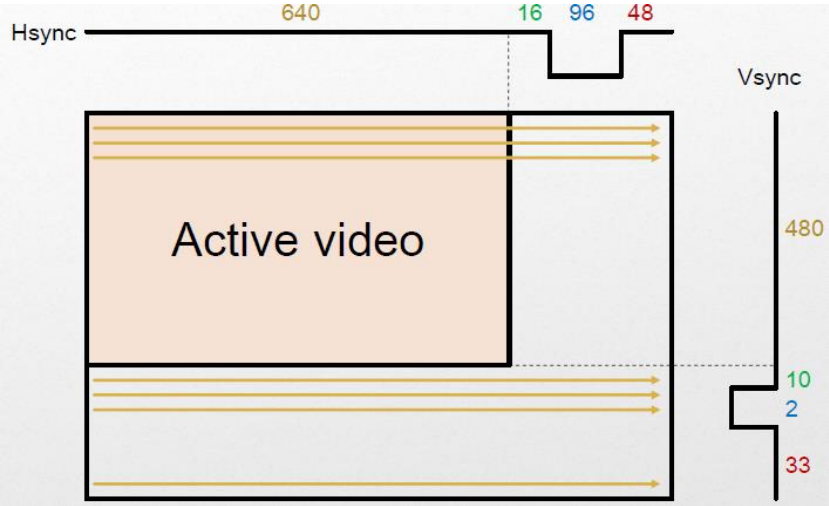
| | |
|---|---|
| Screen refresh rate | 60 Hz |
| Vertical refresh | 31.469 kHz |
| Pixel freq. | 25.175 MHz |

Horizontal timing (line)

| Scanline part | Pixels | Time [µs] |
|---|---|---|
| Active video | 640 | 25.422 |
| Front porch | 16 | 0.636 |
| Sync pulse | 96 | 3.813 |
| Back porch | 48 | 1.907 |
| Whole line | 800 | 31.778 |

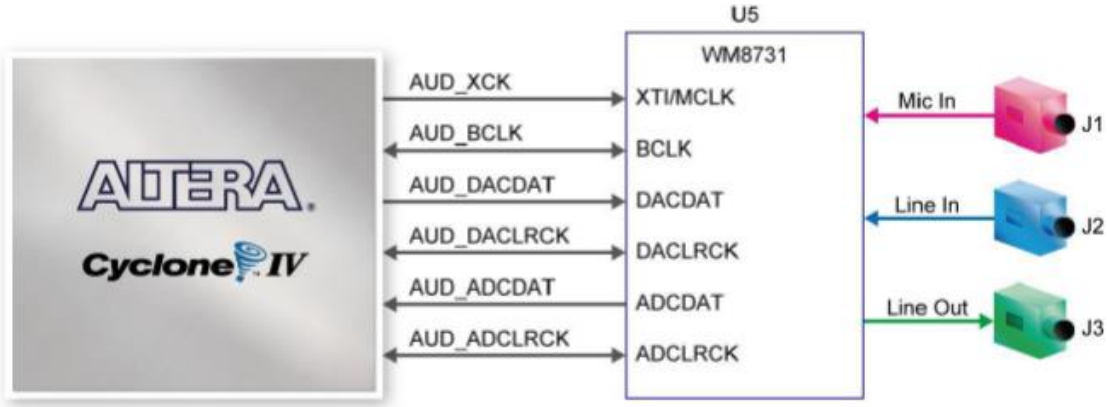
Vertical timing (frame)

| Frame part | Pixels | Time [µs] |
|---|---|---|
| Active video | 480 | 15.253 |
| Front porch | 10 | 0.318 |
| Sync pulse | 2 | 0.0636 |
| Back porch | 33 | 1.049 |
| Whole frame | 525 | 16.683 |




**- WM8731 Audio CODEC**

➢ Schematic Diagram:




➢ Usage:

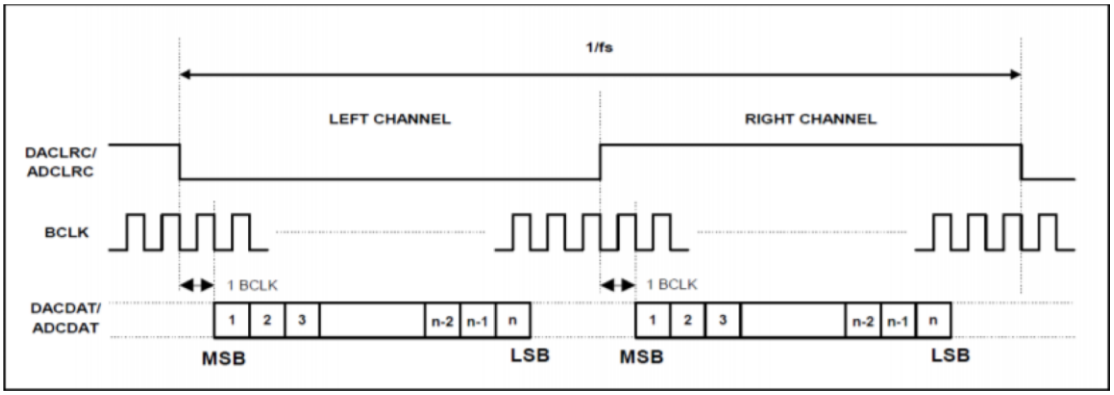
First, we need to set up the WM8731 audio CODEC by changing its register map, and to do so, we must pass configuration to it with the $I^2C$ protocol. The initialization configuration for this lab are as follows.

| | |
|---|---|
| Left Line In | 000_0000_0_1001_0111 |
| Right Line In | 000_0001_0_1001_0111 |
| Left Headphone Out | 000_0010_0_0111_1001 |
| Right Headphone Out | 000_0011_0_0111_1001 |
| Analogue Audio Path Control | 000_0100_0_0001_0101 |
| Digital Audio Path Control | 000_0101_0_0000_0000 |
| Power Down Control | 000_0110_0_0000_0000 |
| Digital Audio Interface Format | 000_0111_0_0100_0010 |
| Sampling Control | 000_1000_0_0001_1001 |
| Active Control | 000_1001_0_0000_0001 |

After initializing, WM8731 will be activated as DAC and ADC between audio signal and the FPGA. The sample rate, or the frequency of DACLRCK/ ADCLRCK, is set to 32kHz. Each DACLRCK/ADCLRCK clock cycle contains one sample of the audio signal, which is 16 bits long in our case. The serial data transmission format follows the rules under $I^2S$ mode.

**- $I^2S$ Protocol**

$I^2S$ mode is one of the audio interfaces offered by WM8731. In a single DACLRCK/ADCLRCK clock, DACLRC/ADCLRC low means the left channel while high means the right channel. When DACLRCK/ADCLRCK changes value, DACDAT/ADCDAT will start to record data after one BCLK clock. The number of bits recorded is 16 in our case, and transmission starts from the MSB to the LSB.



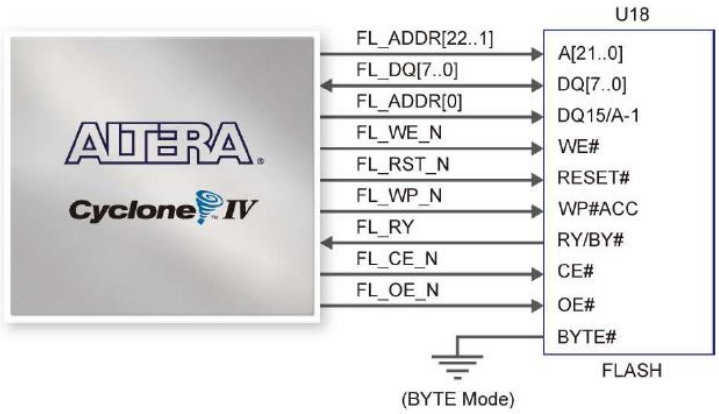

**- Flash Memory**

Flash is a non-volatile memory, that is, the data stored in Flash will not disappear even if the power is turned off. The Flash memory on DE2-115 board is organized as 8M addresses × 8-bit word in byte mode, with 8MB memory capacity in total. The pins for a Flash memory and their functions are shown in the figure below:

| Pin | Description |
|---|---|
| A21–A0 | 22 Address inputs (S29GL064N) |
| A20–A0 | 21 Address inputs (S29GL032N) |
| DQ7–DQ0 | 8 Data inputs/outputs |
| DQ14–DQ0 | 15 Data inputs/outputs |
| DQ15/A-1 | DQ15 (Data input/output, word mode), A-1 (LSB Address input, byte mode) |
| CE# | Chip Enable input |
| OE# | Output Enable input |
| WE# | Write Enable input |
| WP#/ACC | Hardware Write Protect input/Programming Acceleration input |
| ACC | Acceleration input |
| WP# | Hardware Write Protect input |
| RESET# | Hardware Reset Pin input |
| RY/BY# | Ready/Busy output |
| BYTE# | Selects 8-bit or 16-bit mode |
| $V_{CC}$ | 3.0 volt-only single power supply (see *Product Selector Guide* for speed options and voltage supply tolerances) |
| $V_{IO}$ | Output Buffer Power |
| $V_{SS}$ | Device Ground |
| NC | Pin Not Connected Internally |

There are two configurations for the Flash memory, namely, the word mode and the byte mode. The BYTE# pin controls whether the device data I/O pins operate in the byte or word configuration. If the BYTE# pin is set at logic 1, the device is in word configuration, DQ0–DQ15 are active and controlled by CE# and OE#. If the BYTE# pin is set at logic 0, the device is in byte configuration, and only data I/O pins DQ0–DQ7 are active and controlled by CE# and OE#. In the mean time, data I/O pins DQ8–DQ14 are tri-stated, and the DQ15 pin is used as an input for the LSB (A-1) address function.

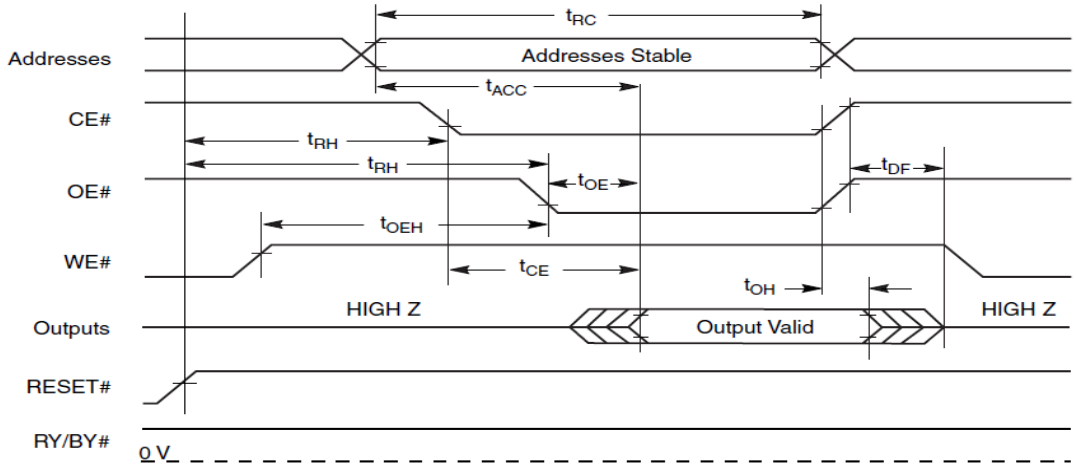
Connections between FPGA and Flash in byte mode are shown below:

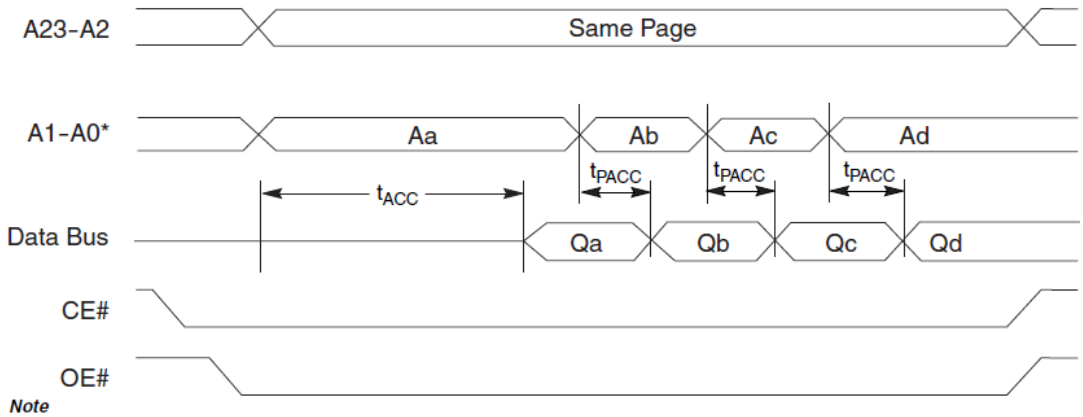



To read array data from the outputs, the system must drive the CE# and OE# pins to $V_{IL}$ while WE# should remain at $V_{IH}$. CE# is the power control and selects the device. OE# is the output control and gates array data to the output pins.

The device is capable of fast page mode read and which provides faster read access speed for random locations within a page. The page size of the device is 8 words/16 bytes. The appropriate page is selected by the higher address bits A[22]–A[2] while address bits A[1]–A[0] in byte mode determine the specific word within a page.

Timing diagrams of normal read operation and page mode rad are as follows:





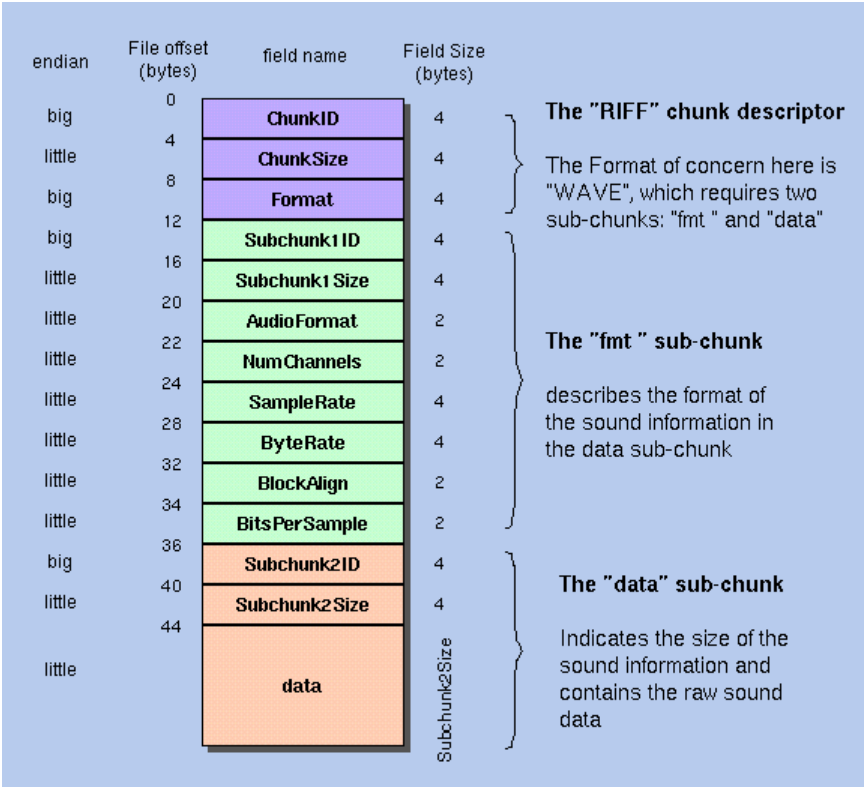

***Note***

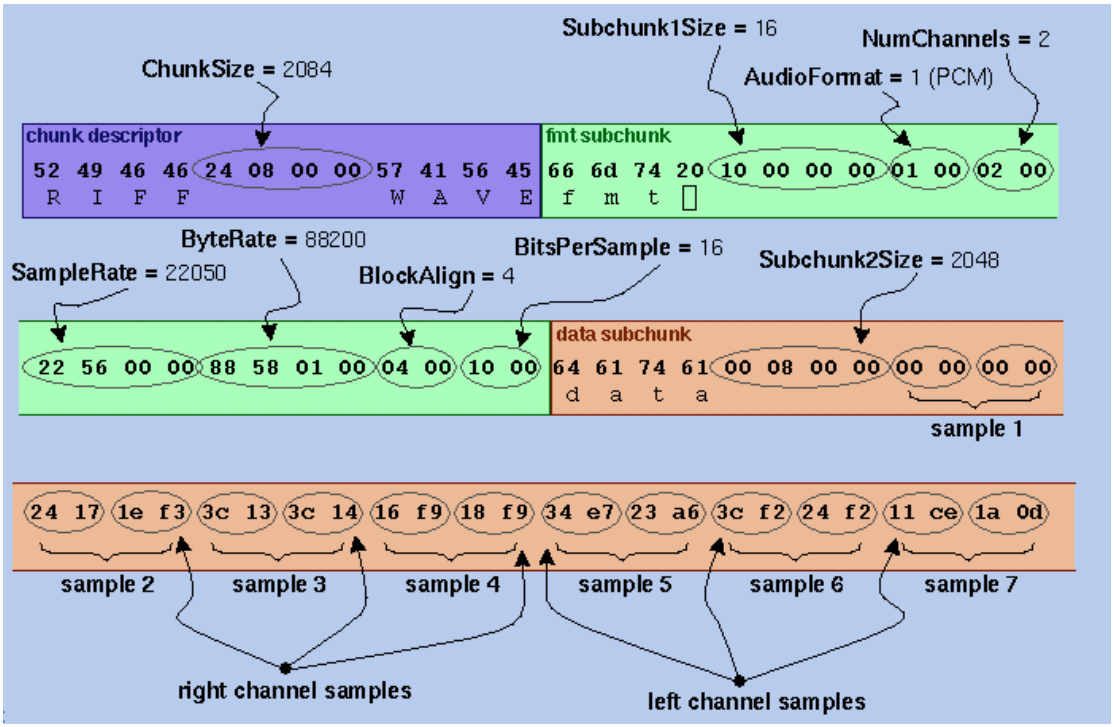
** Figure shows device in word mode. Addresses are A1–A-1 for byte mode.*

**- WAV Format**

Waveform Audio File Format (WAVE, or more commonly known as WAV due to its filename extension) is a Microsoft and IBM uncompressed audio file format standard for storing an audio bitstream on PCs.

The header of a canonical WAV file format and an example are shown in the figures below:






**- SRAM**

The SRAM on DE2_115 has roughly 2MB memory capacity and it transmits data in a parallel way.

There are seven signals associated with the SRAM:

- SRAM_ADDR[19:0]:
  The read/write address of the SRAM.
- SRAM_DQ[15:0]:
  The 16-bit data to be read or stored in the SRAM.

※ Note that this is a bidirectional inout port, meaning that one must beware of the control of this signal to avoid multiple driver of signal.

- SRAM_OE SRAM:
  Output Enable
- SRAM_WE SRAM:
  Write Enable
- SRAM_CE SRAM:
  Chip Enable
- SRAM_LB SRAM:
  Lower Byte Control
- SRAM_UB SRAM:
  Upper Byte Control

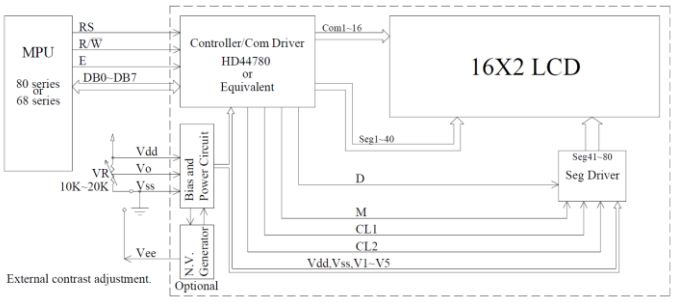
The control signals of the read/write operation are listed below:

| Mode | $\overline{WE}$ | $\overline{CE}$ | $\overline{OE}$ | $\overline{LB}$ | $\overline{UB}$ | I/O PIN I/O0-I/O7 | I/O PIN I/O8-I/O15 | $V_{DD}$ Current |
|---|---|---|---|---|---|---|---|---|
| Not Selected | X | H | X | X | X | High-Z | High-Z | $I_{SB1}$, $I_{SB2}$ |
| Output Disabled | H | L | H | X | X | High-Z | High-Z | $I_{CC}$ |
| | X | L | X | H | H | High-Z | High-Z | |
| Read | H | L | L | L | H | $D_{OUT}$ | High-Z | $I_{CC}$ |
| | H | L | L | H | L | High-Z | $D_{OUT}$ | |
| | H | L | L | L | L | $D_{OUT}$ | $D_{OUT}$ | |
| Write | L | L | X | L | H | $D_{IN}$ | High-Z | $I_{CC}$ |
| | L | L | X | H | L | High-Z | $D_{IN}$ | |
| | L | L | X | L | L | $D_{IN}$ | $D_{IN}$ | |

**- LCD**

The LCD module of DE2-115 has built-in fonts and can be used to display text by sending appropriate commands to the display controller called HD44780.

The block diagram is shown in the figure below:




Pin assignments of LCD module are as follows:

| Signal Name | FPGA Pin No. | Description |
|---|---|---|
| LCD_DATA[0]~[7] | PIN_L3~M5 | LCD Data[0]~[7] |
| LCD_EN | PIN_L4 | LCD Enable (level sensitive: 1, edge sensitive: 1→0) |
| LCD_RW | PIN_M1 | LCD Read/Write Select (0:write 1:read) |
| LCD_RS | PIN_M2 | LCD Command Select (0:command 1:data) |
| LCD_ON | PIN_L5 | Power ON/OFF |
| LCD_BLON | PIN_L6 | LCD Back Light ON/OFF |

The LCD display Module is built in a LSI controller. The controller has two 8-bit registers, an instruction register (IR) and a data register(DR). The IR stores instruction codes, such as display clear and cursor shift, and address information for display data RAM (DDRAM) and character generator (CGRAM). The DR temporarily stores data to be written or read from DDRAM or CGRAM. The address counter (AC) assigns addresses to both DDRAM and CGRAM. This DDRAM is used to store the display data represented in 8-bit character codes. Its extended capacity is 80×8bits or 80 characters. Below figure is the relationships between DDRAM addresses and positions on the liquid crystal display.

| RS | R/W | Operation |
|---|---|---|
| 0 | 0 | IR write as an internal operation (display clear, etc.) |
| 0 | 1 | Read busy flag (DB7) and address counter (DB0 to DB6) |
| 1 | 0 | Write data to DDRAM or CGRAM (DR to DDRAM or CGRAM) |
| 1 | 1 | Read data from DDRAM or CGRAM (DDRAM or CGRAM to DR) |

- **Useful Tools**

**- ALTPLL**

ALTPLL is a clock rate conversion tool provided by Quartus II. In this lab, we use PLL to convert the original clock rate from 50 MHz to 12 MHz and 100 kHz. The 12MHz clock is then fed to the AUD_XCK port and the 100kHz clock is used to initialize WM8731 through $I^2C$ protocol.

| Signal | Type | Description |
|---|---|---|
| inclk0 | input | native 50 MHz clock signal |
| c0 | output | tuned 12 MHz clock signal |
| c1 | output | tuned 100 kHz clock signal |
| c2 | output | tuned 25 MHz clock signal |

**- SignalTap Logic Analyzer**

After programming all your modules onto the FPGA board, Quartus II provides a very useful tool to monitor your signal waveforms – SignalTap. It provides instant waveforms and thus is very helpful when debugging run-time errors.

1. Go to File → New → choose SignalTap II Logic Analyzer File → click OK.
2. At the "Setup" section you can select signals you want to monitor, and set the basic clock at "Signal Configuration".
3. Set the trigger condition, which is basically when to start monitoring the signals you have selected.
4. Save your settings as xxx.stp.
5. Go back to your project and go to Assignments → Settings → SignalTap II Logic Analyzer and specify the stp file.
6. Compile your design.

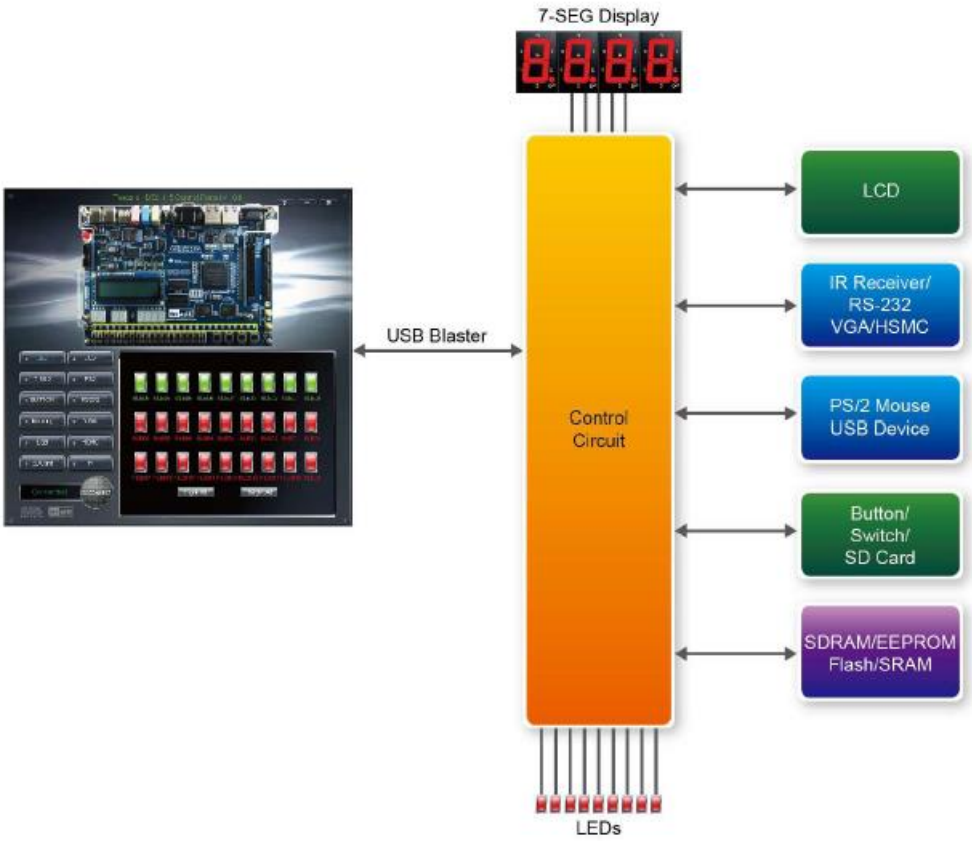
※ SignalTap basically insert probes in your circuits to monitor the signals specified by the user, so we have to recompile the design to activate them.

7. Program the sof file to your FPGA.
8. Open the stp file once again and switch to the “Data” section.
9. Go to Processing → Run Analysis.
10. Satisfy the trigger condition and start to monitor your signals.

**- DE2_115 Control Panel**

The DE2-115 Control Panel allows users to access various components on the board from a host computer. The host computer communicates with the board through a USB connection. The facility can be used to verify the functionality of components on the board or be used as a debug tool while developing RTL code.

The concept of the DE2-115 Control Panel is illustrated in the figure below. The “Control Circuit” that performs the control functions is implemented in the FPGA board. It communicates with the Control Panel window, which is active on the host computer, via the USB Blaster link. The graphical interface is used to issue commands to the control circuit. It handles all requests and performs data transfers between the computer and the DE2-115 board.



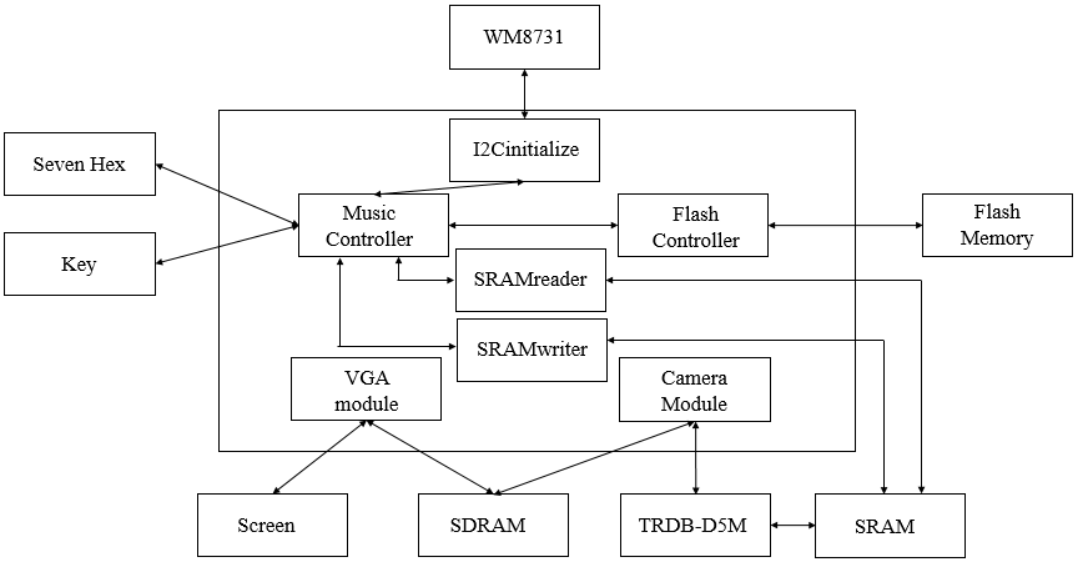

The most important feature lies in that reading/writing a word or an entire file from/to the Flash Memory allows the user to develop multimedia applications (Flash Audio Player, Flash Picture Viewer) without worrying about how to build a Memory Programmer.

- **Overview**

  Here we briefly give an overview of how our system works, and dig into more details of each component later. Beginning with the D5M camera which captures images at about 30 frames per second, the captured CCD raw data are send to the camera controller, where they are converted into a black-and-white grey scale image. The image is stored in SDRAM while the recognition result of which note to play in this musical bar (or measure) is stored in SRAM. The former is for the VGA controller to display images on the screen in real-time, and the latter is for controlling WM8731 audio CODEC to access the correct audio data from Flash memory and play the right sound of the notes. Upon recognition, states of the system will be set according to the notes in this bar, and the system will play the corresponding piano sounds. Having confirmed that the sound combinations in the bar are just as expected, you can save the current bar information and proceed to the next bar. After composing your own melody, just simply play all the bars at once and admire your own piece of music work!

- **Design**

  **- Overall Structure:**



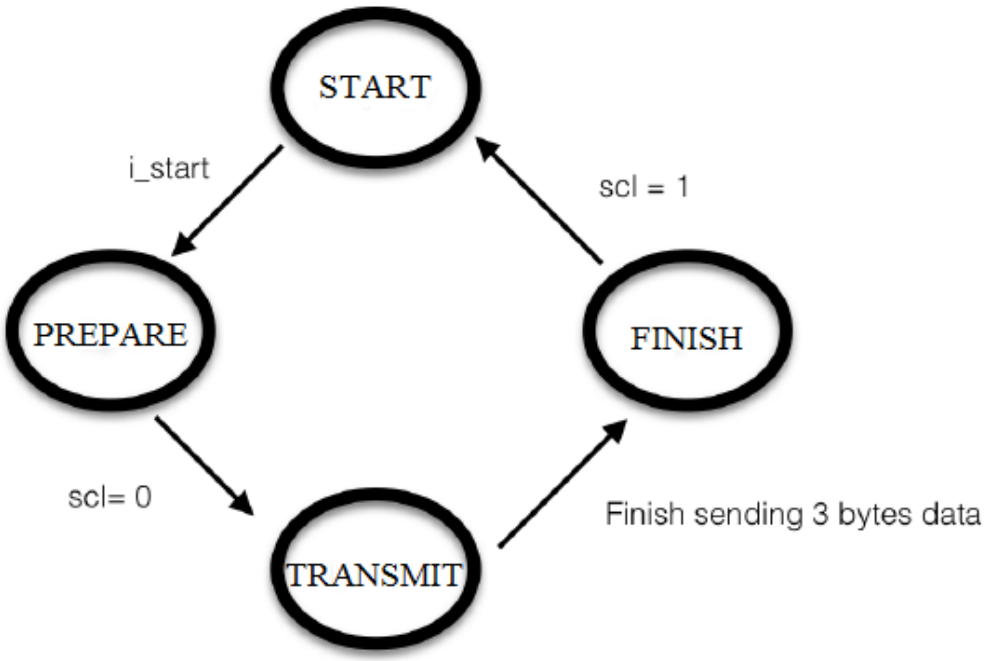

  **- I2Csender:**

  **►Principle**

  In practice, each data transfer would transmit 24 bits of data, which contains three parts. We take the recommended value of Active Control for example.

  1. Hardware location of WM8731( leftmost 7 bits ): 0011010
  2. Location of registers in WM8731( 8th ~ 14th bits from the left ): 00001001
  3. Operation data for register: 0_0000_0001, for example.

► **Finite State Machine**




- START:
  When i_start is high, the module would read the input data (i_data_w = input), and initialize all the counters as well as output data to 0. After initialization, the state would be changed to PREPARE.
- PREPARE:
  When SCL is high, we would set SCL to low state, set output bit to the left most bit of the input data read in START, and shift one bit left of the input data (i_data_w = i_data_r << 1). After doing so, the next state would jump to TRANSMIT.
- TRANSMIT:
  According to the protocol, each time before sending 1 byte data, SCL should be changed from high to low. Therefore, we set SDL = 1 at the first clock cycle and SDL = 0 at the second clock cycle. After doing this, we will start to send the data bit by bit just like what we did in PREPARE (assign output data and shift left 1 bit of input data).
  Upon finishing sending one byte data, we would set the output bit to the ACK signal (1'bz). In practice, we have another signal 'ack' which is set to 1 after sending 8 bit data or otherwise remaining 0 to control the above behavior. For the purpose of depending whether to receive acknowledgement bit or not, we wrote assign o_sda = ack_r ? 1'bz : sda_r.
  After sending 3 bytes data, the next state would be FINISH.
- FINISH:

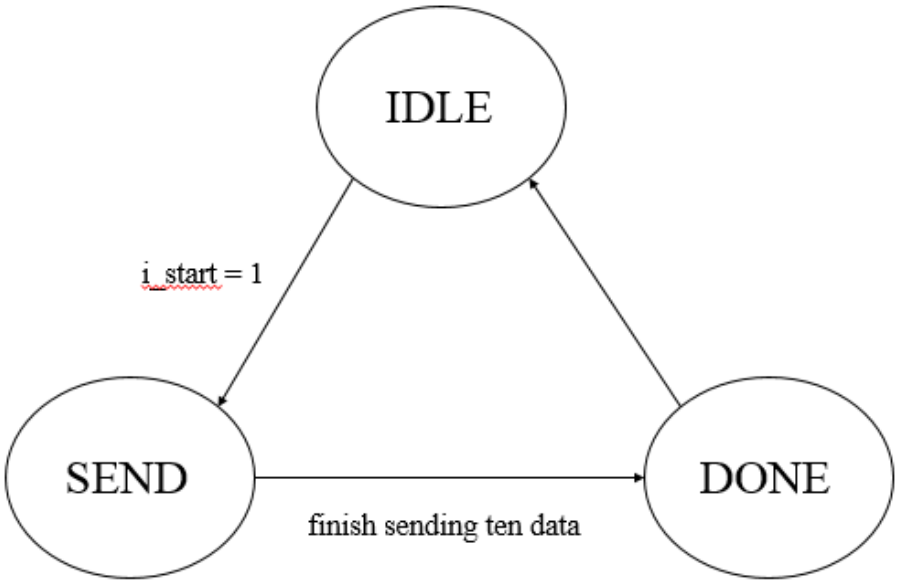
If SCL is high, we set both output bit and finish signal to high and the state would jump to START.

**- I2Cinitialize:**

**►Principle**

In our design, we have to initialize 10 different registers for WM8731, so we need to use I2Csender.sv 10 times to send 24 bits data each time.

**►Finite State Machine**




**- flash:**

**►Principle**

Since the audio data of piano sounds for different notes has 2 channels with 16 bits per sample, we are going to receive 32 bits for one sample from the Flash memory by page mode read and then output the data. We set the Flash memory to byte mode and it will be organized as 8-bit word, so we need to read four addresses (execute four cycle) to get a 32-bit full sample. To do page mode read, we get the higher address bits (first 21 bits) from the input to select appropriate page and then use a counter to determine the lower address bits (last 2 bits). After concatenating them as a whole address, simply send it to the Flash memory and acquire the data.

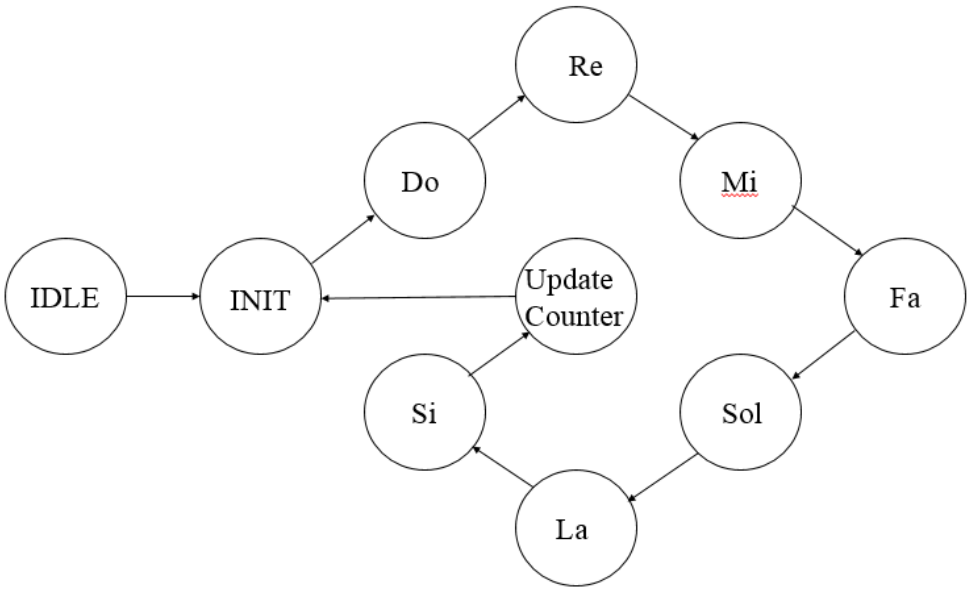
**- Flash_Controller:**

**►Principle**

In our system, there is a state indicating which sound tracks are enable, meaning that the system recognizes the corresponding notes in the image captured by the camera. For example, the state of ‘Do’ would be set to 1 once the system has detected a magnet in the position for the note ‘Do’ by the captured image. Each

sound track is stored in the flash memory at different addresses but with same spatial size. In addition, there is a counter in our system for time synchronization of all the sound tracks. For example, there are 7 sound tracks, referring to 'Do', 'Re', 'Mi', 'Fa', 'Sol', 'La' and 'Si' respectively, stored in flash with address beginning from 00000, 10000, 20000, 30000, 40000, 50000 and 60000 and each being 3EA7 bits long. When the counter = 1, the system reads data from address 00000, 10000, 20000, 30000, 40000, 50000 and 60000 sequentially, and add it to a temporal register if the corresponding state of the track is set (add the value of 00001 to temporal register if the state of sound 'Do' is set, and discard the value of 10001 if the state of 'Re' is not set despite the system has already read its value from the flash memory.) For each audio sampling cycle, read all the values in the address: (beginning of certain sound track) + (counter value), and add it to temporal register according to the states. Finally, send the value of temporal register to Para2Seri module and further being passed to WM8731 audio CODEC chips to play a single note or chord.

**►Finite State Machine**




**- SRAMread and SRAMwrite:**

**►Principle**

The function of SRAMREADER is music storage and clearing data while the function of SRAMREAD is reading audio data of the music. We have to transmit 4 times because there are 8*8=64 bits in a single syllable.

**- Music Controller:**

**►Principle**

There are two modes, namely, Record & Play in this module. The substates PLAY1 to PLAY8 play the role of representing the eight possible notes in a single

bar, and the state returns to IDLE whenever the substate changes. When in the RECORD mode, the composite music signals mentioned above transferred from the object detector are sequentially transmitted to the Flash memory via para2seri module. When in the PLAY mode, however, the mechanism is much more complicated than the previous one. First of all, we have to send signal and address to SRAMreader. Next, it sequentially selects the stored value and transmitts a "done" signal back to MUSIC CONTROLLER. Finally, the control signals are delivered to Flash.

- **Ingenuity**

  **- Dynamic Capturing of Camera (Real Time Processing)**

  At first, we thought we could do the image processing after we capture the image at a particular timing by taking a photo. However, the main problem lies in that we are not able to identify recognition error immediately but have to wait until the photo is taken and properly processed. As a consequence, we decide to process every frame captured by the D5M camera and play the sound out in real time. If anything goes wrong or the melody in the bar does not sound harmonic, the user can immediately recognize and make some adjustments, leading to a more user-friendly interface.

  **- Implementing Chord by Mixing Different Notes**

  In the beginning, we aim to play only single note at any particular timing. Nonetheless, the possible combinations of notes would be rather limited and the player is not capable of reproducing some huge hit popular music. Therefore, the ability to play chord becomes a major concern if we want to increase it usability. At first thought, we are about to simply store the .wav data of every possible chords in the Flash memory. But, with the fact that there are in total $C_1^7 + C_2^7 + C_3^7 + C_4^7 + C_5^7 + C_6^7 + C_7^7 = 127$ different kinds of them in mind, which is far larger than the Flash memory's capacity, we have no choice but to find an alternative method. Eventually, we decide to implement chords by playing distinct single notes at the same time, or more precisely, mix the audio data of different notes together.

- **Problems Encountered**

  **- Data Contamination**

  Since by default SRAM starts to store data from the address 0, the image data we store are relatively likely to be contaminated.

  ✓ **Solution:**

  Cancel the bottom address 0 and start from the preceding address for storage.

  **- 'Crack' Sound when Playing Chord**

  As mentioned above, our chord is achieved by mixing different audio data of

single notes together with a temporal register. With the number of notes in a chord increasing, the temporal register mentioned above may be added over its bits limitation, that is, overflow occurs, resulting in an unbearable ‘crack’ sound.

✓ **Solution:**

For this reason, we designed an overflow handler to deal with the problem. Although the audio data has only 16 bits per clock cycle, the temporal register is set to include 19 bits. Note that the value of audio data are ‘signed’, thus when extending 16 bits data into 19 bits, sign extension must be performed, that is, the MSB should be extended rather than simply adding zeros in the front. After adding all note sounds together, our system uses these additional three bits along with the original MSB to check if overflow has occurred. The followings are the conditions indicating whether overflow has occurred:

```
if (play_data_temp_r[18:15] == 4'b0000) begin // positive no overflow
    play_data_w = play_data_temp_r[15:0];
end else if (play_data_temp_r[18:15] == 4'b1111) begin // negetive no overflow
    play_data_w = play_data_temp_r[15:0];
end else if ((play_data_temp_r[18] == 1'b0) && (play_data_temp_r[17:15] != 4'b0)) begin // positive overflow
    play_data_w = {1'b0, 15'b111_1111_1111_1111}; // clip to maximum positive value
end else begin // negative overflow
    play_data_w = {1'b1, 15'b0}; // clip to minimum negative value
end
```

**- Generated Latch**

counter_r of LCD module may generate possible latch.

✓ **Solution:**

we spent a long time debugging, and it finally came out that we didn't consider the initialization of the address of DDRAM.

**- Analogy to Premature Discharge**

Since LCD needs longer time cycle to initialize and our system fluctuates too fast that LCD is not capable of catching up the speed according to its change.