The Final Project Report for CSEE 4840 Embedded System Design

Invisibility-Curtain

Srivatsan Raveendran (sr3859)

Abhijeet Nayak (an3075)

Guide: Prof. Stephen A. Edwards

S

pring 2022

Table of Contents

1 Introduction

The invisibility cloak is a typical chromakey example of detecting a specific color and masking it in a video stream. This project aims to use the DE-1 SoC to perform edge video and image processing to execute the graphical effect of an invisibility cloak/curtain. The Invisibility cloak functions as if a piece of cloth with a specific range of RGB values is held before the camera; the regions of the cloak in the video frame disappear to reveal the original background behind it. This effect gives an invisibility effect to the person covered in the veil. The project aims to involve the design of hardware systems and interfaces to obtain an end-to-end system capable of performing real-time camera image processing to achieve the stated task.

Our design aims to perform camera interface, video frame acquisition, and its respective format conversion (A2D and YCbCr to RGB) in the FPGA implemented through Altera IP cores. The chroma key effect of replacing foreground color with the background frame is done in hardware in the SRAM. SRAM stores the captured background video frame taken before the program starts and sends back the modified frame to the VGA to display on the monitor.

The introduction briefly discusses some terminologies that are precursors to understanding the video acquisition pipeline. Later in the Systems Diagram section, a detailed discussion of design decisions will be presented.

1.2 YCbCr Color Space

The Luminance-Chrominance (YCrCb) color space contains information about the brightness (luminance or luma) and color (chrominance or chroma). The color is represented as two components, namely chrominance-red (Cr) and chrominance-blue (Cb). The Altera IP allows 8 bits for each Y, Cr, and Cb. There are two possible format modes of interest for our use case:

1. YCrCb 4:4:4 -- This format is the normal YCrCb with all components, as shown in Figure 1. This mode is defined as 8 bits per color and three color planes.

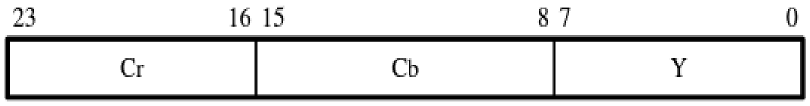


Fig.1 16-bit YCrCb 4:4:4 Color Space

2. YCrCb4:2:2—This format is only half of the Cr and Cb entities, as shown in Figure 2. Each consecutive pixel has alternating Cr or Cb components, with the first pixel in the frame starting with the Y and Cb pixel. This mode is defined as 8 bits per color and two-color planes.

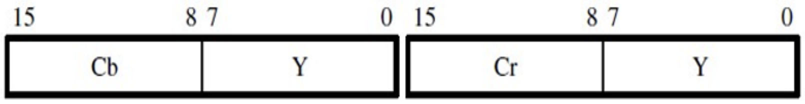


Fig.2 16-bit YCrCb 4:2:2 Color Space

The DE1 SoC uses a 4:4:4 setting for YCbCr.

1.2 RGB Color Space

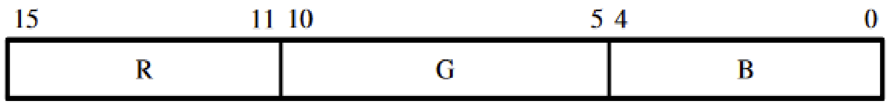


Fig.3 16-bit RGB color space

This format uses 5 bits for red, 6 for green, and 5 for blue, as shown in Figure 3.
If R and B are 5-bit integers and G is a 6-bit integer, then `color = B+(G<<5)+(R<<11);`

1.3 NTSC

The NTSC Standards and Video Capturing: NTSC refers to the National Television Standards Committee. Their standards for interfacing video cameras are as follows.

a) The clock frequency is 27MHz.

b) The cycle frequency is 60 Hz.

c) Video is sent interlaced, implying that two frame cycles are needed to capture a full video display. The first frame is all the odd horizontal lines, and the second frame is all the even horizontal lines.

The graph below shows the I2C timing diagram (SDA- Serial Data; SCL – Serial Clock) for communication between the NTSC peripheral and the DE1 FPGA's ADV7180 video chip.

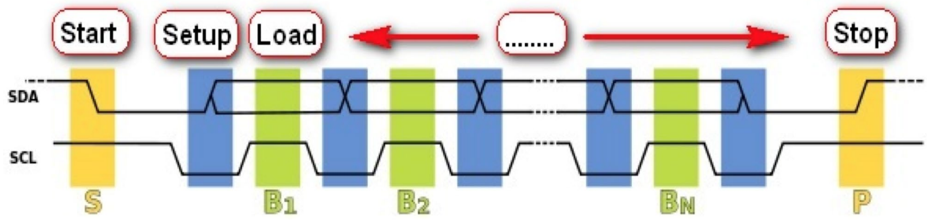


Fig 4. I2C timing diagram

From Page 2 of the datasheet for ADV 7180, we know that the SCL supports a maximum of 400KHz clock. We use this information to build a clock divider RTL that steps down the 27MHz clock into 40KHz.

We create a place in memory to store the 8-bit command, 8-bit address, and 8-bit data that we receive from the I2C camera peripheral. These regions are separate registers.

The Video decoder receives the I2C data and SCL. For the serial shifting of the clock and the data, we generate an I2C SCL and 8-bit values for command address and data. Using the I2C protocol, we wait for an acknowledgment from the decoder and then stop sending this information to the decoder. The substitution technique is used to receive the frames from the decoder into the SDRAM on one end, read the same frame from memory, and display it to the monitor through the VGA Raster module.

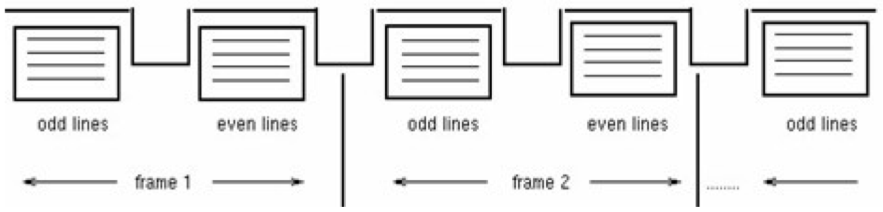


Fig 5. NTSC Interlacing Frame pattern

2 System Block Diagram

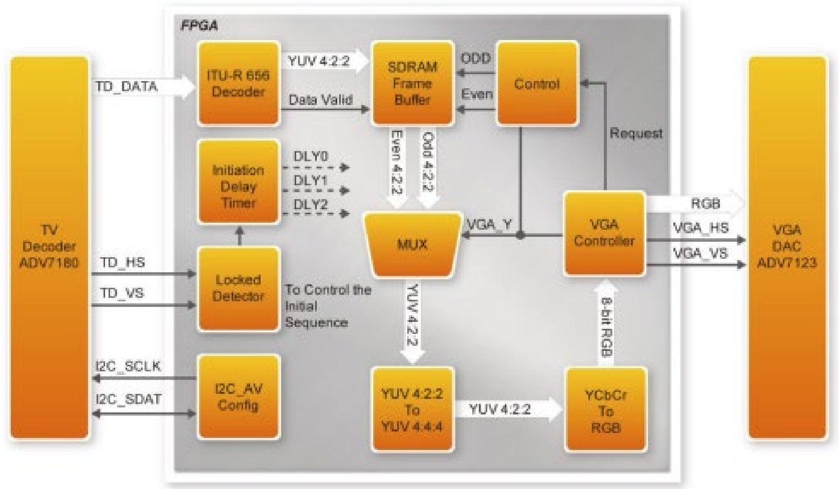


Fig 6. Block diagram for video processing

The hardware components of this system include data acquisition from the NTSC camera via I2C_AV_config, TV_to_VGA video decoder block, logic to display the captured video frame data stored in SRAM via a VGA raster RTL, video frame display, and a four-port SDRAM block to buffer the YCbCr image pixels.

The Video decoder receives the I2C data and SCL. For the serial shifting of the clock and the data, we generate an I2C SCL and 8-bit values for command address and data. Using the I2C protocol, we wait for an acknowledgment from the decoder and then stop sending this information to the decoder. The substitution technique is used to receive the frames from the decoder into the SDRAM on one end, read the same frame from memory, and display it to the monitor through the VGA Raster module.

Inside the TV_to_VGA video decoder block, the ITU-R 656 Decoder block extracts YcrCb 4:2:2 (YUV 4:2:2) video signals from the ITU-R 656 data stream sent from the TV decoder. It also generates a data valid control signal, indicating the valid data output period. De-interlacing needs to be performed on the data source because the video signal for the TV decoder is interlaced. The SDRAM Frame Buffer and a field selection multiplexer (MUX), which the VGA Controller controls, are used to perform the deinterlacing operation. The VGA Controller also generates data requests and odd/even selection signals to the SDRAM Frame Buffer and field selection multiplexer (MUX). The YUV422 to YUV444 block converts the selected YcrCb 4:2:2 (YUV 4:2:2) video data to the YCbCr 4:4:4 (YUV 4:4:4) video data format. Finally, the YcrCb_to_RGB block converts the YCbCr data into RGB data output. The VGA Controller block generates standard VGA synchronous signals VGA_HS and VGA_VS to enable the display on a VGA monitor.

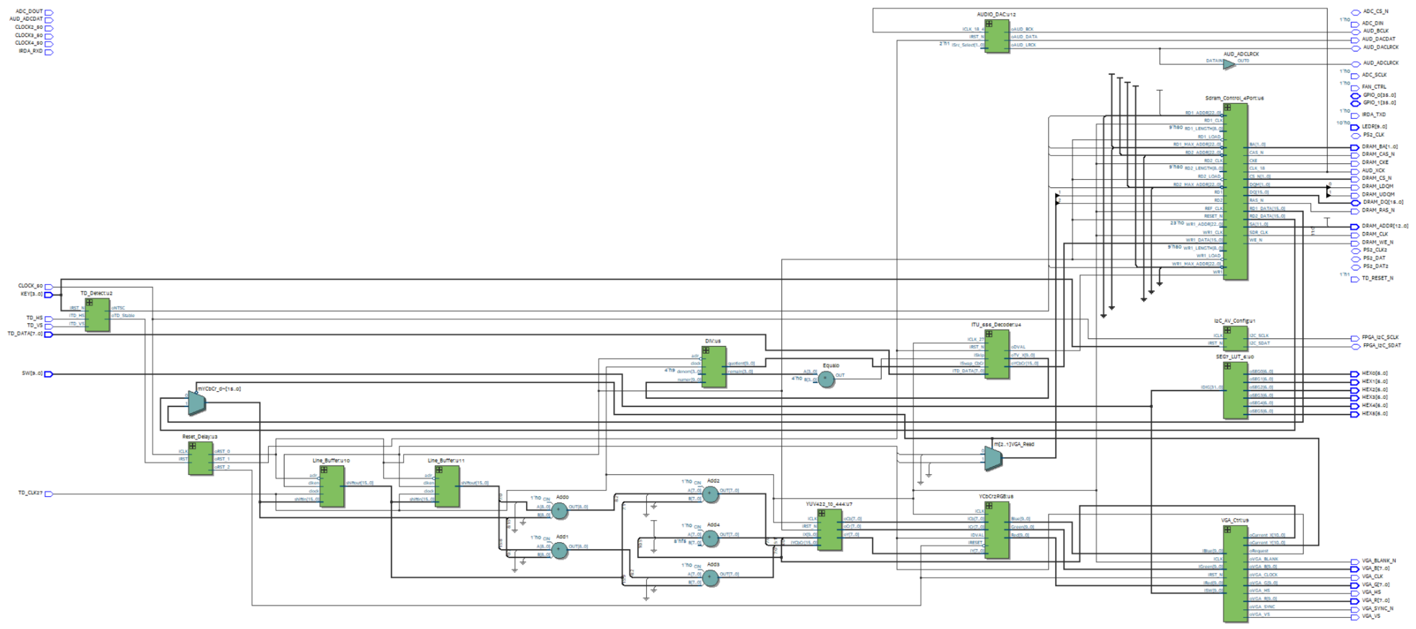


Fig 7.RTL Schematic

Upon flashing the synthesized code, the hardware starts streaming frames. The camera input is updated with newer frames that rewrite the background frame in SDRAM as the clock moves forward. These form the frames we operate upon using the range of RGB values detected on scanning each pixel to replace the video frame pixels with our stored background image stored in SRAM. The data in the memory follows the substitution fashion of access. The new video frame goes to the lower half of memory and moves to the upper half for VGA buffering.

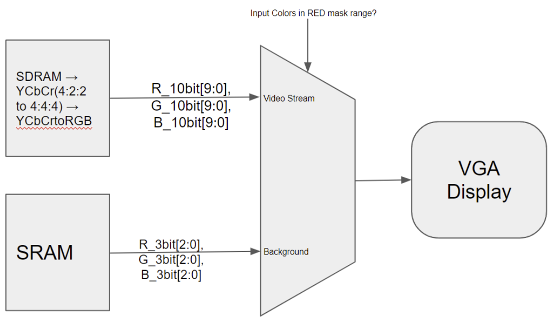

Input Colors in RED mask range?
SDRAM → YCbCr(4:2:2 to 4:4:4) → YCbCrtoRGB
R_10bit[9:0], G_10bit[9:0], B_10bit[9:0]
Video Stream
SRAM
R_3bit[2:0], G_3bit[2:0], B_3bit[2:0]
Background
VGA Display

Fig 8. Chromakey Logic

3 Algorithms

To arrive at the correct steps to implement the system, we first simulate the system through a python program (refer to Appendix).

The following are the stages of the simulated program:

1. General Algorithm Flow

   i) Acquiring RGB background image using OpenCV and storing it in an array

   ii) Flip the image frame

   iii) Conversion of RGB to HSV

   iv) Generating a mask that can be applied on the incoming frame (In software)

   a. Generate two different ranges of HSV values for the color of interest and apply the ranges to threshold the image. This leaves us with two masks.

   b. Perform the addition of these two masks to incorporate the lower and upper bound of the range of HSV values

   c. Apply Morphological opening operation on the mask with a 3,3 filter for two iterations

   d. Apply Morphological image dilation operation on the mask using a 3,3 filter for a single iteration.

   e. Bitwise, invert the mask and call it mask_2

   f. Apply bitwise and on the masked background and the original background using mask_1

   g. Perform bitwise AND on the incoming image with the masked version of the image frame using mask_2

   h. Combine the results from steps 'f' and 'g' using weighted addition

   i. Display the result of the addition as the operated final image.

   In hardware since conversion to HSV requires more synchronization considerations, this was replaced by a RGB masking based on predetermined color threshold.

3.1 Morphological Image Opening

Opening refers to the morphological dilation of the erosion of a set with a structuring element. The effect of the operator is to preserve foreground regions that have a similar shape to this structuring element, or that can completely contain the structuring element while eliminating all other areas of foreground pixels. It is used to preserve intensity patterns in the image. (Fig 6.)

The opening equation is: $(A \circ B = (A \ominus B) \oplus B)$

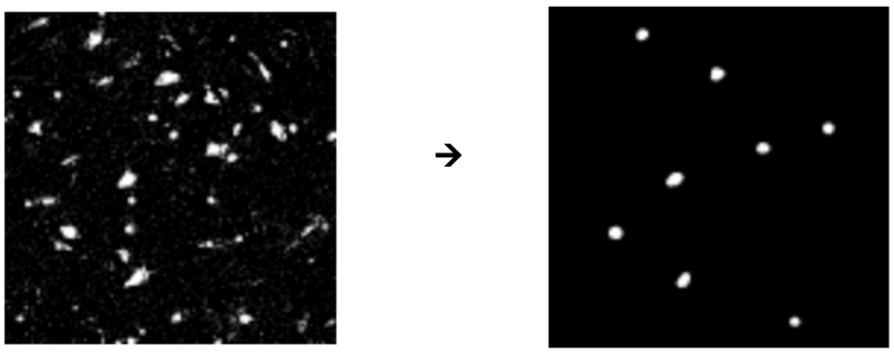


Fig 9. Illustrates the functioning of image opening on a thresholded image.

3.2 Morphological Image Dilation,

The basic effect of the operator on a binary image is to gradually enlarge the boundaries of regions of foreground pixels (i.e., white pixels, typically). Thus areas of foreground pixels grow in size while holes within those regions become smaller. For each background pixel (which we will call the input pixel), we superimpose the structuring element on top of the input image so that the origin of the structuring element coincides with the input pixel position. If at least one pixel in the structuring element coincides with a foreground pixel in the image underneath, then the input pixel is set to the foreground value. If all the corresponding pixels in the image are background, the input pixel is left at the background value.

Fig 8. Below illustrates the working of image dilation on a sample binary image.

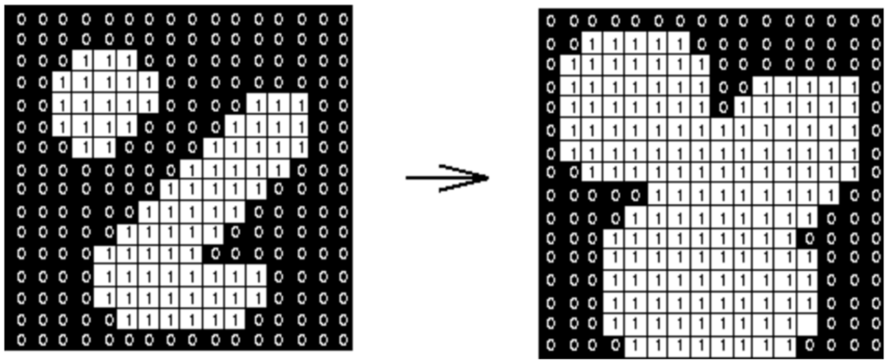


Fig 10. Dilation on sample image

2. Display Logic - VGA Raster algorithm

The following signals and logic must be generated to perform a VGA display scan.

1. vsync, hsync, and blank signals.

2. First, using the 27 Mhz clock obtained from the ADV 7180, we create a counter and generate the needed sync pulses.

We follow the steps below to configure the RTL for VGA counters and pixel generation.

1. The memory configuration is 16 address lines and 32 data bits.
2. We were using a substitution method for video capture and display.
3. Lower 16 bits [0 - 15] were for odd video lines.
4. Upper 16 bits [16 - 31] were for even video lines.
5. Two frames were required to create a full video frame.
6. Vid_udl and Vid_udh store all the even lines of video data.
7. Vid_ldl and Vid_ldh store all the odd lines of video data.
8. There are a total of 210 vertical lines. Also, note we started after 30 vertical lines and ended at 240 vertical lines (240-30= 210). So the full frames is 210 x 2 = 420
9. There were 624 horizontal video pixels (8 bits per pixel). Again note we started the counter at 150 pixels and stopped at 774 pixels (774-150 = 624).

In step 4 of the above algorithm, steps a through h are performed in software, and the resulting image is sent back to the hardware for the display to VGA.

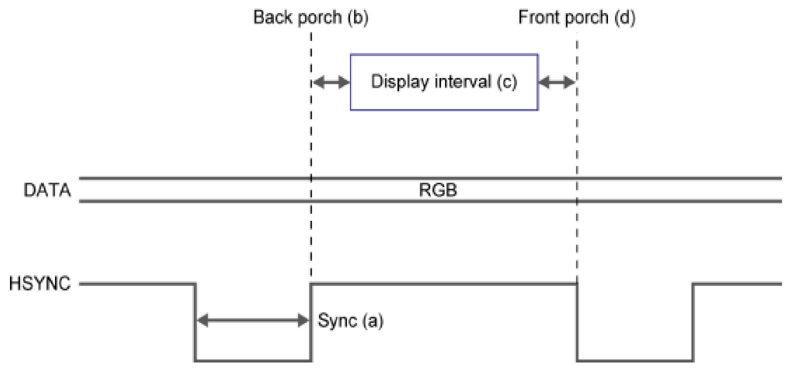


Fig 11. VGA Horizontal timing diagram

4 Resource Budgets

1. How much RAM is needed?

a) Shared Video Buffer

To compute the approximate memory requirement for storing a frame of data, we consider 640 pixels, and there are 480 vertical rows. This means the frame size is 640X480 = 307200. In this case, we would need ~ 307k x8 or 307K bytes. Where 8 is the 8 bits of color data per pixel.

We consider a smaller frame resolution to allow budgeting memory for the incoming video stream and the outgoing VGA display stream to be buffered. In a 624 x 480 resolution, since the camera NTSC standard acquisition happens in an interleaved fashion, we must acquire two frames of size

To optimize the memory requirement, the access pattern that we design is such that the decoded frame written to the camera gets accessed by the VGA controller RTL. The video format we use is the 4:2:2 format, where there are four red bits, two green bits, and two blue bits summing to a total of 8 bits.

The common buffer for Display out and Video-in will be synchronized by a state machine RTL. To display the frame, the 2MBit of frame data will be moved to the VGA Buffer, which will form the lower half of the addresses in the SDRAM.

Stores to the SDRAM are interlaced from the ITU decoder module that gives out 16 bot YCbCr format. But, the reads happen through 2 ports that allow alternate odd and even line access. The ability to read odd/even lines at a given time is obtained from the VGA controller. The read passes through a line buffer (SRAM based shift register) that delays the pixels to allow for writing odd field line first and then the even field line.

Background Storage

→ RGB - 3 bit per pixel

→ 1 Frame = 3 x 3 x 640 x 480 = 2764800 bits ⇒ 337.5 KB

→ Cannot use DRAM due to 2 port deinterlace logic ⇒ SRAM

In order to overcome difficulties with using more than two read ports for reading from a different region in memory and synchronizing it with the time when we detect a red colored pixel, we went ahead and stored a reduced size version of the background in the SRAM. Everytime there is an enable for a detected foreground color the static background image replaces it from the SRAM. This is addressed using the active index of VGA i.e., HCount andV Count after subtracting inactive blank regions.

2. Timing (latency)

Synchronization between the display and video feed will introduce delays due to the different clock rates at which each stores information in the SRAM. Hence, the data available in the RAM is not always the latest. Hence, it must be ensured that the VGA reads only when the RAM is fully populated with the video feed data. This hopes to tackle synchronization-related latency overhead.

5 The Hardware Interface

The FPGA fabric and the ARM core is connected through two Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) bridges. Although these two components can function entirely independently, communication between the ARM core and FPGA fabric can be a bottleneck for the overall system. They are connected with two high-speed 128bit AMBA AXI bus bridges called HPS to FPGA and FPGA to HPS. The data path width for both bridges is not fixed to 128 bits; it can be configured via QSYS to 32, 64, and 128. By having this variable data width, the bridge can be tuned for maximum performance when communication between the FPGA fabric and the HPS L3 occurs.

The different types of interfaces in the DE1 FPGA region are as follows:

1. Composite Video Input – interfaces with NTSC / camcorder
2. VGA Video Output – connects to a VGA monitor for display out
3. PS2 Interface – Interfaces with Keyboards
4. Audio Interface – mic input, line input, speaker output
5. 40 pin general-purpose ports – GPIO-0 (JP1), GPIO-1(JP2)

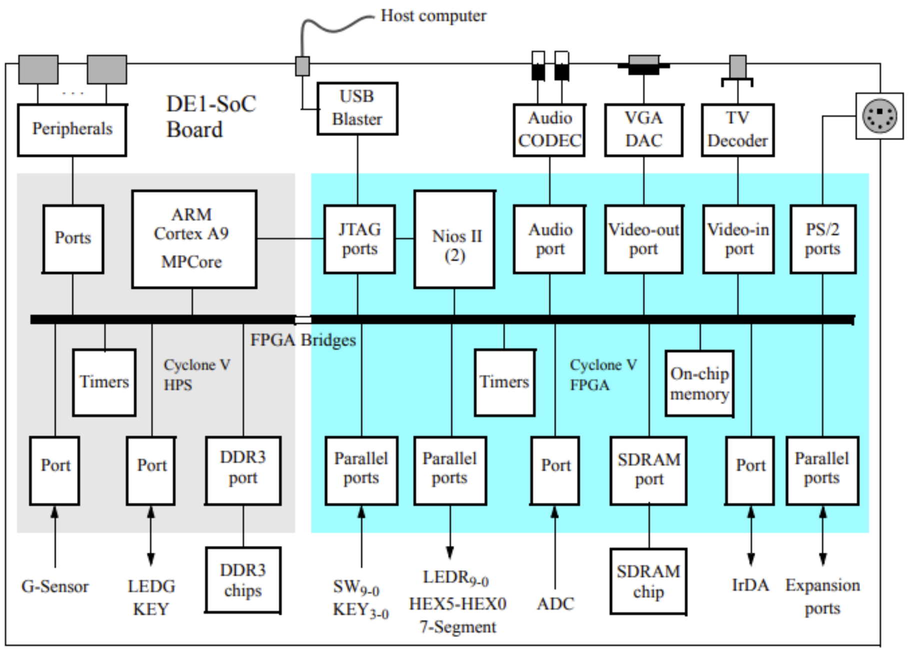


Fig 12. Block diagram of De1-SoC

5.1 Memory:
The DE1-SoC Computer has an SDRAM port and two memory modules implemented using the on-chip memory inside the FPGA. These memories are described below.

5.2 SDRAM
An SDRAM Controller in the FPGA provides an interface to the 64 MB synchronous dynamic RAM (SDRAM) on the DE1-SoC board, organized as 32M x 16 bits. It is accessible by the A9 processor using the word (32-bit), halfword (16-bit), or byte operations and is mapped to the address space 0xC0000000 to 0xC3FFFFFF.
Connections between the FPGA and SDRAM are shown below:

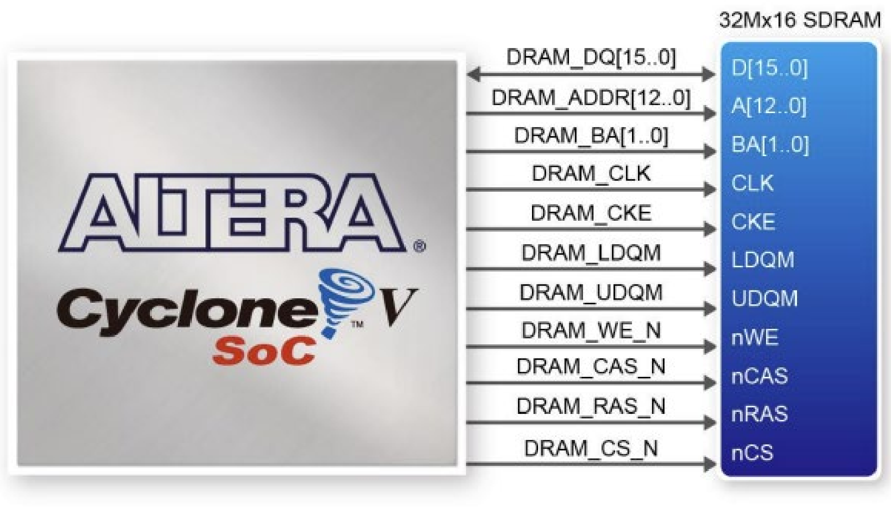


Fig 13: FPGA-SDRAM interface

5.3 On-Chip Memory
The DE1-SoC Computer includes a 256 KB memory implemented inside the FPGA. This memory is organized as 64K x 32 bits and spans addresses 0xC8000000 to 0xC803FFFF. The memory is used as a pixel buffer for the video-out and video-in ports.

5.4 Dual-clock FIFO:
The Dual-Clock FIFO buffers video data entering into the video decoder from the video source and helps transfer a stream between two clock domains. Video streams into the core at the input clock frequency. The data is buffered in a FIFO memory. Then, the data is read out of the FIFO at the output clock frequency and streamed out of the core. The block diagram of the core is represented below:

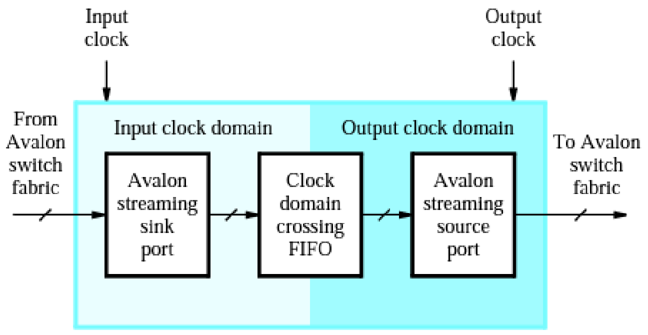

Input clock
Output clock
From Avalon switch fabric
Input clock domain
Output clock domain
To Avalon switch fabric
Avalon streaming sink port
Clock domain crossing FIFO
Avalon streaming source port

Fig 14. Dual-clock FIFO interface

5.8 VGA Port:

The DE1-SoC board has a 15-pin D-SUB connector populated for VGA output. The VGA synchronization signals are generated directly from the Cyclone V SoC FPGA. The Analog Devices ADV7123 triple 10-bit high-speed video DAC (only the higher 8-bits are used) transforms signals from digital to analog to represent three primary colors (red, green, and blue). It can support up to the SXGA standard (1280*1024) with signals transmitted at 100MHz.

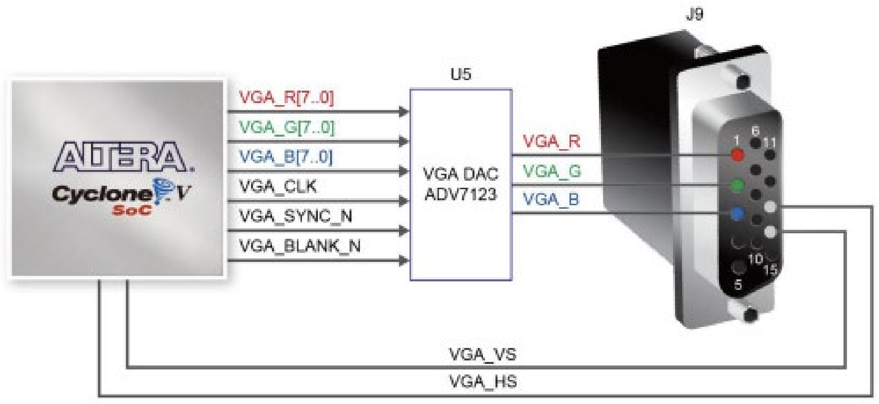


Fig 15. VGA Port Interface

5.9 Video-in decoder:

The chip on the DE1-SoC board is an Analog Devices ADV7180. The ADV7180 is an integrated video decoder that automatically detects and converts standard analog baseband television signals (NTSC, PAL, and SECAM) into 4:2:2 component video data compatible with the 8-bit ITU-R BT.656 interface standard. Video is collected from a composite video source, such as a camcorder, with a composite video RCA jack output. The VGA controller supports a screen resolution of 640 × 480. The registers in the TV decoder can be accessed and set through serial I2C bus by the Cyclone V SoC FPGA or HPS. The video-in controller interface is illustrated in the figure below

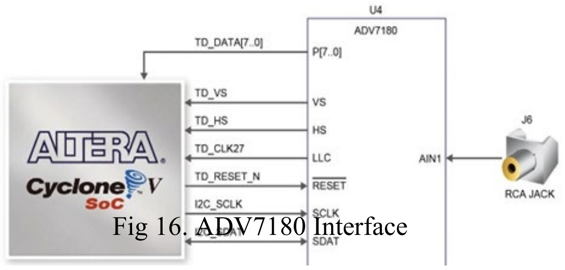


Fig 16. ADV7180 Interface

Signal Name	FPGA Pin No.	Description
TD_Data[0]	Pin_D2	TV Decoder Data[0]
TD_Data[1]	Pin_B1	TV Decoder Data[1]
TD_Data[2]	Pin_E2	TV Decoder Data[2]
TD_Data[3]	Pin_B2	TV Decoder Data[3]
TD_Data[4]	Pin_P1	TV Decoder Data[4]
TD_Data[5]	Pin_E1	TV Decoder Data[5]
TD_Data[6]	Pin_C2	TV Decoder Data[6]
TD_Data[7]	Pin_B3	TV Decoder Data[7]
TD_HS	Pin_A5	TV Decoder H_SYNC
TD_VS	Pin_A3	TV Decoder V_SYNC
TD_CLK27	Pin_H15	TV Decoder Clock Input
TD_RESET	Pin_F6	TV Decoder Reset
TD_SClk	Pin_J12	I2C Clock
TD_SDAT	Pin K12	I2C Data

Table 1- Pin assignments on DE1-SoC FPGA

[TV Decoder Data (7:0)]- 8 bits of video data are connected from the video chip to the FPGA. Pins are assigned according to Table 1.

[TV Decoder H_SYNC] -Horizontal sync pulse generated by the ADV7180 video decoder chip. The pin is set according to Table 1.

[TV Decoder V_SYNC]- Vertical sync pulse generated by the ADV7180 video decoder chip. The pin is set according to Table 1.

[TV Decoder Clock input]- This is a 27 MHZ clock generated by the ADV7180 video chip. The TD_RESET pin must be asserted to an active high logic level to enable the clock.

I2C Data- This is a bi-directional serial data bus pin used to program the internal serial register of the ADV7180 video decoder.

I2C Clock is the serial clock pin used to clock the serial data. The frequency that must be generated is typically below 400 kHz.

5.10 ITU-R 656 Decoder

The ADV7181b decoder detects and converts composite video into digital ITU-R BT.656 format. The format embeds unique codes such as SAV(start of video) and EAV(end of video) within the video stream to avoid transmitting timing signals such as HSYNC and VSYNC. After each start of video (SAV) code, the stream of active data words always start with Chrominance but in our case we discard them and output only a four bit luminance value of each pixel. The ADV chip outputs at a resolution of 720 x 525 at 27Mhz.

The ITU Decoder takes 720 pixels data from the ADV Chip and outputs 640 pixels of data. This is done using a Divider(DIV) module. The ITU Decoder generates the pixel and line numbers which basically are counters. The input iSkip is used to skip pixels. If it is 1 the data valid output(oDVAL) is 0 signifying that the data is not valid. We have another counter(opixelno) which counts only to 640 that is when the data is valid. The field(oField) output represents the field of the pixel data. If field =0 it represents the odd field else if it is 1, it means even field. The output(oTV Y) corresponds to the line numbers. We use the line numbers and the field outputs to calculate the SDRAM address into which we write the pixel data. The outputs oTV X gives the total number of pixels that are output by the ADV chip, which is 720. The pixel number (opixelno) output is the number of valid pixels that is 640. All the outputs are fed to the linebuffer. The iTD DATA is the data input from the ADV chip which contains both luminance and chrominance values of each pixel. It also contains SAV and EAV information. SAV is checked to generate the TV Y and TV outputs.

5.11 Line Buffer

The line buffer receives luminance information from the ITU 656 decoder at 27Mhz. It is housed in the block RAM and stores the luminance information of 1 line i.e. 640 pixels. The luminance information(Y) of each pixel is 8 bits. The line buffer is 80*16 bits in size. The luminance value of 4 pixels that is 24 bits of data is stored in each array index. The Line buffer is a RAM which stores the data from the ITU Decoder and transfers this data over to the SDRAM for the local video. Both the transfers take place on receipt of one line of pixel data.The inputs xpos and ypos correspond to the pixel number and line number respectively and are driven by the opixelno and every other line is stored in the line buffer. This is done by checking the LSB of the inputs xpos and ypos i.e. the 0th bit . We pick up every other line to give the Processor enough time to make data transfers for the network video. Another reason for doing so is we are working with a resolution of 640 x 480. The input data valid gives an indication of whether the data is valid or not. Only after we receive 4 pixels of data do we write into an array index of the linebuffer (since we are skipping every other pixel it is 8 pixels). The signal linebufferfull gives an indication of when the line buffer is full, that is, we have received 640 pixels of data from the ITU decoder. The data is transferred to the SDRAM only when the line buffer is full. The line buffer reads data from the ITU decoder at 27 Mhz. This ensures better video quality. The line number input is used to calculate the memory address in SDRAM that the line buffer is written into once the line buffer is full. This calculation is important as the memory address of the SDRAM into which the line buffer is written determines where it is going to be printed on the VGA. The first line from the odd field that is output from the ITU Decoder is stored in the addresses 0-79 of the SDRAM. The third line(because we are skipping the second line) of the odd field is stored in the addresses 160-239 and so on. The first line of the even field is written to the address 80-159 and the third line is written into the addresses 240-319 and so on.

5.12 TD Detect

This module is used to stabilize the video. It provides the reset signal for the ITU decoder. This module Is required for a stable video.

5.12 Divider

The DIV module takes as input the pixel number (oTV X) from the ITU decoder, divides it by 9 and generates the quotient and remainder. If the remainder is 0 the signal mySkip is set to 1 in the top level entity(top.vhd). This signal is fed as input to the ITU decoder and helps to decide which of the pixels to discard.

6 Milestones

Tentative Date	**Action Item**	**Comments**
03/15	Understanding IP components, VGA adapters, Avalon interconnects, HPS, algorithmic complexity	
03/20	Integrating hardware IPs, adapters, etc. Interfacing Camera with DE-1.	Design documentation of successful IPs integrated
04/01	Mid Review	
04/10	Implement logic for the invisibility cloak	A deeper understanding of video interface, and processing real-time with FPGA
04/15	Store video frame captured on the fly in SRAM or SDRAM	Unsuccessful due to memory and timing constraints
05/01	Adding addition video processing features	
05/05	Testing System and debugging backgrounds	Drafting final report with debug results
05/12	Final Presentation	

7 Additional Features

Feature	Logic	SW [9:0]
Invisible Cloak	RGB range to select b/w two video frames	0
Video OFF (for privacy)	Block the SDRAM output to go into VGA display	1
Captured Background	Select MUX output such that VGA display to output is SRAM output	2
Red Filter Video	Outputting only Red pixels	3
Green Filter Video	Outputting only Green pixels	4
Blue Filter Video	Outputting only Blue pixels	5
Grayscale	29.9% Red + 58.7% Green + 11.4% Blue	6
Invert	Max value - (R + G + B) / 3	7
Low Brightness Video	R, G, B - 200	8
High Brightness Video	R, G, B + 200	9

8 Challenges

- Color Detection
  - Perfecting the threshold for detecting a real-world red shades - varied lighting
- Memory Constraints in SRAM
  - Moved to DRAM
- Synchronization of VGA with NTSC
  - Same frame rate (60Hz)
  - Using Active field lines for sync correction
- DRAM Synchronization
  - Handling reads from 2 different memory blocks – Background & Video Stream

9 Results

Our system is able to produce real-time 10-bit RGB video output with various modes for video processing described in Section 7. On Reset (KEY0), the first video frame is captured into the SRAM of size 3 bit*640*480*3 channels. On turning on SW[0], The stored video frame replaces the sections of real-time video whose pixels have RGB range for the RED shades.

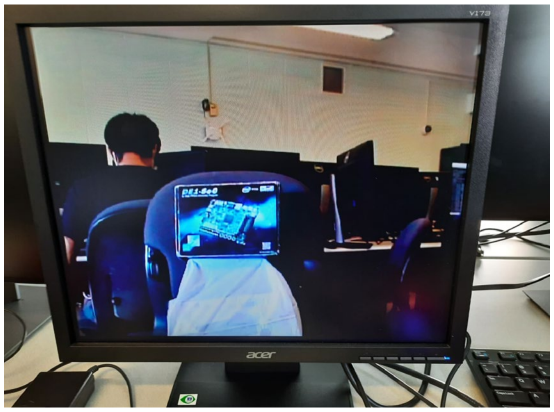


Fig 17. 10-bit RGB Real-time Video output

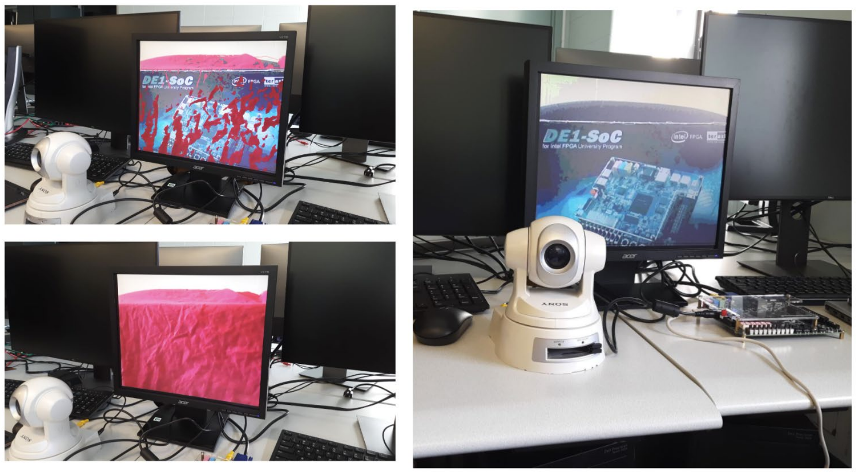


Fig 18. (A) Invisibility Cloak in action (Top left)

(B) Stored Background Video Frame (Right)

(C) Real-time Video when Cloak Function is turned OFF using SW[0] (Bottom left)

We have also included some basic video processing features into our project.

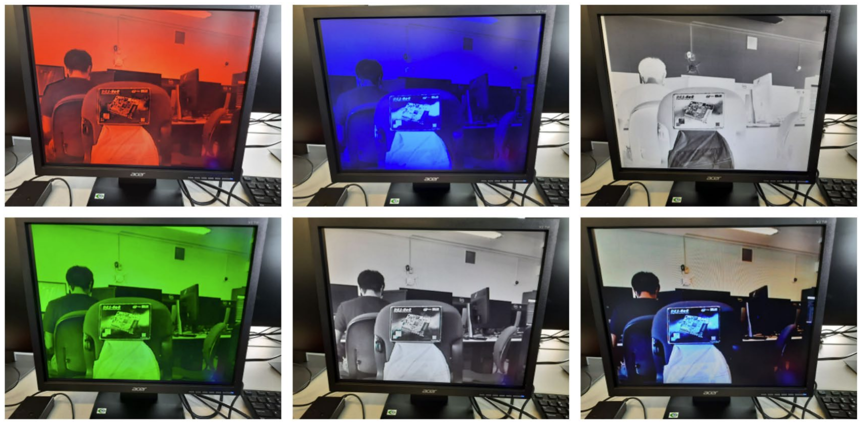


Fig 19. Demo for various Video Processing Modes in following order: Red Filer, Blue Filter, Invert Video, Green Filter, Grayscale Video, Low Brightness.

10 Future Work

- Store and Produce 10-bit RGB invisibility cloak (Currently produces 3-bit RGB).
- Using HSV format for video/image output to better handle brightness and saturation.
- Mimic Video conferencing by sending packets of data over Ethernet.
- Advanced video processing features such as edge detection, Blurring, and Zooming

11 Lessons Learned

From this project, we have learned a very important lesson about memory management and registers. We should prepare a memory budget at the beginning to save ourselves of the time when debugging synthesis failures which the Quartus tool may take one hour to compile and synthesize, only to realize, FPGA is out of memory. Second, signals from unrelated modules will be updated at different moments. We must register them to synchronize the update and prevent glitches. We have had a problem with image distortion that resolves with only two lines of registers added (and two days of debugging). Also, while implementing blocks in hardware can yield more precision and speed, it is much more costly than implementing it in software.

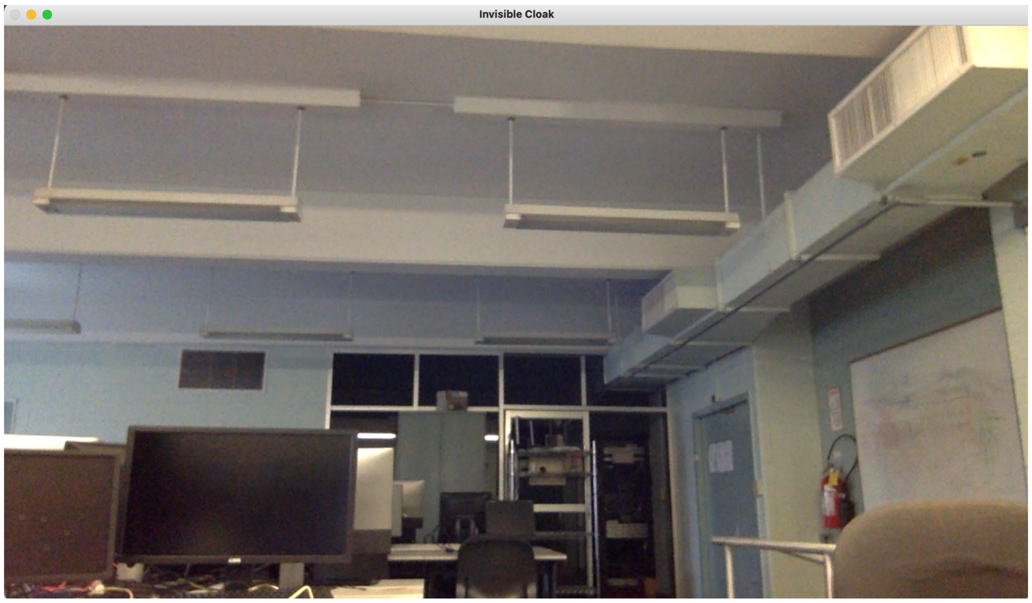
12 Appendix

12.1 Golden Code

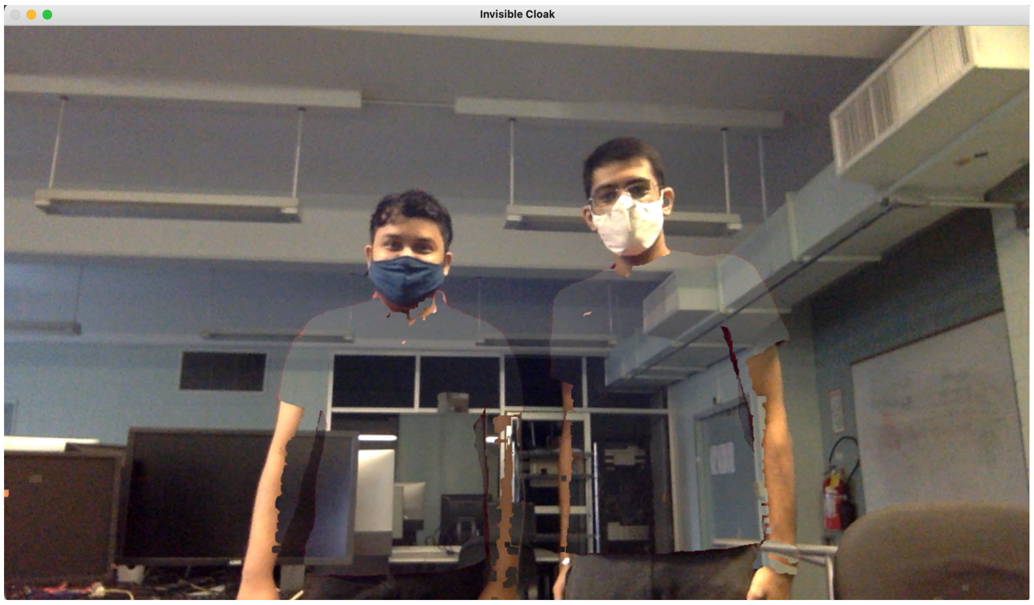
```
# Import Libraries

import numpy as np

import cv2

import time
# camCount=0
#To use webcam  enter 0 and to enter the video path in double quotes
cap = cv2.VideoCapture(0)

time.sleep(3)       # parenthesis has two because the camera needs time to
adjust it self i according to the environment

background = 0
# Capturing the background
# for i in range(60):
ret, background = cap.read()
#capturing image
background = np.flip(background,axis=1)
while(cap.isOpened()):  # Condition for this is when only the webcam is opened
it will only run the code else the code will not run in the background without
the webcam
    ret, img = cap.read() # FPGA
    if not ret:
        break
    # Software _BEGIN_
    img = np.flip(img,axis=1)
    hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV) # FPGA - RGB
    #HSV values
    #setting the values for the cloak
    lower_red = np.array([0,120,70])
    upper_red = np.array([10,255,255])

    mask1 = cv2.inRange(hsv, lower_red,upper_red)

    lower_red = np.array([170,120,70])
    upper_red =  np.array([180,255,255])
    mask2 = cv2.inRange(hsv,lower_red,upper_red)
    mask1 = mask1 + mask2
    mask1 = cv2.morphologyEx(mask1,cv2.MORPH_OPEN,np.ones((3,3),np.uint8),
iterations = 2)
    mask1 = cv2.morphologyEx(mask1, cv2.MORPH_DILATE,np.ones((3,3),np.uint8),
iterations = 1)
    mask2 =cv2.bitwise_not(mask1)
    res1 = cv2.bitwise_and(background, background, mask=mask1)
    res2 = cv2.bitwise_and(img, img, mask=mask2)
     # Software _END_
    final_output = cv2.addWeighted(res1,1,res2,1,0)
    cv2.imshow('Invisible Cloak',final_output)
    k = cv2.waitKey(10)
    if k==27:
        break
cap.release()
Gcv2.destroyAllWindows()
```

12.2 Test Case

Background: Image stream from camera capture - 3 channel (RGB) 720x480 matrix



Output: Processed image after applying the algorithm stated in section 3: RGB 3 channel 720x480 image matrix



12.3 List of Source Files

- **DE1_SoC_TV.v**: Top level hardware module, communicates with video codec via I2C as well as with the segment displays (used frequently in debugging). Passes video stream through decoder blocks. buffers data in SDRAM and presents the output to the VGA controller in RGB.
- **SEG7_LUT_6.v**: segment display for debugging
- **TD_Detect.v** : Detects stable NTSC video frame from ADV 7180 chip and is used to generate resets for other modules
- **Reset_Delay.v**: Generate delayed resets for various modules
- **ITU_656_Decoder.v**: Take in 8-bit data from ADV 7180 and generate 4:2:2 16-bit YCbCr data
- **DIV.v**: Down sample 720 to 640
- **Sdram_Control_4Port.v**: Buffer 4:2:2 16-bit YCbCr video frames with de-interlacing
- **YUV422_to_444.v**: Convert 4:2:2 16-bit YCbCr data into 4:4:4 16-bit YCbCr data
- **YCbCr2RGB.v**: Convert 4:4:4 16-bit YCbCr data into 1-bit RGB pixels
- **VGA_Ctrl.sv**: Controller to display RGB data on monitor with logic for invisibility cloak and other video processing features.
- **Line_Buffer.v**:Shifter logic for delaying one line of video frame used in de-interlacing.
- **I2C_AV_Config.v**: Video decoder setting

12.4 Source Code

```
module DE1_SoC_TV(

      ///////// ADC /////////
      inout                  ADC_CS_N,
      output                 ADC_DIN,
      input                  ADC_DOUT,
      output                 ADC_SCLK,

      ///////// AUD /////////
      input                  AUD_ADCDAT,
      inout                  AUD_ADCLRCK,
      inout                  AUD_BCLK,
      output                 AUD_DACDAT,
      inout                  AUD_DACLRCK,
      output                 AUD_XCK,

      ///////// CLOCK2 /////////
      input                  CLOCK2_50,

      ///////// CLOCK3 /////////
      input                  CLOCK3_50,

      ///////// CLOCK4 /////////
      input                  CLOCK4_50,

      ///////// CLOCK /////////
      input                  CLOCK_50,
      //////// DRAM ////////
      output       [12:0] DRAM_ADDR,
      output       [1:0]  DRAM_BA,
      output              DRAM_CAS_N,
      output              DRAM_CKE,
      output              DRAM_CLK,
      output              DRAM_CS_N,
      inout        [15:0] DRAM_DQ,
      output              DRAM_LDQM,
      output              DRAM_RAS_N,
      output              DRAM_UDQM,
      output              DRAM_WE_N,

      //////// FAN ////////
      output              FAN_CTRL,

      //////// FPGA ////////
      output              FPGA_I2C_SCLK,
      inout               FPGA_I2C_SDAT,

      //////// GPIO ////////
      inout      [35:0]         GPIO_0,
      inout      [35:0]         GPIO_1,


      //////// HEX0 ////////
      output       [6:0]  HEX0,

      //////// HEX1 ////////
      output       [6:0]  HEX1,
      ///////// HEX2 /////////
      output       [6:0]  HEX2,

      ///////// HEX3 /////////
      output       [6:0]  HEX3,

      ///////// HEX4 /////////
      output       [6:0]  HEX4,

      ///////// HEX5 /////////
      output       [6:0]  HEX5,

`ifdef ENABLE_HPS
      ///////// HPS /////////
      inout               HPS_CONV_USB_N,
      output       [14:0] HPS_DDR3_ADDR,
      output       [2:0]  HPS_DDR3_BA,
      output              HPS_DDR3_CAS_N,
      output              HPS_DDR3_CKE,
      output              HPS_DDR3_CK_N,
      output              HPS_DDR3_CK_P,
      output              HPS_DDR3_CS_N,
      output       [3:0]  HPS_DDR3_DM,
      inout        [31:0] HPS_DDR3_DQ,
      inout        [3:0]  HPS_DDR3_DQS_N,
      inout        [3:0]  HPS_DDR3_DQS_P,
      output              HPS_DDR3_ODT,
      output              HPS_DDR3_RAS_N,
      output              HPS_DDR3_RESET_N,
      input              HPS_DDR3_RZQ,
      output             HPS_DDR3_WE_N,
      output             HPS_ENET_GTX_CLK,
      inout              HPS_ENET_INT_N,
      output             HPS_ENET_MDC,
      inout              HPS_ENET_MDIO,
      input              HPS_ENET_RX_CLK,
      input        [3:0] HPS_ENET_RX_DATA,
      input              HPS_ENET_RX_DV,
      output       [3:0] HPS_ENET_TX_DATA,
      output             HPS_ENET_TX_EN,
      inout        [3:0] HPS_FLASH_DATA,
      output             HPS_FLASH_DCLK,
      output             HPS_FLASH_NCSO,
      inout              HPS_GSENSOR_INT,
      inout              HPS_I2C1_SCLK,
      inout              HPS_I2C1_SDAT,
      inout              HPS_I2C2_SCLK,
      inout              HPS_I2C2_SDAT,
      inout              HPS_I2C_CONTROL,
      inout              HPS_KEY,
      inout              HPS_LED,
      inout              HPS_LTC_GPIO,
      output             HPS_SD_CLK,
      inout              HPS_SD_CMD,
      inout        [3:0] HPS_SD_DATA,
      output             HPS_SPIM_CLK,
      input              HPS_SPIM_MISO,
      output             HPS_SPIM_MOSI,
      inout              HPS_SPIM_SS,
      input                  HPS_UART_RX,
      output                 HPS_UART_TX,
      input                  HPS_USB_CLKOUT,
      inout         [7:0]    HPS_USB_DATA,
      input                  HPS_USB_DIR,
      input                  HPS_USB_NXT,
      output                 HPS_USB_STP,
`endif /*ENABLE_HPS*/

      ///////// IRDA /////////
      input                  IRDA_RXD,
      output                 IRDA_TXD,

      ///////// KEY /////////
      input         [3:0]    KEY,

      ///////// LEDR /////////
      output        [9:0]    LEDR,

      ///////// PS2 /////////
      inout                  PS2_CLK,
      inout                  PS2_CLK2,
      inout                  PS2_DAT,
      inout                  PS2_DAT2,

      ///////// SW /////////
      input         [9:0]    SW,

      ///////// TD /////////
      input                  TD_CLK27,
      input      [7:0]  TD_DATA,
      input             TD_HS,
      output             TD_RESET_N,
      input             TD_VS,

      ///////// VGA /////////
      output      [7:0]  VGA_B,
      output             VGA_BLANK_N,
      output             VGA_CLK,
      output      [7:0]  VGA_G,
      output             VGA_HS,
      output      [7:0]  VGA_R,
      output             VGA_SYNC_N,
      output             VGA_VS
);



//=======================================================
//  REG/WIRE declarations
//=======================================================

wire CLK_18_4;
wire CLK_25;

//   For Audio CODEC
wire      AUD_CTRL_CLK; //   For Audio Controller

//   For ITU-R 656 Decoder
wire [15:0]   YCbCr;
wire [9:0]     TV_X;
wire           TV_DVAL;

//   For VGA Controller
wire [9:0]     mRed;
wire [9:0]     mGreen;
wire [9:0]     mBlue;
wire [10:0]    VGA_X;
wire [10:0]    VGA_Y;
wire           VGA_Read; //   VGA data request
wire           m1VGA_Read;    //   Read odd field
wire           m2VGA_Read;    //   Read even field

//   For YUV 4:2:2 to YUV 4:4:4
wire [7:0]     mY;
wire [7:0]     mCb;
wire [7:0]     mCr;

//   For field select
wire [15:0]    mYCbCr;
wire [15:0]    mYCbCr_d;
wire [15:0]    m1YCbCr;
wire [15:0]    m2YCbCr;
wire [15:0]    m3YCbCr;

//   For Delay Timer
wire           TD_Stable;
wire           DLY0;
wire           DLY1;
wire           DLY2;
```

```
//   For Down Sample
wire [3:0]     Remain;
wire [9:0]     Quotient;

wire           mDVAL;

wire [15:0]    m4YCbCr;
wire [15:0]    m5YCbCr;
wire [8:0]     Tmp1,Tmp2;
wire [7:0]     Tmp3,Tmp4;

wire             NTSC;
wire             PAL;

//====================================================================
============
// Structural coding
//====================================================================
============

//   All inout port turn to tri-state

assign    AUD_ADCLRCK    =    AUD_DACLRCK;
assign    GPIO_A    =    36'hzzzzzzzzz;
assign    GPIO_B    =    36'hzzzzzzzzz;

//   Turn On TV Decoder
assign    TD_RESET_N    =    1'b1;
assign    AUD_XCK   =    AUD_CTRL_CLK;

assign    LED  =    VGA_Y;


assign    m1VGA_Read     =    VGA_Y[0]       ?    1'b0      :
     VGA_Read  ;
assign    m2VGA_Read     =    VGA_Y[0]       ?    VGA_Read  :    1'b0
     ;
assign    mYCbCr_d  =    !VGA_Y[0]      ?    m1YCbCr        :
                                                                m2YCbCr
     ;
assign    mYCbCr         =    m5YCbCr;

assign    Tmp1 =    m4YCbCr[7:0]+mYCbCr_d[7:0];
assign    Tmp2 =    m4YCbCr[15:8]+mYCbCr_d[15:8];
assign    Tmp3 =    Tmp1[8:2]+m3YCbCr[7:1];
assign    Tmp4 =    Tmp2[8:2]+m3YCbCr[15:9];
assign    m5YCbCr   =    {Tmp4,Tmp3};

//assign wr_offset =    frame_counter?BASE_OFFSET:0;
//   7 segment LUT
SEG7_LUT_6               u0   (    .oSEG0(HEX0),
                                   .oSEG1(HEX1),
                                   .oSEG2(HEX2),
                                   .oSEG3(HEX3),
                                   .oSEG4(HEX4),
                                   .oSEG5(HEX5),
                                   .iDIG(SW) );
```

```
//   TV Decoder Stable Check
TD_Detect          u2   (    .oTD_Stable(TD_Stable),
                              .oNTSC(NTSC),
                              .oPAL(PAL),
                              .iTD_VS(TD_VS),
                              .iTD_HS(TD_HS),
                              .iRST_N(KEY[0])    );

//   Reset Delay Timer
Reset_Delay             u3   (    .iCLK(CLOCK_50),
                                  .iRST(TD_Stable),
                                  .oRST_0(DLY0),
                                  .oRST_1(DLY1),
                                  .oRST_2(DLY2));

//   ITU-R 656 to YUV 4:2:2
ITU_656_Decoder         u4   (    //   TV Decoder Input
                                  .iTD_DATA(TD_DATA),
                                  //   Position Output
                                  .oTV_X(TV_X),
                                  //   YUV 4:2:2 Output
                                  .oYCbCr(YCbCr),
                                  .oDVAL(TV_DVAL),
                                  //   Control Signals
                                  .iSwap_CbCr(Quotient[0]),
                                  .iSkip(Remain==4'h0),
                                  .iRST_N(DLY1),
                                  .iCLK_27(TD_CLK27) );

//   For Down Sample 720 to 640
```

```
DIV                  u5   (    .aclr(!DLY0),
                                    .clock(TD_CLK27),
                                    .denom(4'h9),
                                    .numer(TV_X),
                                    .quotient(Quotient),
                                    .remain(Remain));

//   SDRAM frame buffer
Sdram_Control_4Port     u6   (    //   HOST Side
                                  .REF_CLK(TD_CLK27),
                                    .CLK_18(AUD_CTRL_CLK),
                                  .RESET_N(DLY0),

                                    //   FIFO Write Side 1
                                    .WR1_DATA(YCbCr),
                                    .WR1(TV_DVAL),
                                    .WR1_FULL(WR1_FULL),
                                    .WR1_ADDR(0),
                                    .WR1_MAX_ADDR((NTSC ? 640 * 507 :
640 * 576)),   //   525-18
                                    .WR1_LENGTH(9'h80),
                                    .WR1_LOAD(!DLY0),
                                    .WR1_CLK(TD_CLK27),

                                    //   FIFO Read Side 1
                               .RD1_DATA(m1YCbCr),
                               .RD1(m1VGA_Read),
                               .RD1_ADDR(NTSC ? 640 * 13 : 640 * 22),
              //   Read odd field and bypass blanking
                                    .RD1_MAX_ADDR(NTSC ? 640 * 253 :
```

```
640 * 262),
                                   .RD1_LENGTH(9'h80),
                              .RD1_LOAD(!DLY0),
                                   .RD1_CLK(TD_CLK27),
                                   //   FIFO Read Side 2
                                  .RD2_DATA(m2YCbCr),
                              .RD2(m2VGA_Read ),
                              .RD2_ADDR(NTSC ? 640 * 267 : 640 *
310),                    //   Read even field and bypass blanking
                                   .RD2_MAX_ADDR(NTSC ? 640 * 507 :
640 * 550),
                                   .RD2_LENGTH(9'h80),
                              .RD2_LOAD(!DLY0),
                                   .RD2_CLK(TD_CLK27),
                                   //   SDRAM Side
                                 .SA(DRAM_ADDR),
                                 .BA(DRAM_BA),
                                 .CS_N(DRAM_CS_N),
                                 .CKE(DRAM_CKE),
                                 .RAS_N(DRAM_RAS_N),
                              .CAS_N(DRAM_CAS_N),
                              .WE_N(DRAM_WE_N),
                                 .DQ(DRAM_DQ),
                              .DQM({DRAM_UDQM,DRAM_LDQM}),
                                   .SDR_CLK(DRAM_CLK) );

//   YUV 4:2:2 to YUV 4:4:4
YUV422_to_444      u7   (    //   YUV 4:2:2 Input
                                   .iYCbCr(mYCbCr),
                                   //   YUV  4:4:4 Output
```

```
                                .oY(mY),
                                .oCb(mCb),
                                .oCr(mCr),
                                //   Control Signals
                                .iX(VGA_X-160),
                                .iCLK(TD_CLK27),
                                .iRST_N(DLY0));

//   YCbCr 8-bit to RGB-10 bit
YCbCr2RGB              u8   (   //   Output Side
                                .Red(mRed),
                                .Green(mGreen),
                                .Blue(mBlue),
                                .oDVAL(mDVAL),
                                //   Input Side
                                .iY(mY),
                                .iCb(mCb),
                                .iCr(mCr),
                                .iDVAL(VGA_Read),
                                //   Control Signal
                                .iRESET(!DLY2),
                                .iCLK(TD_CLK27));

//   VGA Controller
wire [9:0] vga_r10;
wire [9:0] vga_g10;
wire [9:0] vga_b10;
assign VGA_R = vga_r10[9:2];
assign VGA_G = vga_g10[9:2];
assign VGA_B = vga_b10[9:2];
```

```
VGA_Ctrl            u9   (    //   Host Side
                                   .iRed(mRed),
                                   .iGreen(mGreen),
                                   .iBlue(mBlue),
                                   .oCurrent_X(VGA_X),
                                   .oCurrent_Y(VGA_Y),
                                   .oRequest(VGA_Read),
                                   //   VGA Side
                                   .oVGA_R(vga_r10 ),
                                   .oVGA_G(vga_g10 ),
                                   .oVGA_B(vga_b10 ),
                                   .oVGA_HS(VGA_HS),
                                   .oVGA_VS(VGA_VS),
                                   .oVGA_SYNC(VGA_SYNC_N),
                                   .oVGA_BLANK(VGA_BLANK_N),
                                   .oVGA_CLOCK(VGA_CLK),
                                   //   Control Signal
                                   .iCLK(TD_CLK27),
                                   .iRST_N(DLY2),
              .iSW(SW));

//   Line buffer, delay one line
Line_Buffer u10    (    .aclr(!DLY0),
                        .clken(VGA_Read),
                        .clock(TD_CLK27),
                        .shiftin(mYCbCr_d),
                        .shiftout(m3YCbCr));

Line_Buffer u11    (    .aclr(!DLY0),
                        .clken(VGA_Read),
                        .clock(TD_CLK27),
                        .shiftin(m3YCbCr),
                        .shiftout(m4YCbCr));

AUDIO_DAC     u12  (    //   Audio Side
                        .oAUD_BCK(AUD_BCLK),
                        .oAUD_DATA(AUD_DACDAT),
                        .oAUD_LRCK(AUD_DACLRCK),
                        //   Control Signals
                        .iSrc_Select(2'b01),
                               .iCLK_18_4(AUD_CTRL_CLK),
                        .iRST_N(DLY1) );

//   Audio CODEC and video decoder setting
I2C_AV_Config u1   (    //   Host Side
                             .iCLK(CLOCK_50),
                             .iRST_N(KEY[0]),
                             //   I2C Side
                             .I2C_SCLK(FPGA_I2C_SCLK),
                             .I2C_SDAT(FPGA_I2C_SDAT)    );
                             //Store Key[1]

endmodule
```

```
module SEG7_LUT_6 (     oSEG0,oSEG1,oSEG2,oSEG3,oSEG4,oSEG5,iDIG );
input     [31:0]    iDIG;
output    [6:0]     oSEG0,oSEG1,oSEG2,oSEG3,oSEG4,oSEG5;
```

```
SEG7_LUT u0   (    oSEG0,iDIG[3:0]         );
SEG7_LUT u1   (    oSEG1,iDIG[7:4]         );
SEG7_LUT u2   (    oSEG2,iDIG[11:8]   );
SEG7_LUT u3   (    oSEG3,iDIG[15:12]  );
SEG7_LUT u4   (    oSEG4,iDIG[19:16]  );
SEG7_LUT u5   (    oSEG5,iDIG[23:20]  );


endmodule
```

```
module SEG7_LUT    (    oSEG,iDIG );
input     [3:0]    iDIG;
output    [6:0]    oSEG;
reg       [6:0]    oSEG;

always @(iDIG)
begin
         case(iDIG)
         4'h1: oSEG = 7'b1111001;    // ---t----
         4'h2: oSEG = 7'b0100100;    // |   |
         4'h3: oSEG = 7'b0110000;    // lt      rt
         4'h4: oSEG = 7'b0011001;    // |   |
         4'h5: oSEG = 7'b0010010;    // ---m----
         4'h6: oSEG = 7'b0000010;    // |   |
         4'h7: oSEG = 7'b1111000;    // lb      rb
         4'h8: oSEG = 7'b0000000;    // |   |
         4'h9: oSEG = 7'b0011000;    // ---b----
         4'ha: oSEG = 7'b0001000;
```

```
          4'hb: oSEG = 7'b0000011;
          4'hc: oSEG = 7'b1000110;
          4'hd: oSEG = 7'b0100001;
          4'he: oSEG = 7'b0000110;
          4'hf: oSEG = 7'b0001110;
          4'h0: oSEG = 7'b1000000;
          endcase
end

endmodule
```

```
module TD_Detect(
          oTD_Stable,
          oNTSC,
          oPAL,
                        iTD_VS,
                        iTD_HS,
                        iRST_N    );
input         iTD_VS;
input         iTD_HS;
input         iRST_N;
output    oTD_Stable;
output   oNTSC;
output   oPAL;
reg           NTSC;
reg           PAL;
reg           Pre_VS;
reg  [7:0]    Stable_Cont;
```

```verilog
assign   oTD_Stable    =    NTSC || PAL;
assign  oNTSC   = NTSC;
assign  oPAL    = PAL;

always@(posedge iTD_HS or negedge iRST_N)
     if(!iRST_N)
     begin
          Pre_VS          <= 1'b0;
          Stable_Cont    <=   4'h0;
          NTSC <=   1'b0;
          PAL       <=   1'b0;
     end
     else
     begin
          Pre_VS    <=   iTD_VS;
          if(!iTD_VS)
            Stable_Cont <=   Stable_Cont+1'b1;
          else
            Stable_Cont <=   0;

          if({Pre_VS,iTD_VS}==2'b01)
          begin
               if((Stable_Cont>=4 && Stable_Cont<=14))
                 NTSC    <=   1'b1;
               else
                 NTSC    <=   1'b0;

               if((Stable_Cont>=8'h14 && Stable_Cont<=8'h1f))
                 PAL     <=   1'b1;
               else
```

```
                PAL     <=   1'b0;
          end
     end

endmodule
```

```
module    Reset_Delay(iCLK,iRST,oRST_0,oRST_1,oRST_2);
input          iCLK;
input          iRST;
output reg     oRST_0;
output reg     oRST_1;
output reg     oRST_2;

reg  [21:0]    Cont;

always@(posedge iCLK or negedge iRST)
begin
     if(!iRST)
     begin
          Cont <=   0;
          oRST_0    <=   0;
          oRST_1    <=   0;
          oRST_2    <=   0;
     end
     else
     begin
          if(Cont!=22'h3FFFFF)
          Cont <=   Cont+1;
          if(Cont>=22'h1FFFFF)
          oRST_0    <=   1;
          if(Cont>=22'h2FFFFF)
          oRST_1    <=   1;
          if(Cont>=22'h3FFFFF)
          oRST_2    <=   1;
     end
end

endmodule
```

```
module    ITU_656_Decoder(   //    TV Decoder Input
                                 iTD_DATA,
                                 //    Position Output
                                 oTV_X,
                                 oTV_Y,
                                 oTV_Cont,
                                 //    YUV 4:2:2 Output
                                 oYCbCr,
                                 oDVAL,
                                 //    Control Signals
                                 iSwap_CbCr,
                                 iSkip,
                                 iRST_N,
                                 iCLK_27   );
input     [7:0]     iTD_DATA;
input               iSwap_CbCr;
input               iSkip;
input               iRST_N;
input               iCLK_27;
```

```
output    [15:0]    oYCbCr;
output    [9:0]     oTV_X;
output    [9:0]     oTV_Y;
output    [31:0]    oTV_Cont;
output              oDVAL;

//   For detection
reg       [23:0]    Window;        //   Sliding window register
reg       [17:0]    Cont;          //   Counter
reg                 Active_Video;
reg                 Start;
reg                 Data_Valid;
reg                 Pre_Field;
reg                 Field;
wire           SAV;
reg                 FVAL;
reg       [9:0]     TV_Y;
reg       [31:0]    Data_Cont;

//   For ITU-R 656 to ITU-R 601
reg       [7:0]     Cb;
reg       [7:0]     Cr;
reg       [15:0]    YCbCr;

assign    oTV_X     =    Cont>>1;
assign    oTV_Y     =    TV_Y;
assign    oYCbCr    =    YCbCr;
assign    oDVAL     =    Data_Valid;
assign    SAV       =    (Window==24'hFF0000)&(iTD_DATA[4]==1'b0);
assign    oTV_Cont= Data_Cont;
```

```
always@(posedge iCLK_27 or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          //   Register initial
          Active_Video<=1'b0;
          Start          <=   1'b0;
          Data_Valid     <=   1'b0;
          Pre_Field <=   1'b0;
          Field          <=   1'b0;
          Window         <=   24'h0;
          Cont      <=   18'h0;
          Cb             <=   8'h0;
          Cr             <=   8'h0;
          YCbCr          <=   16'h0;
          FVAL      <=   1'b0;
          TV_Y      <=   10'h0;
          Data_Cont <=   32'h0;
     end
     else
     begin
          //   Sliding window
          Window    <=   {Window[15:0],iTD_DATA};
          //   Active data counter
          if(SAV)
          Cont <=   18'h0;
          else if(Cont<1440)
          Cont <=   Cont+1'b1;
          //   Check the video data is active?
          if(SAV)
          Active_Video<=1'b1;
          else if(Cont==1440)
          Active_Video<=1'b0;
          //   Is the frame starting?
          Pre_Field <=   Field;
          if({Pre_Field,Field}==2'b10)
          Start          <=   1'b1;
          //   Field and frame valid check
          if(Window==24'hFF0000)
          begin
               FVAL <=   !iTD_DATA[5];
               Field     <=   iTD_DATA[6];
          end
          //   ITU-R 656 to ITU-R 601
          if(iSwap_CbCr)
          begin
               case(Cont[1:0])          //   Swap
               0:   Cb        <=    iTD_DATA;
               1:   YCbCr     <=   {iTD_DATA,Cr};
               2:   Cr        <=    iTD_DATA;
               3:   YCbCr     <=   {iTD_DATA,Cb};
               endcase
          end
          else
          begin
               case(Cont[1:0])          //   Normal
               0:   Cb        <=    iTD_DATA;
               1:   YCbCr     <=   {iTD_DATA,Cb};
               2:   Cr        <=    iTD_DATA;
```

```
            3:   YCbCr     <=   {iTD_DATA,Cr};
            endcase
        end
        //   Check data valid
        if(      Start                    //   Frame Start?
            &&   FVAL                 //   Frame valid?
            &&   Active_Video         //   Active video?
            &&   Cont[0]                  //   Complete ITU-R 601?
            &&   !iSkip   )               //   Is it a non-skip
pixel?
        Data_Valid    <=   1'b1;
        else
        Data_Valid    <=   1'b0;
        //   TV decoder line counter for one field
        if(FVAL && SAV)
        TV_Y<=    TV_Y+1;
        if(!FVAL)
        TV_Y<=    0;
        //   Data counter for one field
        if(!FVAL)
        Data_Cont <=   0;
        if(Data_Valid)
        Data_Cont <=   Data_Cont+1'b1;
    end
end

endmodule
```

```
`timescale 1 ps / 1 ps
// synopsys translate_on
```

```
module DIV (
     aclr,
     clock,
     denom,
     numer,
     quotient,
     remain);

     input       aclr;
     input       clock;
     input     [3:0]  denom;
     input     [9:0]  numer;
     output    [9:0]  quotient;
     output    [3:0]  remain;

     wire [3:0] sub_wire0;
     wire [9:0] sub_wire1;
     wire [3:0] remain = sub_wire0[3:0];
     wire [9:0] quotient = sub_wire1[9:0];

     lpm_divide    LPM_DIVIDE_component (
                   .aclr (aclr),
                   .clock (clock),
                   .denom (denom),
                   .numer (numer),
                   .remain (sub_wire0),
                   .quotient (sub_wire1),
                   .clken (1'b1));
     defparam
          LPM_DIVIDE_component.lpm_drepresentation = "UNSIGNED",
          LPM_DIVIDE_component.lpm_hint =
"LPM_REMAINDERPOSITIVE=TRUE",
          LPM_DIVIDE_component.lpm_nrepresentation = "UNSIGNED",
          LPM_DIVIDE_component.lpm_pipeline = 1,
          LPM_DIVIDE_component.lpm_type = "LPM_DIVIDE",
          LPM_DIVIDE_component.lpm_widthd = 4,
          LPM_DIVIDE_component.lpm_widthn = 10;


endmodule
```

```
module Sdram_Control_4Port(
          //   HOST Side
        REF_CLK,
        RESET_N,
          CLK,
          //   FIFO Write Side 1
        WR1_DATA,
          WR1,
          WR1_ADDR,
          WR1_MAX_ADDR,
          WR1_LENGTH,
          WR1_LOAD,
          WR1_CLK,
          WR1_FULL,
          WR1_USE,
          //   FIFO Write Side 2
        WR2_DATA,
          WR2,
          WR2_ADDR,
          WR2_MAX_ADDR,
          WR2_LENGTH,
          WR2_LOAD,
          WR2_CLK,
          WR2_FULL,
          WR2_USE,
          //   FIFO Read Side 1
         RD1_DATA,
          RD1,
          RD1_ADDR,
          RD1_MAX_ADDR,
          RD1_LENGTH,
          RD1_LOAD,
          RD1_CLK,
          RD1_EMPTY,
          RD1_USE,
          //   FIFO Read Side 2
         RD2_DATA,
          RD2,
          RD2_ADDR,
          RD2_MAX_ADDR,
          RD2_LENGTH,
          RD2_LOAD,
          RD2_CLK,
          RD2_EMPTY,
          RD2_USE,
          //   SDRAM Side
         SA,
         BA,
         CS_N,
         CKE,
         RAS_N,
         CAS_N,
         WE_N,
         DQ,
         DQM,
          SDR_CLK,
          CLK_18
         );


`include        "Sdram_Params.h"
//   HOST Side
input                           REF_CLK;                //System
Clock
input                           RESET_N;                //System
Reset
//   FIFO Write Side 1
input   [`DSIZE-1:0]            WR1_DATA;               //Data
input
input                                WR1;
     //Write Request
input    [`ASIZE-1:0]            WR1_ADDR;              //Write
start address
input    [`ASIZE-1:0]            WR1_MAX_ADDR;          //Write max
address
input    [8:0]                       WR1_LENGTH;
     //Write length
input                                WR1_LOAD;
     //Write register load & fifo clear
```

```
input                                      WR1_CLK;
     //Write fifo clock
output                                     WR1_FULL;
     //Write fifo full
output    [15:0]                           WR1_USE;
     //Write fifo used
//   FIFO Write Side 2
input    [`DSIZE-1:0]                WR2_DATA;                //Data
input
input                                      WR2;
     //Write Request
input     [`ASIZE-1:0]                WR2_ADDR;               //Write
start address
input     [`ASIZE-1:0]                WR2_MAX_ADDR;           //Write max
address
input     [8:0]                            WR2_LENGTH;
     //Write length
input                                      WR2_LOAD;
     //Write register load & fifo clear
input                                      WR2_CLK;
     //Write fifo clock
output                                     WR2_FULL;
     //Write fifo full
output    [15:0]                           WR2_USE;
     //Write fifo used
//   FIFO Read Side 1
output   [`DSIZE-1:0]                RD1_DATA;                //Data
output
input                                      RD1;                     //Read
Request
```

```
input     [`ASIZE-1:0]            RD1_ADDR;                //Read
start address
input     [`ASIZE-1:0]            RD1_MAX_ADDR;            //Read max
address
input     [8:0]                        RD1_LENGTH;
     //Read length
input                                  RD1_LOAD;                //Read
register load & fifo clear
input                                  RD1_CLK;                 //Read
fifo clock
output                                 RD1_EMPTY;
     //Read fifo empty
output    [15:0]                       RD1_USE;                 //Read
fifo used
//   FIFO Read Side 2
output   [`DSIZE-1:0]            RD2_DATA;                //Data
output
input                                  RD2;                     //Read
Request
input     [`ASIZE-1:0]            RD2_ADDR;                //Read
start address
input     [`ASIZE-1:0]            RD2_MAX_ADDR;            //Read max
address
input     [8:0]                        RD2_LENGTH;
     //Read length
input                                  RD2_LOAD;                //Read
register load & fifo clear
input                                  RD2_CLK;                 //Read
fifo clock
output                                 RD2_EMPTY;
```

```
     //Read fifo empty
output    [15:0]                     RD2_USE;              //Read
fifo used
//   SDRAM Side
output   [11:0]                SA;                      //SDRAM
address output
output   [1:0]                 BA;                      //SDRAM
bank address
output   [1:0]                 CS_N;                    //SDRAM
Chip Selects
output                         CKE;                     //SDRAM
clock enable
output                         RAS_N;                   //SDRAM Row
address Strobe
output                         CAS_N;                   //SDRAM
Column address Strobe
output                         WE_N;                    //SDRAM
write enable
inout    [`DSIZE-1:0]          DQ;                      //SDRAM
data bus
output   [`DSIZE/8-1:0]        DQM;                     //SDRAM
data mask lines
output                               SDR_CLK;
     //SDRAM clock
//   Internal Registers/Wires
//   Controller
reg       [`ASIZE-1:0]          mADDR;
     //Internal address
reg       [8:0]                      mLENGTH;
     //Internal length
```

```
reg        [`ASIZE-1:0]            rWR1_ADDR;
     //Register write address
reg        [`ASIZE-1:0]            rWR2_ADDR;
     //Register write address
reg        [`ASIZE-1:0]            rRD1_ADDR;
     //Register read address
reg        [`ASIZE-1:0]            rRD2_ADDR;
     //Register read address
reg        [1:0]                        WR_MASK;
     //Write port active mask
reg        [1:0]                        RD_MASK;                //Read
port active mask
reg                                     mWR_DONE;               //Flag
write done, 1 pulse SDR_CLK
reg                                     mRD_DONE;               //Flag
read done, 1 pulse SDR_CLK
reg                                     mWR,Pre_WR;
     //Internal WR edge capture
reg                                     mRD,Pre_RD;
     //Internal RD edge capture
reg  [9:0]                         ST;
     //Controller status
reg        [1:0]                        CMD;
     //Controller command
reg                                     PM_STOP;                //Flag
page mode stop
reg                                     PM_DONE;                //Flag
page mode done
reg                                     Read;
     //Flag read active
```

```
reg                                         Write;
     //Flag write active
reg      [`DSIZE-1:0]                  mDATAOUT;
//Controller Data output
wire     [`DSIZE-1:0]                  mDATAIN;
//Controller Data input
wire     [`DSIZE-1:0]                  mDATAIN1;
//Controller Data input 1
wire     [`DSIZE-1:0]                  mDATAIN2;
//Controller Data input 2
wire                                   CMDACK;
//Controller command acknowledgement
//   DRAM Control
reg       [`DSIZE/8-1:0]                DQM;                    //SDRAM
data mask lines
reg      [11:0]                        SA;                     //SDRAM
address output
reg      [1:0]                         BA;                     //SDRAM
bank address
reg      [1:0]                         CS_N;                   //SDRAM
Chip Selects
reg                                    CKE;                    //SDRAM
clock enable
reg                                    RAS_N;                  //SDRAM Row
address Strobe
reg                                    CAS_N;                  //SDRAM
Column address Strobe
reg                                    WE_N;                   //SDRAM
write enable
wire     [`DSIZE-1:0]                  DQOUT;
```

```
     //SDRAM data out link
wire      [`DSIZE/8-1:0]          IDQM;                    //SDRAM
data mask lines
wire     [11:0]                  ISA;                     //SDRAM
address output
wire     [1:0]                   IBA;                     //SDRAM
bank address
wire     [1:0]                   ICS_N;                   //SDRAM
Chip Selects
wire                             ICKE;                    //SDRAM
clock enable
wire                             IRAS_N;                  //SDRAM Row
address Strobe
wire                             ICAS_N;                  //SDRAM
Column address Strobe
wire                             IWE_N;                   //SDRAM
write enable
//   FIFO Control
reg                                   OUT_VALID;
     //Output data request to read side fifo
reg                                   IN_REQ;
     //Input  data request to write side fifo
wire [15:0]                       write_side_fifo_rusedw1;
wire [15:0]                       read_side_fifo_wusedw1;
wire [15:0]                       write_side_fifo_rusedw2;
wire [15:0]                       read_side_fifo_wusedw2;
//   DRAM Internal Control
wire     [`ASIZE-1:0]            saddr;
wire                             load_mode;
wire                             nop;
wire                                reada;
wire                                writea;
wire                                refresh;
wire                                precharge;
wire                                oe;
wire                                 ref_ack;
wire                                 ref_req;
wire                                 init_req;
wire                                 cm_ack;
wire                                 active;
output                                CLK;
output wire                         CLK_18;

Sdram_PLL sdram_pll1    (
                   .refclk(REF_CLK),
                   .rst(1'b0),
                   .outclk_0(CLK),
                   .outclk_1(SDR_CLK),
                   .outclk_2(CLK_18)
                   );

control_interface control1 (
                .CLK(CLK),
                .RESET_N(RESET_N),
                .CMD(CMD),
                .ADDR(mADDR),
                .REF_ACK(ref_ack),
                .CM_ACK(cm_ack),
                .NOP(nop),
                .READA(reada),
                .WRITEA(writea),
                .REFRESH(refresh),
                .PRECHARGE(precharge),
                .LOAD_MODE(load_mode),
                .SADDR(saddr),
                .REF_REQ(ref_req),
                   .INIT_REQ(init_req),
                .CMD_ACK(CMDACK)
                );

command command1(
                .CLK(CLK),
                .RESET_N(RESET_N),
                .SADDR(saddr),
                .NOP(nop),
                .READA(reada),
                .WRITEA(writea),
                .REFRESH(refresh),
                   .LOAD_MODE(load_mode),
                .PRECHARGE(precharge),
                .REF_REQ(ref_req),
                   .INIT_REQ(init_req),
                .REF_ACK(ref_ack),
                .CM_ACK(cm_ack),
                .OE(oe),
                   .PM_STOP(PM_STOP),
                   .PM_DONE(PM_DONE),
                .SA(ISA),
                .BA(IBA),
                .CS_N(ICS_N),
                .CKE(ICKE),
                .RAS_N(IRAS_N),
                .CAS_N(ICAS_N),
                .WE_N(IWE_N)
                );

sdr_data_path data_path1(
                .CLK(CLK),
                .RESET_N(RESET_N),
                .DATAIN(mDATAIN),
                .DM(2'b00),
                .DQOUT(DQOUT),
                .DQM(IDQM)
                );

Sdram_WR_FIFO write_fifo1(
                   .data(WR1_DATA),
                   .wrreq(WR1),
                   .wrclk(WR1_CLK),
                   .aclr(WR1_LOAD),
                   .rdreq(IN_REQ&WR_MASK[0]),
                   .rdclk(CLK),
                   .q(mDATAIN1),
                   .wrfull(WR1_FULL),
                   .wrusedw(WR1_USE),
                   .rdusedw(write_side_fifo_rusedw1)
                   );

Sdram_WR_FIFO write_fifo2(
                   .data(WR2_DATA),
                   .wrreq(WR2),
                   .wrclk(WR2_CLK),
                   .aclr(WR2_LOAD),
                   .rdreq(IN_REQ&WR_MASK[1]),
                   .rdclk(CLK),
                   .q(mDATAIN2),
                   .wrfull(WR2_FULL),
                   .wrusedw(WR2_USE),
                   .rdusedw(write_side_fifo_rusedw2)
                   );

assign   mDATAIN  =   (WR_MASK[0])  ?   mDATAIN1  :
                                             mDATAIN2  ;

Sdram_RD_FIFO read_fifo1(
                   .data(mDATAOUT),
                   .wrreq(OUT_VALID&RD_MASK[0]),
                   .wrclk(CLK),
                   .aclr(RD1_LOAD),
                   .rdreq(RD1),
                   .rdclk(RD1_CLK),
                   .q(RD1_DATA),
                   .wrusedw(read_side_fifo_wusedw1),
                   .rdempty(RD1_EMPTY),
                   .rdusedw(RD1_USE)
                   );

Sdram_RD_FIFO read_fifo2(
                   .data(mDATAOUT),
                   .wrreq(OUT_VALID&RD_MASK[1]),
                    .wrclk(CLK),
                    .aclr(RD2_LOAD),
                    .rdreq(RD2),
                    .rdclk(RD2_CLK),
                    .q(RD2_DATA),
                    .wrusedw(read_side_fifo_wusedw2),
                    .rdempty(RD2_EMPTY),
                    .rdusedw(RD2_USE)
                    );

always @(posedge CLK)
begin
     SA       <= (ST==SC_CL+mLENGTH)            ?    12'h200   :
     ISA;
    BA       <= IBA;
    CS_N     <= ICS_N;
    CKE      <= ICKE;
    RAS_N    <= (ST==SC_CL+mLENGTH)             ?    1'b0 :
     IRAS_N;
    CAS_N    <= (ST==SC_CL+mLENGTH)             ?    1'b1 :
     ICAS_N;
    WE_N     <= (ST==SC_CL+mLENGTH)             ?    1'b0 :    IWE_N;
     PM_STOP   <= (ST==SC_CL+mLENGTH)           ?    1'b1 :    1'b0;
     PM_DONE   <= (ST==SC_CL+SC_RCD+mLENGTH+2)  ?    1'b1 :    1'b0;
     DQM       <= ( active && (ST>=SC_CL) )?    (
     ((ST==SC_CL+mLENGTH) && Write)?  2'b11     :    2'b00     )    :
     2'b11     ;
     mDATAOUT<= DQ;
end
```

```
assign  DQ = oe ? DQOUT : `DSIZE'hzzzz;
assign   active    =    Read | Write;

always@(posedge CLK or negedge RESET_N)
begin
     if(RESET_N==0)
     begin
          CMD              <=  0;
          ST               <=  0;
          Pre_RD           <=  0;
          Pre_WR           <=  0;
          Read      <=   0;
          Write            <=   0;
          OUT_VALID <=   0;
          IN_REQ           <=   0;
          mWR_DONE  <=   0;
          mRD_DONE  <=   0;
     end
     else
     begin
          Pre_RD    <=   mRD;
          Pre_WR    <=   mWR;
          case(ST)
          0:   begin
                    if({Pre_RD,mRD}==2'b01)
                    begin
                         Read <=   1;
                         Write     <=   0;
                         CMD       <=   2'b01;
                         ST        <=   1;
```

```
            end
            else if({Pre_WR,mWR}==2'b01)
            begin
                Read <=   0;
                Write     <=   1;
                CMD       <=   2'b10;
                ST        <=   1;
            end
        end
    1:  begin
            if(CMDACK==1)
            begin
                CMD<=2'b00;
                ST<=2;
            end
        end
    default:
        begin
            if(ST!=SC_CL+SC_RCD+mLENGTH+1)
            ST<=ST+1;
            else
            ST<=0;
        end
    endcase

    if(Read)
    begin
        if(ST==SC_CL+SC_RCD+1)
        OUT_VALID <=   1;
        else if(ST==SC_CL+SC_RCD+mLENGTH+1)
```

```
                begin
                    OUT_VALID <=    0;
                    Read      <=    0;
                    mRD_DONE  <=    1;
                end
            end
            else
            mRD_DONE  <=    0;

            if(Write)
            begin
                if(ST==SC_CL-1)
                IN_REQ    <=    1;
                else if(ST==SC_CL+mLENGTH-1)
                IN_REQ    <=    0;
                else if(ST==SC_CL+SC_RCD+mLENGTH)
                begin
                    Write     <=    0;
                    mWR_DONE<=      1;
                end
            end
            else
            mWR_DONE<=      0;

        end
end
//   Internal Address & Length Control
always@(posedge CLK or negedge RESET_N)
begin
    if(!RESET_N)
        begin
            rWR1_ADDR     <=   WR1_ADDR;
            rWR2_ADDR     <=   WR2_ADDR;
            rRD1_ADDR     <=   RD1_ADDR;
            rRD2_ADDR     <=   RD2_ADDR;
        end
        else
        begin
            //  Write Side 1
            if(WR1_LOAD)
                rWR1_ADDR <=   WR1_ADDR;
            else if(mWR_DONE&WR_MASK[0])
            begin
                if(rWR1_ADDR<WR1_MAX_ADDR-WR1_LENGTH)
                rWR1_ADDR <=   rWR1_ADDR+WR1_LENGTH;
                else
                rWR1_ADDR <=   WR1_ADDR;
            end
            //  Write Side 2
            if(WR2_LOAD)
                rWR2_ADDR <=   WR2_ADDR;
            else if(mWR_DONE&WR_MASK[1])
            begin
                if(rWR2_ADDR<WR2_MAX_ADDR-WR2_LENGTH)
                rWR2_ADDR <=   rWR2_ADDR+WR2_LENGTH;
                else
                rWR2_ADDR <=   WR2_ADDR;
            end
            //  Read Side 1
            if(RD1_LOAD)
```

```
                rRD1_ADDR <=    RD1_ADDR;
            else if(mRD_DONE&RD_MASK[0])
            begin
                if(rRD1_ADDR<RD1_MAX_ADDR-RD1_LENGTH)
                rRD1_ADDR <=    rRD1_ADDR+RD1_LENGTH;
                else
                rRD1_ADDR <=    RD1_ADDR;
            end
            //  Read Side 2
            if(RD2_LOAD)
                rRD2_ADDR <=    RD2_ADDR;
            else if(mRD_DONE&RD_MASK[1])
            begin
                if(rRD2_ADDR<RD2_MAX_ADDR-RD2_LENGTH)
                rRD2_ADDR <=    rRD2_ADDR+RD2_LENGTH;
                else
                rRD2_ADDR <=    RD2_ADDR;
            end
        end
end
//  Auto Read/Write Control
always@(posedge CLK or negedge RESET_N)
begin
    if(!RESET_N)
    begin
        mWR       <=   0;
        mRD       <=   0;
        mADDR     <=   0;
        mLENGTH   <=   0;
        WR_MASK  <=    0;
```

```
            RD_MASK <=    0;
        end
        else
        begin
            if( (mWR==0) && (mRD==0) && (ST==0) &&
                (WR_MASK==0)  &&   (RD_MASK==0) &&
                (WR1_LOAD==0) &&   (RD1_LOAD==0) &&
                (WR2_LOAD==0) &&   (RD2_LOAD==0) )
            begin
                //  Read Side 1
                if( (read_side_fifo_wusedw1 < RD1_LENGTH) )
                begin
                    mADDR    <=   rRD1_ADDR;
                    mLENGTH  <=   RD1_LENGTH;
                    WR_MASK  <=   2'b00;
                    RD_MASK  <=   2'b01;
                    mWR      <=   0;
                    mRD      <=   1;
                end
                //  Read Side 2
                else if( (read_side_fifo_wusedw2 < RD2_LENGTH) )
                begin
                    mADDR    <=   rRD2_ADDR;
                    mLENGTH  <=   RD2_LENGTH;
                    WR_MASK  <=   2'b00;
                    RD_MASK  <=   2'b10;
                    mWR      <=   0;
                    mRD      <=   1;
                end
                //  Write Side 1
```

```
              else if( (write_side_fifo_rusedw1 >= WR1_LENGTH) &&
(WR1_LENGTH!=0) )
              begin
                   mADDR     <=   rWR1_ADDR;
                   mLENGTH   <=   WR1_LENGTH;
                   WR_MASK   <=   2'b01;
                   RD_MASK   <=   2'b00;
                   mWR       <=   1;
                   mRD       <=   0;
              end
              //   Write Side 2
              else if( (write_side_fifo_rusedw2 >= WR2_LENGTH) &&
(WR2_LENGTH!=0) )
              begin
                   mADDR     <=   rWR2_ADDR;
                   mLENGTH   <=   WR2_LENGTH;
                   WR_MASK   <=   2'b10;
                   RD_MASK   <=   2'b00;
                   mWR       <=   1;
                   mRD       <=   0;
              end
         end
         if(mWR_DONE)
         begin
              WR_MASK   <=   0;
              mWR       <=   0;
         end
         if(mRD_DONE)
         begin
              RD_MASK   <=   0;
```

```
               mRD       <=   0;
          end
     end
end

endmodule
```

```
`timescale 1 ps / 1 ps
module Sdram_PLL (
          input  wire  refclk,   //  refclk.clk
          input  wire  rst,      //   reset.reset
          output wire  outclk_0, // outclk0.clk
          output wire  outclk_1, // outclk1.clk
          output wire  outclk_2  // outclk2.clk
     );

     Sdram_PLL_0002 sdram_pll_inst (
          .refclk   (refclk),   //  refclk.clk
          .rst      (rst),      //   reset.reset
          .outclk_0 (outclk_0), // outclk0.clk
          .outclk_1 (outclk_1), // outclk1.clk
          .outclk_2 (outclk_2), // outclk2.clk
          .locked   ()          // (terminated)
     );

endmodule
```

```
module control_interface(
        CLK,
        RESET_N,
        CMD,
        ADDR,
        REF_ACK,
          INIT_ACK,
        CM_ACK,
        NOP,
        READA,
        WRITEA,
        REFRESH,
        PRECHARGE,
        LOAD_MODE,
        SADDR,
        REF_REQ,
          INIT_REQ,
        CMD_ACK
        );

`include        "Sdram_Params.h"

input                       CLK;                    // System
Clock
input                       RESET_N;                // System
Reset
input   [2:0]               CMD;                    // Command
input
input   [`ASIZE-1:0]        ADDR;                   // Address
input                       REF_ACK;                // Refresh
request acknowledge
```

```
input                                   INIT_ACK;               //
Initial request acknowledge
input                              CM_ACK;                 // Command
acknowledge
output                             NOP;                    // Decoded
NOP command
output                             READA;                  // Decoded
READA command
output                             WRITEA;                 // Decoded
WRITEA command
output                             REFRESH;                // Decoded
REFRESH command
output                             PRECHARGE;              // Decoded
PRECHARGE command
output                             LOAD_MODE;              // Decoded
LOAD_MODE command
output  [`ASIZE-1:0]               SADDR;                  //
Registered version of ADDR
output                             REF_REQ;                // Hidden
refresh request
output                             INIT_REQ;               // Hidden
initial request
output                             CMD_ACK;                // Command
acknowledge



reg                                NOP;
reg                                READA;
reg                                WRITEA;
reg                                 REFRESH;
reg                                 PRECHARGE;
reg                                 LOAD_MODE;
reg        [`ASIZE-1:0]             SADDR;
reg                                 REF_REQ;
reg                                 INIT_REQ;
reg                                 CMD_ACK;

// Internal signals
reg        [15:0]                   timer;
reg         [15:0]                        init_timer;



// Command decode and ADDR register
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
        begin
                NOP                 <= 0;
                READA               <= 0;
                WRITEA              <= 0;
                SADDR               <= 0;
        end

        else
        begin

                SADDR <= ADDR;                                            //
register the address to keep proper
```

```
                                                                        //
alignment with the command

                if (CMD == 3'b000)                                      //
NOP command
                        NOP <= 1;
                else
                        NOP <= 0;

                if (CMD == 3'b001)                                      //
READA command
                        READA <= 1;
                else
                        READA <= 0;

                if (CMD == 3'b010)                                      //
WRITEA command
                        WRITEA <= 1;
                else
                        WRITEA <= 0;

        end
end


//  Generate CMD_ACK
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
                CMD_ACK <= 0;
        else
                if ((CM_ACK == 1) & (CMD_ACK == 0))
                        CMD_ACK <= 1;
                else
                        CMD_ACK <= 0;
end


// refresh timer
always @(posedge CLK or negedge RESET_N) begin
        if (RESET_N == 0)
        begin
                timer           <= 0;
                REF_REQ         <= 0;
        end
        else
        begin
                if (REF_ACK == 1)
                   begin
                   timer <= REF_PER;
                       REF_REQ   <=0;
                   end
                   else if (INIT_REQ == 1)
                   begin
                   timer <= REF_PER+200;
                       REF_REQ   <=0;
                   end
                else
                   timer <= timer - 1'b1;
                if (timer==0)
                    REF_REQ    <= 1;

        end
end

// initial timer
always @(posedge CLK or negedge RESET_N) begin
        if (RESET_N == 0)
        begin
                init_timer      <= 0;
                   REFRESH         <= 0;
                PRECHARGE        <= 0;
                   LOAD_MODE     <= 0;
                   INIT_REQ      <= 0;
        end
        else
        begin
                if (init_timer < (INIT_PER+201))
                        init_timer    <= init_timer+1;

                   if (init_timer < INIT_PER)
                   begin
                        REFRESH       <=0;
                        PRECHARGE <=0;
                        LOAD_MODE <=0;
                        INIT_REQ  <=1;
                   end
                   else if(init_timer == (INIT_PER+20))
                   begin
```

```
                        REFRESH      <=0;
                        PRECHARGE <=1;
                        LOAD_MODE <=0;
                        INIT_REQ  <=0;
                    end
                    else if(  (init_timer == (INIT_PER+40))    ||
                                   (init_timer == (INIT_PER+60))
     ||
                                   (init_timer == (INIT_PER+80))
     ||
                                   (init_timer == (INIT_PER+100))
     ||
                                   (init_timer == (INIT_PER+120))
     ||
                                   (init_timer == (INIT_PER+140))
     ||
                                   (init_timer == (INIT_PER+160))
     ||
                                   (init_timer == (INIT_PER+180))   )
                    begin
                        REFRESH      <=1;
                        PRECHARGE <=0;
                        LOAD_MODE <=0;
                        INIT_REQ  <=0;
                    end
                    else if(init_timer == (INIT_PER+200))
                    begin
                        REFRESH      <=0;
                        PRECHARGE <=0;
                        LOAD_MODE <=1;
                        INIT_REQ  <=0;
                    end
                    else
                    begin
                        REFRESH       <=0;
                        PRECHARGE <=0;
                        LOAD_MODE <=0;
                        INIT_REQ  <=0;

                    end
        end
end

endmodule
```

```
module command(
        CLK,
        RESET_N,
        SADDR,
        NOP,
        READA,
        WRITEA,
        REFRESH,
        PRECHARGE,
        LOAD_MODE,
        REF_REQ,
          INIT_REQ,
          PM_STOP,
          PM_DONE,
        REF_ACK,
        CM_ACK,
        OE,
        SA,
        BA,
        CS_N,
        CKE,
        RAS_N,
        CAS_N,
        WE_N
        );

`include        "Sdram_Params.h"

input                           CLK;                    // System
Clock
input                           RESET_N;                // System
Reset
input   [`ASIZE-1:0]            SADDR;                  // Address
input                           NOP;                    // Decoded
NOP command
input                           READA;                  // Decoded
READA command
input                           WRITEA;                 // Decoded
WRITEA command
input                           REFRESH;                // Decoded
REFRESH command
input                           PRECHARGE;              // Decoded
PRECHARGE command
input                           LOAD_MODE;              // Decoded
LOAD_MODE command
```

```
input                          REF_REQ;                  // Hidden
refresh request
input                               INIT_REQ;                 //
Hidden initial request
input                               PM_STOP;                  //
Page mode stop
input                               PM_DONE;                  //
Page mode done
output                         REF_ACK;                  // Refresh
request acknowledge
output                         CM_ACK;                   // Command
acknowledge
output                         OE;                       // OE
signal for data path module
output  [11:0]                 SA;                       // SDRAM
address
output  [1:0]                  BA;                       // SDRAM
bank address
output  [1:0]                  CS_N;                     // SDRAM
chip selects
output                         CKE;                      // SDRAM
clock enable
output                         RAS_N;                    // SDRAM
RAS
output                         CAS_N;                    // SDRAM
CAS
output                         WE_N;                     // SDRAM
WE_N
```

```
reg                                CM_ACK;
reg                                REF_ACK;
reg                                OE;
reg        [11:0]                  SA;
reg        [1:0]                   BA;
reg        [1:0]                   CS_N;
reg                                CKE;
reg                                RAS_N;
reg                                CAS_N;
reg                                WE_N;



// Internal signals
reg                                do_reada;
reg                                do_writea;
reg                                do_refresh;
reg                                do_precharge;
reg                                do_load_mode;
reg                                     do_initial;
reg                                command_done;
reg        [7:0]                   command_delay;
reg        [1:0]                   rw_shift;
reg                                do_act;
reg                                rw_flag;
reg                                do_rw;
reg        [6:0]                   oe_shift;
reg                                oe1;
reg                                oe2;
reg                                oe3;
reg                                 oe4;
reg      [3:0]                      rp_shift;
reg                                 rp_done;
reg                                      ex_read;
reg                                      ex_write;

wire     [`ROWSIZE - 1:0]           rowaddr;
wire     [`COLSIZE - 1:0]           coladdr;
wire     [`BANKSIZE - 1:0]          bankaddr;

assign   rowaddr   = SADDR[`ROWSTART + `ROWSIZE - 1: `ROWSTART];
// assignment of the row address bits from SADDR
assign   coladdr   = SADDR[`COLSTART + `COLSIZE - 1:`COLSTART];
// assignment of the column address bits
assign   bankaddr  = SADDR[`BANKSTART + `BANKSIZE - 1:`BANKSTART];
// assignment of the bank address bits



// This always block monitors the individual command lines and
issues a command
// to the next stage if there currently another command already
running.
//
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
        begin
                do_reada        <= 0;
                do_writea       <= 0;
                do_refresh      <= 0;
                do_precharge    <= 0;
                do_load_mode    <= 0;
                   do_initial           <= 0;
                command_done    <= 0;
                command_delay   <= 0;
                rw_flag         <= 0;
                rp_shift        <= 0;
                rp_done         <= 0;
                   ex_read              <= 0;
                   ex_write       <= 0;
        end

        else
        begin

//  Issue the appropriate command if the sdram is not currently
busy
              if( INIT_REQ == 1 )
              begin
                do_reada        <= 0;
                do_writea       <= 0;
                do_refresh      <= 0;
                do_precharge    <= 0;
                do_load_mode    <= 0;
                   do_initial           <= 1;
                command_done    <= 0;
                command_delay   <= 0;
                rw_flag         <= 0;
                rp_shift        <= 0;
                rp_done         <= 0;
                   ex_read            <= 0;
                   ex_write      <= 0;
              end
              else
              begin
                   do_initial         <= 0;

                if ((REF_REQ == 1 | REFRESH == 1) & command_done ==
0 & do_refresh == 0 & rp_done == 0          // Refresh
                        & do_reada == 0 & do_writea == 0)
                        do_refresh <= 1;
                else
                        do_refresh <= 0;

                if ((READA == 1) & (command_done == 0) & (do_reada
== 0) & (rp_done == 0) & (REF_REQ == 0))     // READA
                begin
                           do_reada <= 1;
                            ex_read <= 1;
                   end
                else
                        do_reada <= 0;

                if ((WRITEA == 1) & (command_done == 0) &
(do_writea == 0) & (rp_done == 0) & (REF_REQ == 0))  // WRITEA
                begin
                           do_writea <= 1;
                            ex_write <= 1;
                   end
                else
                        do_writea <= 0;

                if ((PRECHARGE == 1) & (command_done == 0) &
(do_precharge == 0))                                   // PRECHARGE
                        do_precharge <= 1;
                else
                        do_precharge <= 0;

                if ((LOAD_MODE == 1) & (command_done == 0) &
(do_load_mode == 0))                                   // LOADMODE
                        do_load_mode <= 1;
                else
                        do_load_mode <= 0;

// set command_delay shift register and command_done flag
// The command delay shift register is a timer that is used to
ensure that
// the SDRAM devices have had sufficient time to finish the last
command.

                if ((do_refresh == 1) | (do_reada == 1) |
(do_writea == 1) | (do_precharge == 1)
                            | (do_load_mode == 1))
                begin
                        command_delay <= 8'b11111111;
                        command_done  <= 1;
                        rw_flag <= do_reada;
                end
                else
                begin
                        command_done        <= command_delay[0];
// the command_delay shift operation
                        command_delay     <= (command_delay>>1);
                end


 // start additional timer that is used for the refresh, writea,
reada commands
                if (command_delay[0] == 0 & command_done == 1)
                begin
                   rp_shift <= 4'b1111;
                   rp_done <= 1;
                end
                else
                begin
                        if(SC_PM == 0)
                        begin
                            rp_shift  <= (rp_shift>>1);
                        rp_done       <= rp_shift[0];
                        end
                        else
                        begin
                            if( (ex_read == 0) && (ex_write == 0) )
                            begin
                                 rp_shift  <= (rp_shift>>1);
                        rp_done       <= rp_shift[0];
                            end
                            else
```

```
                             begin
                                  if( PM_STOP==1 )
                                  begin
                                       rp_shift  <= (rp_shift>>1);
                                  rp_done     <= rp_shift[0];
                                       ex_read        <= 1'b0;
                                       ex_write  <= 1'b0;
                                  end
                             end
                        end
                end
              end
        end
end


// logic that generates the OE signal for the data path module
// For normal burst write he duration of OE is dependent on the
configured burst length.
// For page mode accesses(SC_PM=1) the OE signal is turned on at
the start of the write command
// and is left on until a PRECHARGE(page burst terminate) is
detected.
//
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
        begin
                oe_shift <= 0;
                oe1      <= 0;
                oe2      <= 0;
                OE       <= 0;
        end
        else
        begin
                if (SC_PM == 0)
                begin
                        if (do_writea == 1)
                        begin
                                if (SC_BL == 1)
//  Set the shift register to the appropriate
                                        oe_shift <= 0;
// value based on burst length.
                                else if (SC_BL == 2)
                                        oe_shift <= 1;
                                else if (SC_BL == 4)
                                        oe_shift <= 7;
                                else if (SC_BL == 8)
                                        oe_shift <= 127;
                                oe1 <= 1;
                        end
                        else
                        begin
                                oe_shift <= (oe_shift>>1);
                                oe1  <= oe_shift[0];
                                oe2  <= oe1;
                                oe3  <= oe2;
                                oe4  <= oe3;
                                if (SC_RCD == 2)
                                        OE <= oe3;
                                else
                                        OE <= oe4;
                        end
                end
                else
                begin
                        if (do_writea == 1)
// OE generation for page mode accesses
                                oe4   <= 1;
                        else if (do_precharge == 1 | do_reada == 1
| do_refresh==1 | do_initial == 1 | PM_STOP==1 )
                                oe4   <= 0;
                        OE <= oe4;
                end

        end
end




// This always block tracks the time between the activate command
and the
// subsequent WRITEA or READA command, RC.  The shift register is
set using
// the configuration register setting SC_RCD. The shift register is
loaded with
// a single '1' with the position within the register dependent on
SC_RCD.
// When the '1' is shifted out of the register it sets so_rw which
```

```
triggers
// a writea or reada command
//
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
        begin
                rw_shift <= 0;
                do_rw    <= 0;
        end

        else
        begin

                if ((do_reada == 1) | (do_writea == 1))
                begin
                        if (SC_RCD == 1)
// Set the shift register
                                do_rw <= 1;
                        else if (SC_RCD == 2)
                                rw_shift <= 1;
                        else if (SC_RCD == 3)
                                rw_shift <= 2;
                end
                else
                begin
                        rw_shift <= (rw_shift>>1);
                        do_rw    <= rw_shift[0];
                end
        end
```

```
end

// This always block generates the command acknowledge, CM_ACK,
signal.
// It also generates the acknowledge signal, REF_ACK, that
acknowledges
// a refresh request that was generated by the internal refresh
timer circuit.
always @(posedge CLK or negedge RESET_N)
begin

        if (RESET_N == 0)
        begin
                CM_ACK   <= 0;
                REF_ACK  <= 0;
        end

        else
        begin
                if (do_refresh == 1 & REF_REQ == 1)
// Internal refresh timer refresh request
                        REF_ACK <= 1;
                else if ((do_refresh == 1) | (do_reada == 1) |
(do_writea == 1) | (do_precharge == 1)   // externa  commands
                        | (do_load_mode))
                        CM_ACK <= 1;
                else
                begin
                        REF_ACK <= 0;
                        CM_ACK  <= 0;
                end
        end
end









// This always block generates the address, cs, cke, and command
signals(ras,cas,wen)
//
always @(posedge CLK ) begin
        if (RESET_N==0) begin
                SA    <= 0;
                BA    <= 0;
                CS_N  <= 1;
                RAS_N <= 1;
                CAS_N <= 1;
                WE_N  <= 1;
                CKE   <= 0;
        end
        else begin
                CKE <= 1;

// Generate SA
                if (do_writea == 1 | do_reada == 1)    // ACTIVATE
command is being issued, so present the row address
                        SA <= rowaddr;
                else
                        SA <= coladdr;                   // else
alway present column address
                if ((do_rw==1) | (do_precharge))
                        SA[10] <= !SC_PM;                // set
SA[10] for autoprecharge read/write or for a precharge all command
                                                         // don't set
it if the controller is in page mode.
                if (do_precharge==1 | do_load_mode==1)
                        BA <= 0;                         // Set BA=0
if performing a precharge or load_mode command
                else
                        BA <= bankaddr[1:0];             // else set
it with the appropriate address bits

                if (do_refresh==1 | do_precharge==1 |
do_load_mode==1 | do_initial==1)
                        CS_N <= 0;
// Select both chip selects if performing
                else
// refresh, precharge(all) or load_mode
                begin
                        CS_N[0] <= SADDR[`ASIZE-1];
// else set the chip selects based off of the
                        CS_N[1] <= ~SADDR[`ASIZE-1];
// msb address bit
                end

                   if(do_load_mode==1)
                   SA     <= {2'b00,SDR_CL,SDR_BT,SDR_BL};
```

```
//Generate the appropriate logic levels on RAS_N, CAS_N, and WE_N
//depending on the issued command.
//
                if ( do_refresh==1 ) begin
// Refresh: S=00, RAS=0, CAS=0, WE=1
                        RAS_N <= 0;
                        CAS_N <= 0;
                        WE_N  <= 1;
                end
                else if ((do_precharge==1) & ((oe4 == 1) | (rw_flag
== 1))) begin       // burst terminate if write is active
                        RAS_N <= 1;
                        CAS_N <= 1;
                        WE_N  <= 0;
                end
                else if (do_precharge==1) begin                 //
Precharge All: S=00, RAS=0, CAS=1, WE=0
                        RAS_N <= 0;
                        CAS_N <= 1;
                        WE_N  <= 0;
                end
                else if (do_load_mode==1) begin                 //
Mode Write: S=00, RAS=0, CAS=0, WE=0
                        RAS_N <= 0;
                        CAS_N <= 0;
                        WE_N  <= 0;
                end
                else if (do_reada == 1 | do_writea == 1) begin  //
```

```
Activate: S=01 or 10, RAS=0, CAS=1, WE=1
                        RAS_N <= 0;
                        CAS_N <= 1;
                        WE_N  <= 1;
                end
                else if (do_rw == 1) begin                              //
Read/Write: S=01 or 10, RAS=1, CAS=0, WE=0 or 1
                        RAS_N <= 1;
                        CAS_N <= 0;
                        WE_N  <= rw_flag;
                end
                   else if (do_initial ==1) begin
                        RAS_N <= 1;
                        CAS_N <= 1;
                        WE_N  <= 1;
                   end
                else begin                                              //
No Operation: RAS=1, CAS=1, WE=1
                        RAS_N <= 1;
                        CAS_N <= 1;
                        WE_N  <= 1;
                end
        end
end

endmodule
```

```
module sdr_data_path(
        CLK,
        RESET_N,
        DATAIN,
        DM,
        DQOUT,
        DQM
        );

`include        "Sdram_Params.h"

input                           CLK;                    // System
Clock
input                           RESET_N;                // System
Reset
input   [`DSIZE-1:0]            DATAIN;                 // Data
input from the host
input   [`DSIZE/8-1:0]          DM;                     // byte
data masks
output  [`DSIZE-1:0]            DQOUT;
output  [`DSIZE/8-1:0]          DQM;                    // SDRAM
data mask ouputs
reg     [`DSIZE/8-1:0]          DQM;



// Allign the input and output data to the SDRAM control path
always @(posedge CLK or negedge RESET_N)
begin
        if (RESET_N == 0)
         DQM      <= `DSIZE/8-1'hF;
        else
```

```
         DQM      <=   DM;
end

assign DQOUT = DATAIN;

endmodule
```

```
`timescale 1 ps / 1 ps
// synopsys translate_on
module Sdram_WR_FIFO (
     aclr,
     data,
     rdclk,
     rdreq,
     wrclk,
     wrreq,
     q,
     rdempty,
     rdusedw,
     wrfull,
     wrusedw);

     input       aclr;
     input     [15:0]  data;
     input       rdclk;
     input       rdreq;
     input       wrclk;
     input       wrreq;
     output    [15:0]  q;
```

```
    output      rdempty;
    output    [8:0]  rdusedw;
    output      wrfull;
    output    [8:0]  wrusedw;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri0   aclr;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire  sub_wire0;
    wire [15:0] sub_wire1;
    wire  sub_wire2;
    wire [8:0] sub_wire3;
    wire [8:0] sub_wire4;
    wire  wrfull = sub_wire0;
    wire [15:0] q = sub_wire1[15:0];
    wire  rdempty = sub_wire2;
    wire [8:0] wrusedw = sub_wire3[8:0];
    wire [8:0] rdusedw = sub_wire4[8:0];

    dcfifo   dcfifo_component (
                   .rdclk (rdclk),
                   .wrclk (wrclk),
                   .wrreq (wrreq),
                   .aclr (aclr),
                   .data (data),
                   .rdreq (rdreq),
                   .wrfull (sub_wire0),
                   .q (sub_wire1),
                   .rdempty (sub_wire2),
                   .wrusedw (sub_wire3),
                   .rdusedw (sub_wire4),
                   .rdfull (),
                   .wrempty ());
     defparam
          dcfifo_component.intended_device_family = "Cyclone V",
          dcfifo_component.lpm_hint = "RAM_BLOCK_TYPE=M10K",
          dcfifo_component.lpm_numwords = 512,
          dcfifo_component.lpm_showahead = "OFF",
          dcfifo_component.lpm_type = "dcfifo",
          dcfifo_component.lpm_width = 16,
          dcfifo_component.lpm_widthu = 9,
          dcfifo_component.overflow_checking = "ON",
          dcfifo_component.rdsync_delaypipe = 4,
          dcfifo_component.read_aclr_synch = "OFF",
          dcfifo_component.underflow_checking = "ON",
          dcfifo_component.use_eab = "ON",
          dcfifo_component.write_aclr_synch = "OFF",
          dcfifo_component.wrsync_delaypipe = 4;


endmodule
```

```
`timescale 1 ps / 1 ps
// synopsys translate_on
module Sdram_RD_FIFO (
     aclr,
     data,
     rdclk,
     rdreq,
     wrclk,
     wrreq,
     q,
     rdempty,
     rdusedw,
     wrfull,
     wrusedw);

     input        aclr;
     input     [15:0]  data;
     input        rdclk;
     input        rdreq;
     input        wrclk;
     input        wrreq;
     output    [15:0]  q;
     output       rdempty;
     output    [8:0]  rdusedw;
     output       wrfull;
     output    [8:0]  wrusedw;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
     tri0   aclr;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif
```

```
    wire  sub_wire0;
    wire [15:0] sub_wire1;
    wire  sub_wire2;
    wire [8:0] sub_wire3;
    wire [8:0] sub_wire4;
    wire  wrfull = sub_wire0;
    wire [15:0] q = sub_wire1[15:0];
    wire  rdempty = sub_wire2;
    wire [8:0] wrusedw = sub_wire3[8:0];
    wire [8:0] rdusedw = sub_wire4[8:0];

    dcfifo   dcfifo_component (
                  .rdclk (rdclk),
                  .wrclk (wrclk),
                  .wrreq (wrreq),
                  .aclr (aclr),
                  .data (data),
                  .rdreq (rdreq),
                  .wrfull (sub_wire0),
                  .q (sub_wire1),
                  .rdempty (sub_wire2),
                  .wrusedw (sub_wire3),
                  .rdusedw (sub_wire4),
                  .rdfull (),
                  .wrempty ());
    defparam
         dcfifo_component.intended_device_family = "Cyclone V",
         dcfifo_component.lpm_hint = "RAM_BLOCK_TYPE=M10K",
         dcfifo_component.lpm_numwords = 512,
         dcfifo_component.lpm_showahead = "OFF",
          dcfifo_component.lpm_type = "dcfifo",
          dcfifo_component.lpm_width = 16,
          dcfifo_component.lpm_widthu = 9,
          dcfifo_component.overflow_checking = "ON",
          dcfifo_component.rdsync_delaypipe = 4,
          dcfifo_component.read_aclr_synch = "OFF",
          dcfifo_component.underflow_checking = "ON",
          dcfifo_component.use_eab = "ON",
          dcfifo_component.write_aclr_synch = "OFF",
          dcfifo_component.wrsync_delaypipe = 4;


endmodule
```

```
module YUV422_to_444    (    //    YUV 4:2:2 Input
                                  iYCbCr,
                                  //    YUV  4:4:4 Output
                                  oY,
                                  oCb,
                                  oCr,
                                  //    Control Signals
                                  iX,
                                  iCLK,
                                  iRST_N    );
//    YUV 4:2:2 Input
input     [15:0]    iYCbCr;
//    YUV  4:4:4 Output
output    [7:0]     oY;
output    [7:0]     oCb;
output    [7:0]     oCr;
```

```
//   Control Signals
input     [9:0]     iX;
input               iCLK;
input               iRST_N;
//   Internal Registers
reg       [7:0]     mY;
reg       [7:0]     mCb;
reg       [7:0]     mCr;

assign    oY   =    mY;
assign    oCb  =    mCb;
assign    oCr  =    mCr;

always@(posedge iCLK or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          mY   <=   0;
          mCb  <=   0;
          mCr  <=   0;
     end
     else
     begin
          if(iX[0])
          {mY,mCr}  <=   iYCbCr;
          else
          {mY,mCb}  <=   iYCbCr;
     end
end
```

```
endmodule
```

```
module YCbCr2RGB ( Red,Green,Blue,oDVAL,
                        iY,iCb,iCr,iDVAL,
                        iRESET,iCLK);
//   Input
input [7:0] iY,iCb,iCr;
input iDVAL,iRESET,iCLK;
wire iCLK;
//   Output
output [9:0] Red,Green,Blue;
output reg    oDVAL;
//   Internal Registers/Wires
reg [9:0] oRed,oGreen,oBlue;
reg  [3:0] oDVAL_d;
reg [19:0] X_OUT,Y_OUT,Z_OUT;
wire [26:0] X,Y,Z;

assign    Red  =    oRed;
assign    Green=    oGreen;
assign    Blue =    oBlue;

always@(posedge iCLK)
begin
     if(iRESET)
     begin
          oDVAL<=0;
          oDVAL_d<=0;
          oRed<=0;
          oGreen<=0;
```

```
            oBlue<=0;
        end
        else
        begin
            // Red
            if(X_OUT[19])
            oRed<=0;
            else if(X_OUT[18:0]>1023)
            oRed<=1023;
            else
            oRed<=X_OUT[9:0];
            // Green
            if(Y_OUT[19])
            oGreen<=0;
            else if(Y_OUT[18:0]>1023)
            oGreen<=1023;
            else
            oGreen<=Y_OUT[9:0];
            // Blue
            if(Z_OUT[19])
            oBlue<=0;
            else if(Z_OUT[18:0]>1023)
            oBlue<=1023;
            else
            oBlue<=Z_OUT[9:0];
            // Control
            {oDVAL,oDVAL_d}<={oDVAL_d,iDVAL};
        end
end
```

```
always@(posedge iCLK)
begin
     if(iRESET)
     begin
          X_OUT<=0;
          Y_OUT<=0;
          Z_OUT<=0;
     end
     else
     begin
          X_OUT<=( X - 114131 ) >>7;
          Y_OUT<=( Y + 69370  ) >>7;
          Z_OUT<=( Z - 141787 ) >>7;
     end
end

//   Y          596,     0,             817
MAC_3 u0(
               .aclr0(iRESET),
               .clock0(iCLK),
               .dataa_0(iY),
               .dataa_1(iCb),
               .dataa_2(iCr),
               .datab_0(17'h00254),
               .datab_1(17'h00000),
               .datab_2(17'h00331),
               .result(X)
               );
//MAC_3 u0(    iRESET,       iCLK,iY,           iCb,     iCr,
//             17'h00254,    17'h00000,    17'h00331,
//                      X);
//   Cb         596,     -200,         -416
MAC_3 u1(
               .aclr0(iRESET),
               .clock0(iCLK),
               .dataa_0(iY),
               .dataa_1(iCb),
               .dataa_2(iCr),
               .datab_0(17'h00254),
               .datab_1(17'h3FF38),
               .datab_2(17'h3FE60),
               .result(Y)
               );
//MAC_3 u1(   iRESET,       iCLK,iY,           iCb,      iCr,
//            17'h00254,    17'h3FF38,    17'h3FE60,
//            Y         );
//   Cr       596,     1033,         0
MAC_3 u2(
               .aclr0(iRESET),
               .clock0(iCLK),
               .dataa_0(iY),
               .dataa_1(iCb),
               .dataa_2(iCr),
               .datab_0(17'h00254),
               .datab_1(17'h00409),
               .datab_2(17'h00000),
               .result(Z)
               );
//MAC_3 u2(   iRESET,       iCLK,iY,           iCb,      iCr,
//            17'h00254,    17'h00409,    17'h00000,
//              Z         );

endmodule
```

```
module    VGA_Ctrl  (    //   Host Side
                              iRed,
                              iGreen,
                              iBlue,
                              oCurrent_X,
                              oCurrent_Y,
                              oAddress,
                              oRequest,
                              //   VGA Side
                              oVGA_R,
                              oVGA_G,
                              oVGA_B,
                              oVGA_HS,
                              oVGA_VS,
                              oVGA_SYNC,
                              oVGA_BLANK,
                              oVGA_CLOCK,
                              //   Control Signal
                              iCLK,
                              iRST_N,
                              iSW);
//   Host Side
input          [9:0]     iRed;
input          [9:0]     iGreen;
input          [9:0]     iBlue;
output         [21:0]    oAddress;
output          [10:0]    oCurrent_X;
output          [10:0]    oCurrent_Y;
output                    oRequest;
//   VGA Side
output          [9:0]     oVGA_R;
output          [9:0]     oVGA_G;
output          [9:0]     oVGA_B;
output    reg             oVGA_HS;
output    reg             oVGA_VS;
output                    oVGA_SYNC;
output                    oVGA_BLANK;
output                    oVGA_CLOCK;
reg             [10:0]    oColorX, oColorY;
//   Control Signal
input                     iCLK;
input                     iRST_N;
input             [9:0]            iSW;
//   Internal Registers
reg             [10:0]    H_Cont;
reg             [10:0]    V_Cont;
reg               [8:0]           pixel;
reg               [9:0]            tempR, tempG, tempB;
/////////////////////////////////////////////////////////////
//   Horizontal    Parameter
parameter H_FRONT   =    16;
parameter H_SYNC    =    96;
parameter H_BACK    =    48;
parameter H_ACT     =    640;
parameter H_BLANK   =    H_FRONT+H_SYNC+H_BACK;
parameter H_TOTAL   =    H_FRONT+H_SYNC+H_BACK+H_ACT;
```

```
/////////////////////////////////////////////////////////////
//   Vertical Parameter
parameter V_FRONT   =    11;
parameter V_SYNC    =    2;
parameter V_BACK    =    31;
parameter V_ACT     =    480;
parameter V_BLANK   =    V_FRONT+V_SYNC+V_BACK;
parameter V_TOTAL   =    V_FRONT+V_SYNC+V_BACK+V_ACT;
/////////////////////////////////////////////////////////////
assign    oVGA_SYNC =    1'b1;                //   This pin is unused.
assign    oVGA_BLANK     =    ~((H_Cont<H_BLANK)||(V_Cont<V_BLANK));
assign    oVGA_CLOCK     =    ~iCLK;
assign    oVGA_R         =    tempR;
assign    oVGA_G         =    tempG;
assign    oVGA_B         =    tempB;
assign    oAddress  =    oCurrent_Y*H_ACT+oCurrent_X;
assign    oRequest  =    ((H_Cont>=H_BLANK && H_Cont<H_TOTAL)  &&
                              (V_Cont>=V_BLANK && V_Cont<V_TOTAL));
assign    oCurrent_X     =    (H_Cont>=H_BLANK)  ?    H_Cont-H_BLANK:
     11'h0     ;
assign    oCurrent_Y     =    (V_Cont>=V_BLANK)  ?    V_Cont-V_BLANK:
     11'h0     ;

reg [2:0] R [640*480];
reg [2:0] G [640*480];
reg [2:0] B [640*480];
reg [$clog2(640*480*2):0] counter;

wire done;
```

```
assign done = !iRST_N ? 0 : counter == 640*480*2;

always @(posedge iCLK) begin
  if (!oCurrent_Y && !done) begin
    counter <= 0;
  end else if(counter < 640*480*2) begin
    R[oCurrent_Y*640+oCurrent_X] <= iRed[9:7];
    G[oCurrent_Y*640+oCurrent_X] <= iGreen[9:7];
    B[oCurrent_Y*640+oCurrent_X] <= iBlue[9:7];
  end

  if ((counter < 640*480*2) && !done) begin
      counter <= counter + 1;
  end

  pixel <= {
             R[oCurrent_Y*640+oCurrent_X],
          G[oCurrent_Y*640+oCurrent_X],
          B[oCurrent_Y*640+oCurrent_X]
           };
end

always @ (*)  begin
  if (iSW[0] && iRed >= 600 && iRed <= 1023 && iBlue <= 360 &&
iBlue >= 0 && iGreen <= 640) begin // Cloak Mask
    tempR = {pixel[8:6], 7'b1111100};
    tempG = {pixel[5:3], 7'b1111100};
    tempB = {pixel[2:0], 7'b1111100};
  end else if(iSW[1]) begin // Video OFF
    tempR = 10'b0;
    tempG = 10'b0;
    tempB = 10'b0;
  end else if(iSW[2]) begin // Captured Foreground Image
    tempR = {pixel[8:6], 7'b1111100};
    tempG = {pixel[5:3], 7'b1111100};
    tempB = {pixel[2:0], 7'b1111100};
  end else if(iSW[3]) begin // Red Filter
    tempR = iRed;
    tempG = 10'b0;
    tempB = 10'b0;
  end else if(iSW[4]) begin // Green Filter
    tempR = 10'b0;
    tempG = iGreen;
    tempB = 10'b0;
  end else if(iSW[5]) begin // Blue Filter
    tempR = 10'b0;
    tempG = 10'b0;
    tempB = iBlue;
 end else if(iSW[6]) begin // Grayscale Filter
    tempR = (iRed != 0) ? (299 * iRed / 1000) + (587 * iGreen /
1000) + (114 * iBlue / 1000) : 0;
    tempG = (iGreen != 0) ? (299 * iRed / 1000) + (587 * iGreen /
1000) + (114 * iBlue / 1000) : 0;
    tempB = (iBlue != 0) ? (299 * iRed / 1000) + (587 * iGreen /
1000) + (114 * iBlue / 1000) : 0;
 end else if(iSW[7]) begin // Invert Video
    tempR = 1023 - ((iRed + iGreen + iBlue) / 3);
    tempG = 1023 - ((iRed + iGreen + iBlue) / 3);
    tempB = 1023 - ((iRed + iGreen + iBlue) / 3);
 end else if(iSW[8]) begin // Dark Video
```

```
    tempR = (iRed > 200) ? iRed - 200 : 0;
    tempG = (iGreen > 200) ? iGreen - 200 : 0;
    tempB = (iBlue > 200) ? iBlue - 200 : 0;
 end else if(iSW[9]) begin // Bright Video
    tempR = (iRed + 200 > 1023) ? 1023 : iRed + 200;
    tempG = (iGreen + 200 > 1023) ? 1023 : iGreen + 200;
    tempB = (iBlue + 200 > 1023) ? 1023 : iBlue + 200;
 end else begin // Real-time Video
    tempR = iRed;
    tempG = iGreen;
    tempB = iBlue;
  end
end

always@(posedge iCLK or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          H_Cont         <=   0;
          oVGA_HS   <=   1;
     end
     else
     begin
          if(H_Cont<H_TOTAL)
          H_Cont    <=   H_Cont+1'b1;
          else
          H_Cont    <=   0;
          //   Horizontal Sync
          if(H_Cont==H_FRONT-1)            //   Front porch end
          oVGA_HS   <=   1'b0;
          if(H_Cont==H_FRONT+H_SYNC-1)//   Sync pulse end
          oVGA_HS   <=   1'b1;
     end
end

//   Vertical Generator: Refer to the horizontal sync
always@(posedge oVGA_HS or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          V_Cont         <=   0;
          oVGA_VS   <=   1;
     end
     else
     begin
          if(V_Cont<V_TOTAL)
          V_Cont    <=   V_Cont+1'b1;
          else
          V_Cont    <=   0;
          //   Vertical Sync
          if(V_Cont==V_FRONT-1)       //   Front porch end
          oVGA_VS   <=   1'b0;
          if(V_Cont==V_FRONT+V_SYNC-1)//   Sync pulse end
          oVGA_VS   <=   1'b1;
     end
end

endmodule
```

```verilog
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module Line_Buffer (
     aclr,
     clken,
     clock,
     shiftin,
     shiftout,
     taps);

     input       aclr;
     input       clken;
     input       clock;
     input     [15:0]  shiftin;
     output    [15:0]  shiftout;
     output    [15:0]  taps;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
     tri1   aclr;
     tri1   clken;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

     wire [15:0] sub_wire0;
     wire [15:0] sub_wire1;
     wire [15:0] shiftout = sub_wire0[15:0];
     wire [15:0] taps = sub_wire1[15:0];
     altshift_taps ALTSHIFT_TAPS_component (
                    .aclr (aclr),
                    .clock (clock),
                    .clken (clken),
                    .shiftin (shiftin),
                    .shiftout (sub_wire0),
                    .taps (sub_wire1));
     defparam
          ALTSHIFT_TAPS_component.intended_device_family = "Cyclone
V",
          ALTSHIFT_TAPS_component.lpm_hint = "RAM_BLOCK_TYPE=M10K",
          ALTSHIFT_TAPS_component.lpm_type = "altshift_taps",
          ALTSHIFT_TAPS_component.number_of_taps = 1,
          ALTSHIFT_TAPS_component.tap_distance = 640,
          ALTSHIFT_TAPS_component.width = 16;


endmodule
```

```
module AUDIO_DAC ( //   Memory Side
                        oFLASH_ADDR,iFLASH_DATA,
                        oSDRAM_ADDR,iSDRAM_DATA,
                        oSRAM_ADDR,iSRAM_DATA,
                        //   Audio Side
                        oAUD_BCK,
                        oAUD_DATA,
                        oAUD_LRCK,
                        //   Control Signals
                        iSrc_Select,
                        iCLK_18_4,
                         iRST_N    );

parameter REF_CLK          =    18562000; //   18.432   MHz
parameter SAMPLE_RATE      =    48000;         //   48        KHz
parameter DATA_WIDTH       =    16;            //   16        Bits
parameter CHANNEL_NUM      =    2;             //   Dual Channel

parameter SIN_SAMPLE_DATA  =    48;
parameter FLASH_DATA_NUM=    1048576;  //   1    MWords
parameter SDRAM_DATA_NUM=    4194304;  //   4    MWords
parameter SRAM_DATA_NUM =    262144;        //   256  KWords

parameter FLASH_ADDR_WIDTH=  20;            //   20   Address Line
parameter SDRAM_ADDR_WIDTH=  22;            //   22   Address Line
parameter SRAM_ADDR_WIDTH=   18;            //   18   Address   Line

parameter FLASH_DATA_WIDTH=  8;             //   8    Bits
parameter SDRAM_DATA_WIDTH=  16;            //   16   Bits
parameter SRAM_DATA_WIDTH=   16;            //   16   Bits

////////////  Input Source Number    //////////////
parameter SIN_SANPLE       =    0;
parameter FLASH_DATA       =    1;
parameter SDRAM_DATA       =    2;
parameter SRAM_DATA     =    3;
///////////////////////////////////////////////////
//   Memory Side
output    [FLASH_ADDR_WIDTH-1:0]  oFLASH_ADDR;
input     [FLASH_DATA_WIDTH-1:0]  iFLASH_DATA;
```

```
output    [SDRAM_ADDR_WIDTH:0]    oSDRAM_ADDR;
input     [SDRAM_DATA_WIDTH-1:0]  iSDRAM_DATA;
output    [SRAM_ADDR_WIDTH:0]         oSRAM_ADDR;
input     [SRAM_DATA_WIDTH-1:0]   iSRAM_DATA;
//   Audio Side
output              oAUD_DATA;
output              oAUD_LRCK;
output    reg       oAUD_BCK;
//   Control Signals
input     [1:0]     iSrc_Select;
input               iCLK_18_4;
input               iRST_N;
//   Internal Registers and Wires
reg       [3:0]     BCK_DIV;
reg       [8:0]     LRCK_1X_DIV;
reg       [7:0]     LRCK_2X_DIV;
reg       [6:0]     LRCK_4X_DIV;
reg       [3:0]     SEL_Cont;
////////  DATA Counter  ////////
reg       [5:0]     SIN_Cont;
reg       [FLASH_ADDR_WIDTH-1:0]  FLASH_Cont;
reg       [SDRAM_ADDR_WIDTH-1:0]  SDRAM_Cont;
reg       [SRAM_ADDR_WIDTH-1:0]   SRAM_Cont;
////////////////////////////////////
reg       [DATA_WIDTH-1:0]   Sin_Out;
reg       [DATA_WIDTH-1:0]   FLASH_Out;
reg       [DATA_WIDTH-1:0]   SDRAM_Out;
reg       [DATA_WIDTH-1:0]   SRAM_Out;
reg       [DATA_WIDTH-1:0]   FLASH_Out_Tmp;
reg       [DATA_WIDTH-1:0]   SDRAM_Out_Tmp;
reg         [DATA_WIDTH-1:0]   SRAM_Out_Tmp;
reg                                 LRCK_1X;
reg                                 LRCK_2X;
reg                                 LRCK_4X;

////////////  AUD_BCK Generator  //////////////
always@(posedge iCLK_18_4 or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          BCK_DIV       <=   0;
          oAUD_BCK  <=   0;
     end
     else
     begin
          if(BCK_DIV >=
REF_CLK/(SAMPLE_RATE*DATA_WIDTH*CHANNEL_NUM*2)-1 )
          begin
               BCK_DIV       <=   0;
               oAUD_BCK  <=   ~oAUD_BCK;
          end
          else
          BCK_DIV       <=   BCK_DIV+1;
     end
end
///////////////////////////////////////////////////
////////////  AUD_LRCK Generator //////////////
always@(posedge iCLK_18_4 or negedge iRST_N)
begin
     if(!iRST_N)
```

```
    begin
        LRCK_1X_DIV    <=   0;
        LRCK_2X_DIV    <=   0;
        LRCK_4X_DIV    <=   0;
        LRCK_1X        <=   0;
        LRCK_2X        <=   0;
        LRCK_4X        <=   0;
    end
    else
    begin
        //   LRCK 1X
        if(LRCK_1X_DIV >= REF_CLK/(SAMPLE_RATE*2)-1 )
        begin
            LRCK_1X_DIV    <=   0;
            LRCK_1X   <=   ~LRCK_1X;
        end
        else
        LRCK_1X_DIV         <=   LRCK_1X_DIV+1;
        //   LRCK 2X
        if(LRCK_2X_DIV >= REF_CLK/(SAMPLE_RATE*4)-1 )
        begin
            LRCK_2X_DIV    <=   0;
            LRCK_2X   <=   ~LRCK_2X;
        end
        else
        LRCK_2X_DIV         <=   LRCK_2X_DIV+1;
        //   LRCK 4X
        if(LRCK_4X_DIV >= REF_CLK/(SAMPLE_RATE*8)-1 )
        begin
            LRCK_4X_DIV    <=   0;
```

```
               LRCK_4X   <=   ~LRCK_4X;
          end
          else
          LRCK_4X_DIV        <=   LRCK_4X_DIV+1;
     end
end
assign    oAUD_LRCK =    LRCK_1X;
///////////////////////////////////////////////////
//////////    Sin LUT ADDR Generator  //////////////
always@(negedge LRCK_1X or negedge iRST_N)
begin
     if(!iRST_N)
     SIN_Cont  <=   0;
     else
     begin
          if(SIN_Cont < SIN_SAMPLE_DATA-1 )
          SIN_Cont  <=   SIN_Cont+1;
          else
          SIN_Cont  <=   0;
     end
end
///////////////////////////////////////////////////
//////////    FLASH ADDR Generator    //////////////
always@(negedge LRCK_4X or negedge iRST_N)
begin
     if(!iRST_N)
     FLASH_Cont     <=   0;
     else
     begin
          if(FLASH_Cont < FLASH_DATA_NUM-1 )
          FLASH_Cont    <=   FLASH_Cont+1;
          else
          FLASH_Cont    <=   0;
     end
end
assign    oFLASH_ADDR   =    FLASH_Cont;
///////////////////////////////////////////////////
//////////      FLASH DATA Reorder    //////////////
always@(posedge LRCK_4X or negedge iRST_N)
begin
     if(!iRST_N)
     FLASH_Out_Tmp <=   0;
     else
     begin
          if(FLASH_Cont[0])
          FLASH_Out_Tmp[15:8]     <=   iFLASH_DATA;
          else
          FLASH_Out_Tmp[7:0] <=   iFLASH_DATA;
     end
end
always@(negedge LRCK_2X or negedge iRST_N)
begin
     if(!iRST_N)
     FLASH_Out <=   0;
     else
     FLASH_Out <=   FLASH_Out_Tmp;
end
///////////////////////////////////////////////////
//////////    SDRAM ADDR Generator    //////////////
always@(negedge LRCK_2X or negedge iRST_N)
```

```
begin
	if(!iRST_N)
	SDRAM_Cont	<=	0;
	else
	begin
		if(SDRAM_Cont < SDRAM_DATA_NUM-1 )
		SDRAM_Cont	<=	SDRAM_Cont+1;
		else
		SDRAM_Cont	<=	0;
	end
end
assign	oSDRAM_ADDR	=	SDRAM_Cont;
////////////////////////////////////////////////////
//////////	SDRAM DATA Latch	//////////////
always@(posedge LRCK_2X or negedge iRST_N)
begin
	if(!iRST_N)
	SDRAM_Out_Tmp <=	0;
	else
	SDRAM_Out_Tmp <=	iSDRAM_DATA;
end
always@(negedge LRCK_2X or negedge iRST_N)
begin
	if(!iRST_N)
	SDRAM_Out <=	0;
	else
	SDRAM_Out <=	SDRAM_Out_Tmp;
end
////////////////////////////////////////////////////
////////////	SRAM ADDR Generator	////////////
```

```verilog
always@(negedge LRCK_2X or negedge iRST_N)
begin
    if(!iRST_N)
    SRAM_Cont <=   0;
    else
    begin
        if(SRAM_Cont < SRAM_DATA_NUM-1 )
        SRAM_Cont <=   SRAM_Cont+1;
        else
        SRAM_Cont <=   0;
    end
end
assign   oSRAM_ADDR    =    SRAM_Cont;
///////////////////////////////////////////////////
//////////      SRAM DATA Latch      //////////////
always@(posedge LRCK_2X or negedge iRST_N)
begin
    if(!iRST_N)
    SRAM_Out_Tmp  <=   0;
    else
    SRAM_Out_Tmp  <=   iSRAM_DATA;
end
always@(negedge LRCK_2X or negedge iRST_N)
begin
    if(!iRST_N)
    SRAM_Out  <=   0;
    else
    SRAM_Out  <=   SRAM_Out_Tmp;
end
///////////////////////////////////////////////////
```

```
//////////    16 Bits PISO MSB First  //////////////
always@(negedge oAUD_BCK or negedge iRST_N)
begin
     if(!iRST_N)
     SEL_Cont  <=   0;
     else
     SEL_Cont  <=   SEL_Cont+1;
end
assign    oAUD_DATA =    (iSrc_Select==SIN_SANPLE)    ?
     Sin_Out[~SEL_Cont] :
                              (iSrc_Select==FLASH_DATA)    ?
     FLASH_Out[~SEL_Cont]:
                              (iSrc_Select==SDRAM_DATA)    ?
     SDRAM_Out[~SEL_Cont]:

     SRAM_Out[~SEL_Cont]     ;

////////////////////////////////////////////////////
////////////  Sin Wave ROM Table //////////////
always@(SIN_Cont)
begin
    case(SIN_Cont)
    0  :  Sin_Out      <=      0       ;
    1  :  Sin_Out      <=      4276    ;
    2  :  Sin_Out      <=      8480    ;
    3  :  Sin_Out      <=      12539   ;
    4  :  Sin_Out      <=      16383   ;
    5  :  Sin_Out      <=      19947   ;
    6  :  Sin_Out      <=      23169   ;
    7  :  Sin_Out      <=      25995   ;
```

```
8   :  Sin_Out       <=      28377   ;
9   :  Sin_Out       <=      30272   ;
10   :  Sin_Out      <=      31650   ;
11   :  Sin_Out      <=      32486   ;
12   :  Sin_Out      <=      32767   ;
13   :  Sin_Out      <=      32486   ;
14   :  Sin_Out      <=      31650   ;
15   :  Sin_Out      <=      30272   ;
16   :  Sin_Out      <=      28377   ;
17   :  Sin_Out      <=      25995   ;
18   :  Sin_Out      <=      23169   ;
19   :  Sin_Out      <=      19947   ;
20   :  Sin_Out      <=      16383   ;
21   :  Sin_Out      <=      12539   ;
22   :  Sin_Out      <=      8480    ;
23   :  Sin_Out      <=      4276    ;
24   :  Sin_Out      <=      0       ;
25   :  Sin_Out      <=      61259   ;
26   :  Sin_Out      <=      57056   ;
27   :  Sin_Out      <=      52997   ;
28   :  Sin_Out      <=      49153   ;
29   :  Sin_Out      <=      45589   ;
30   :  Sin_Out      <=      42366   ;
31   :  Sin_Out      <=      39540   ;
32   :  Sin_Out      <=      37159   ;
33   :  Sin_Out      <=      35263   ;
34   :  Sin_Out      <=      33885   ;
35   :  Sin_Out      <=      33049   ;
36   :  Sin_Out      <=      32768   ;
37   :  Sin_Out      <=      33049   ;
```

```
    38  :  Sin_Out      <=      33885   ;
    39  :  Sin_Out      <=      35263   ;
    40  :  Sin_Out      <=      37159   ;
    41  :  Sin_Out      <=      39540   ;
    42  :  Sin_Out      <=      42366   ;
    43  :  Sin_Out      <=      45589   ;
    44  :  Sin_Out      <=      49152   ;
    45  :  Sin_Out      <=      52997   ;
    46  :  Sin_Out      <=      57056   ;
    47  :  Sin_Out      <=      61259   ;
     default   :
               Sin_Out         <=        0         ;
     endcase
end
//////////////////////////////////////////////////////

endmodule
```

```
module I2C_AV_Config (  //    Host Side
                              iCLK,
                              iRST_N,
                              //    I2C Side
                              I2C_SCLK,
                              I2C_SDAT  );
//    Host Side
input          iCLK;
input          iRST_N;
//    I2C Side
output         I2C_SCLK;
```

```
inout           I2C_SDAT;
//   Internal Registers/Wires
reg  [15:0]     mI2C_CLK_DIV;
reg  [23:0]     mI2C_DATA;
reg             mI2C_CTRL_CLK;
reg             mI2C_GO;
wire      mI2C_END;
wire      mI2C_ACK;
reg  [15:0]     LUT_DATA;
reg  [5:0]      LUT_INDEX;
reg  [3:0]      mSetup_ST;

//   Clock Setting
parameter CLK_Freq  =    50000000; //   50   MHz
parameter I2C_Freq  =    20000;         //   20   KHz
//   LUT Data Number
parameter LUT_SIZE  =    51;
//   Audio Data Index
parameter Dummy_DATA     =    0;
parameter SET_LIN_L =    1;
parameter SET_LIN_R =    2;
parameter SET_HEAD_L     =    3;
parameter SET_HEAD_R     =    4;
parameter A_PATH_CTRL    =    5;
parameter D_PATH_CTRL    =    6;
parameter POWER_ON  =    7;
parameter SET_FORMAT     =    8;
parameter SAMPLE_CTRL    =    9;
parameter SET_ACTIVE     =    10;
//   Video Data Index
```

```
parameter SET_VIDEO =    11;

/////////////////////   I2C Control Clock  ////////////////////////
always@(posedge iCLK or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          mI2C_CTRL_CLK <=   0;
          mI2C_CLK_DIV  <=   0;
     end
     else
     begin
          if( mI2C_CLK_DIV   < (CLK_Freq/I2C_Freq) )
          mI2C_CLK_DIV  <=   mI2C_CLK_DIV+1;
          else
          begin
               mI2C_CLK_DIV  <=   0;
               mI2C_CTRL_CLK <=   ~mI2C_CTRL_CLK;
          end
     end
end
////////////////////////////////////////////////////////////////////
/
I2C_Controller     u0   (    .CLOCK(mI2C_CTRL_CLK),      //
     Controller Work Clock
                             .I2C_SCLK(I2C_SCLK),        //   I2C
CLOCK
                             .I2C_SDAT(I2C_SDAT),        //   I2C
DATA
                             .I2C_DATA(mI2C_DATA),       //
     DATA:[SLAVE_ADDR,SUB_ADDR,DATA]
                              .GO(mI2C_GO),                    //
     GO transfor
                              .END(mI2C_END),                  //
     END transfor
                              .ACK(mI2C_ACK),                  //
     ACK
                              .RESET(iRST_N));
////////////////////////////////////////////////////////////////////
/
//////////////////////  Config Control////////////////////////////
always@(posedge mI2C_CTRL_CLK or negedge iRST_N)
begin
     if(!iRST_N)
     begin
          LUT_INDEX <=   0;
          mSetup_ST <=   0;
          mI2C_GO        <=   0;
     end
     else
     begin
          if(LUT_INDEX<LUT_SIZE)
          begin
               case(mSetup_ST)
               0:   begin
                         if(LUT_INDEX<SET_VIDEO)
                         mI2C_DATA <=   {8'h34,LUT_DATA};
                         else
                         mI2C_DATA <=   {8'h40,LUT_DATA};
                         mI2C_GO        <=   1;
```

```
                    mSetup_ST <=   1;
                end
            1:  begin
                    if(mI2C_END)
                    begin
                        if(!mI2C_ACK)
                        mSetup_ST <=   2;
                        else
                        mSetup_ST <=   0;

                        mI2C_GO      <=   0;
                    end
                end
            2:  begin
                    LUT_INDEX <=   LUT_INDEX+1;
                    mSetup_ST <=   0;
                end
            endcase
        end
    end
end
/////////////////////////////////////////////////////////
/
/////////////////////   Config Data LUT
//////////////////////////
always
begin
    case(LUT_INDEX)
    //   Audio Config Data
    SET_LIN_L :    LUT_DATA  <=   16'h001A;
```

```
     SET_LIN_R :    LUT_DATA  <=   16'h021A;
     SET_HEAD_L     :    LUT_DATA <=   16'h047B;
     SET_HEAD_R     :    LUT_DATA <=   16'h067B;
     A_PATH_CTRL    :    LUT_DATA <=   16'h08F8;
     D_PATH_CTRL    :    LUT_DATA <=   16'h0A06;
     POWER_ON  :    LUT_DATA  <=   16'h0C00;
     SET_FORMAT     :    LUT_DATA <=   16'h0E01;
     SAMPLE_CTRL    :    LUT_DATA <=   16'h1002;
     SET_ACTIVE     :    LUT_DATA <=   16'h1201;
     //   Video Config Data
     SET_VIDEO+1    :    LUT_DATA     <=     16'h0000;//04
     SET_VIDEO+2    :    LUT_DATA     <=     16'hc301;
    SET_VIDEO+3     :    LUT_DATA     <=     16'hc480;
    SET_VIDEO+4     :    LUT_DATA     <=     16'h0457;
     SET_VIDEO+5    :    LUT_DATA     <=     16'h1741;
     SET_VIDEO+6    :    LUT_DATA     <=     16'h5801;
     SET_VIDEO+7    :    LUT_DATA     <=     16'h3da2;
     SET_VIDEO+8    :    LUT_DATA     <=     16'h37a0;
     SET_VIDEO+9    :    LUT_DATA     <=     16'h3e6a;
     SET_VIDEO+10   :    LUT_DATA     <=     16'h3fa0;
     SET_VIDEO+11   :    LUT_DATA     <=     16'h0e80;
     SET_VIDEO+12   :    LUT_DATA     <=     16'h5581;
     SET_VIDEO+13   :    LUT_DATA     <=     16'h37A0;    // Polarity
regiser
     SET_VIDEO+14   :    LUT_DATA     <=     16'h0880; // Contrast
Register
     SET_VIDEO+15   :    LUT_DATA     <=     16'h0a18; // Brightness
Register
     SET_VIDEO+16   :    LUT_DATA     <=     16'h2c8e; // AGC Mode
control
```

```
     SET_VIDEO+17  :    LUT_DATA     <=     16'h2df8;   // Chroma
Gain Control 1
     SET_VIDEO+18  :    LUT_DATA     <=     16'h2ece; // Chroma Gain
Control 2
     SET_VIDEO+19  :    LUT_DATA     <=     16'h2ff4; // Luma Gain
Control 1
     SET_VIDEO+20  :    LUT_DATA     <=     16'h30b2; // Luma Gain
Control 2
     SET_VIDEO+21  :    LUT_DATA     <=     16'h0e00;

     default:      LUT_DATA  <=   16'd0 ;
     endcase
end
////////////////////////////////////////////////////////////////////
/
endmodule
```

13 Reference

1. Invisible Cloak using OpenCV | Python Project, https://www.geeksforgeeks.org/invisible-cloak-using-opencv-python-project/
2. DE-1 SoC Manual and Datasheet: https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1004282204-de1-soc-user-manual.pdf
3. Video capture using DE1-SoC: https://hackaday.io/project/19945-video-capture-using-de1-soc-hps
4. ADV 7180 Video Decoder Datasheet - ADV7180 (Rev. J)
5. I2C Protocol Working - https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/
6. Display VGA Video Raster Scan Algorithm - http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/video.pdf
7. Altera Memory System Design - https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/edh_ed51008.pdf
8. FPGA NTSC Video Feed and Processing - https://inst.eecs.berkeley.edu/~cs150/Documents/VideoNutshell.pdf
9. Altera Avalon Bus Specification sheet - https://www.ee.ryerson.ca/~courses/coe718/Data-Sheets/sopc/mnl_avalon_bus.pdf
10. UToronto Tutorials http://www-ug.eecg.toronto.edu/msl/manuals/tutorial_DE1-SoC.v5.1.pdf