차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

Input/Output: Image Sensor, LCD Drive

Xuan-Truong Nguyen

2022.1.5 (Thu)

Road map

Review

Image sensor

Sensor, display model

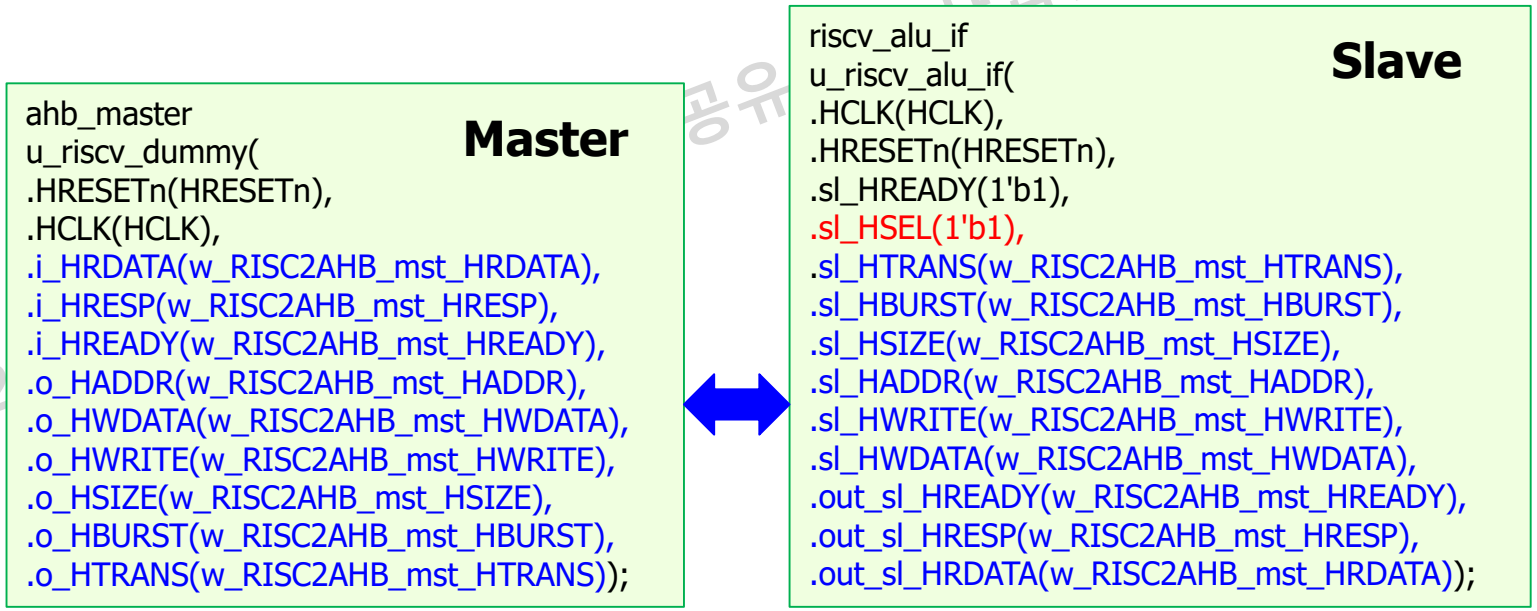
Brightness adjustment

Reversible Color Transform (RCT)

A target system

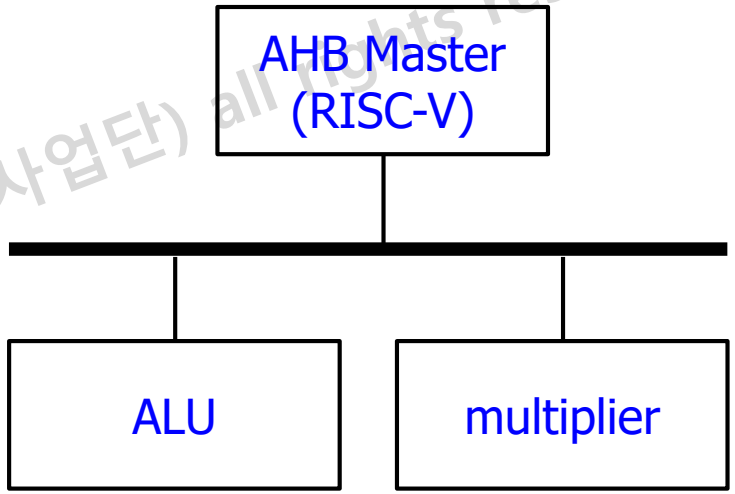
- A simple system has **one AHB master (RISV-V dummy) and one AHB slave (ALU)**
  - AHB ports of the master and slave are connected directly.
  - The slave is always selected by the master, i.e., sl_HSEL = 1'b1.

⇒ In practice, a system usually consists of multiple slaves



A target system

- Build a target system that consists of an AHB master and two AHB slaves
  - Master: RISC-V dummy
  - Slaves: ALU and multiplier
- Two questions:
  - How to make your own AHB slave IP, i.e. multiplier?
  - How to add your own IP into a system?



Map (map.v)

- ALU and Multiplier have different base addresses

```
//------------------------------------------------------------
// Base Address
//------------------------------------------------------------
`define RISCV_ALU_BASE_ADDR         32'hE000_0000
`define RISCV_MULTIPLIER_BASE_ADDR  32'hE000_1000
`define RISCV_BASE_ADDRESS_MASK     32'hFFFF_F000
//------------------------------------------------------------
// ALU
//------------------------------------------------------------
`define RISCV_REG_ALU_OP_I   (`RISCV_ALU_BASE_ADDR + 32'h00)
`define RISCV_REG_ALU_A_I    (`RISCV_ALU_BASE_ADDR + 32'h04)
`define RISCV_REG_ALU_B_I    (`RISCV_ALU_BASE_ADDR + 32'h08)
`define RISCV_REG_ALU_P_O    (`RISCV_ALU_BASE_ADDR + 32'h0C)

//------------------------------------------------------------
// Multipler
//------------------------------------------------------------
`define RISCV_REG_MUL_OP_I       (`RISCV_MULTIPLIER_BASE_ADDR + 32'h00)
`define RISCV_REG_MUL_A_I        (`RISCV_MULTIPLIER_BASE_ADDR + 32'h04)
`define RISCV_REG_MUL_B_I        (`RISCV_MULTIPLIER_BASE_ADDR + 32'h08)
`define RISCV_REG_MUL_A_SIGNED   (`RISCV_MULTIPLIER_BASE_ADDR + 32'h0C)
`define RISCV_REG_MUL_B_SIGNED   (`RISCV_MULTIPLIER_BASE_ADDR + 32'h10)
`define RISCV_REG_MUL_P_O_LOW    (`RISCV_MULTIPLIER_BASE_ADDR + 32'h14)
`define RISCV_REG_MUL_P_O_HIGH   (`RISCV_MULTIPLIER_BASE_ADDR + 32'h18)
`define RISCV_REG_MUL_STALL_W    (`RISCV_MULTIPLIER_BASE_ADDR + 32'h1C)
```

AHB decoder

- Observation: Read/Write commands from a master are valid
  ⇒ Select signals are not necessary
- RISC-V writes "alu_a_i" to a register of ALU defined by an address:
  - Base address: RISCV_ALU_BASE_ADDR
  - Offset: 0x04

```
//------------------------------------------------------------
// Base Address
//------------------------------------------------------------
`define RISCV_ALU_BASE_ADDR          32'hE000_0000
`define RISCV_MULTIPLIER_BASE_ADDR   32'hE000_1000
`define RISCV_BASE_ADDRESS_MASK      32'hFFFF_F000
//------------------------------------------------------------
// ALU
//------------------------------------------------------------
`define RISCV_REG_ALU_OP_I   (`RISCV_ALU_BASE_ADDR + 32'h00)
`define RISCV_REG_ALU_A_I    (`RISCV_ALU_BASE_ADDR + 32'h04)
`define RISCV_REG_ALU_B_I    (`RISCV_ALU_BASE_ADDR + 32'h08)
`define RISCV_REG_ALU_P_O    (`RISCV_ALU_BASE_ADDR + 32'h0C)
```

```
#(8*p)
    //sl_HSEL_alu = 1'b1;
    //sl_HSEL_multiplier = 1'b0;
#(8*p)
        alu_a_i = 32'h0;
        alu_b_i = 32'h0;
        alu_op_i = `ALU_SLT;
#(4*p) u_top_system.u_riscv_dummy.task_AHBwrite(`RISCV_REG_ALU_A_I, alu_a_i);
#(4*p) u_top_system.u_riscv_dummy.task_AHBwrite(`RISCV_REG_ALU_B_I, alu_b_i);
#(4*p) u_top_system.u_riscv_dummy.task_AHBwrite(`RISCV_REG_ALU_OP_I, alu_op_i);
#(4*p) u_top_system.u_riscv_dummy.task_AHBread (`RISCV_REG_ALU_P_O, alu_p_o);
```

AHB Decoder/Interconnect (top_system.v)

- To generate a select signal for an AHB slave, it decodes an access address
  - ALU : 32'hE0000_0xxx (xxx= don't care).
  - Multiplier : 32'hE0000_1xxx (xxx= don't care).

⇒ Select an AHB slave IP by its BASE ADDRESS

```
//------------------------------------------------
// Base Address
//------------------------------------------------
`define RISCV_ALU_BASE_ADDR         32'hE000_0000
`define RISCV_MULTIPLIER_BASE_ADDR  32'hE000_1000
`define RISCV_BASE_ADDRESS_MASK     32'hFFFF_F000
//------------------------------------------------------------------
// Select
//------------------------------------------------------------------
always@(*) begin
    sl_HSEL_alu = 1'b0;
    sl_HSEL_multiplier =1'b0;
    //Insert your code
    //{{{
    if((w_RISC2AHB_mst_HADDR & `RISCV_BASE_ADDRESS_MASK) == `RISCV_ALU_BASE_ADDR)
        sl_HSEL_alu = 1'b1;
    if((w_RISC2AHB_mst_HADDR & `RISCV_BASE_ADDRESS_MASK) == `RISCV_MULTIPLIER_BASE_ADDR)
        sl_HSEL_multiplier = 1'b1;
    //}}}
end
```

Address decoder (riscv_alu_if.v)

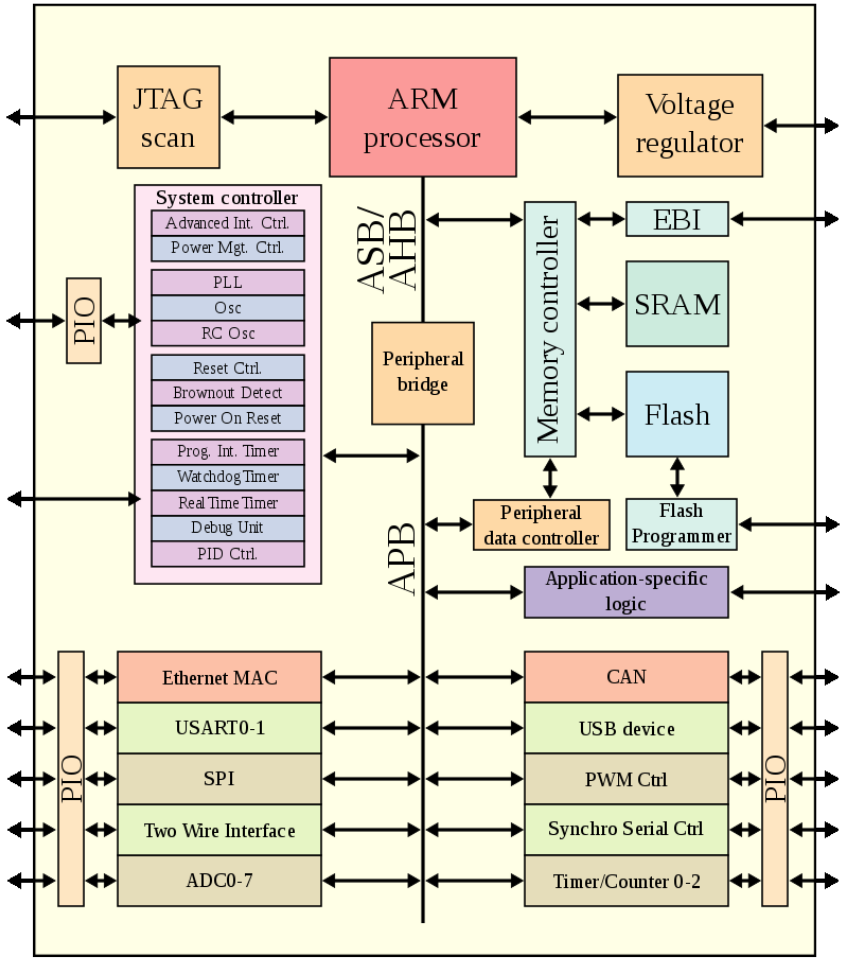
- At each AHB slave IP, use an OFFSET to access a register
  - Offset is 4 bytes, so it ignores W_WB_DATA LSB bits
  - Access all registers, so it needs W_REGS bits.
    - Example: W_REGS = 2 to access 4 registers at ALU

```
//------------------------------------------
// Decode Stage: Address Phase
//------------------------------------------
always @(posedge HCLK or negedge HRESETn)
begin
    if(~HRESETn)
    begin
        //control
        q_sel_sl_reg <= 0;
        q_ld_sl_reg <= 1'b0;
    end
    else begin
        if(sl_HSEL && sl_HREADY && ((sl_HTRANS == `TRANS_NONSEQ) || (sl_HTRANS == `TRANS_SEQ)))
        begin
            q_sel_sl_reg <= sl_HADDR[W_REGS+W_WB_DATA-1:W_WB_DATA];
            q_ld_sl_reg <= sl_HWRITE;
        end
        else begin
            q_ld_sl_reg <= 1'b0;
        end
    end
end
```

```
//------------------------------------------------------------
// Base Address
//------------------------------------------------------------
`define RISCV_ALU_BASE_ADDR         32'hE000_0000
`define RISCV_MULTIPLIER_BASE_ADDR  32'hE000_1000
`define RISCV_BASE_ADDRESS_MASK     32'hFFFF_F000
//------------------------------------------------------------
// ALU
//------------------------------------------------------------
`define RISCV_REG_ALU_OP_I  (`RISCV_ALU_BASE_ADDR + 32'h00)
`define RISCV_REG_ALU_A_I   (`RISCV_ALU_BASE_ADDR + 32'h04)
`define RISCV_REG_ALU_B_I   (`RISCV_ALU_BASE_ADDR + 32'h08)
`define RISCV_REG_ALU_P_O   (`RISCV_ALU_BASE_ADDR + 32'h0C)
```

A System-on-a-Chip (SoC) for ARM

- A microcontroller-based system on a chip
- Functional components
  - Processor cores
  - Memory
  - Interfaces: for communications
    - industry standards such as USB, FireWire, Ethernet, USART, SPI, HDMI, I²C, etc
  - Digital signal processors (DSPs)
    - Perform signal processing operations in SoCs for sensors, actuators, data collection, data analysis and multimedia processing

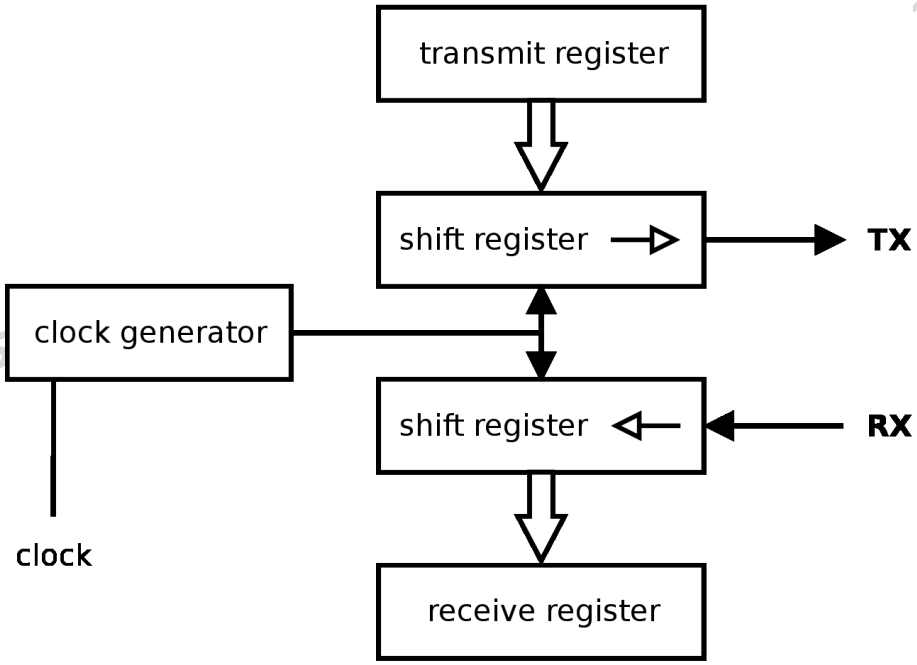


https://en.wikipedia.org/wiki/System_on_a_chip

Input/output

- UART: Universal asynchronous receiver-transmitter
  - a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable
  - A clock generator
  - Input and output shift registers
  - Transmit/receive control
  - Read/write control logic

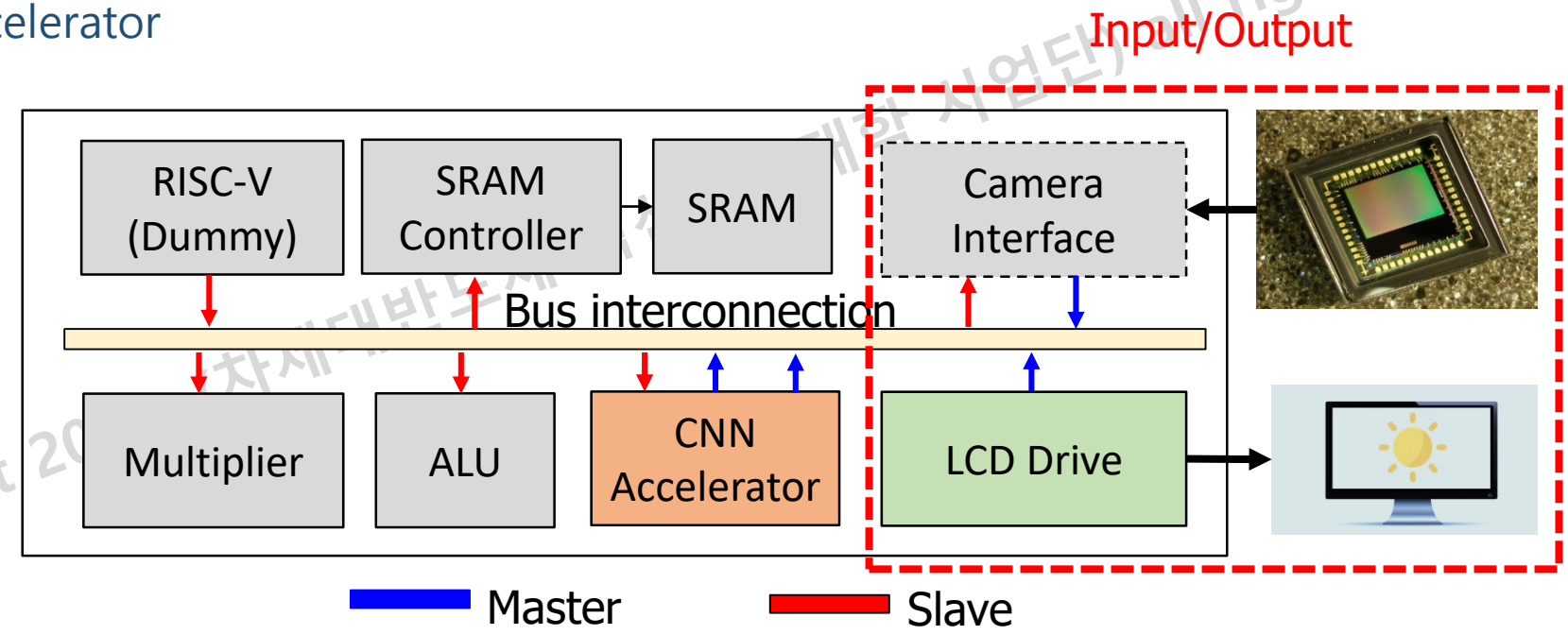
- USB: Universal Serial Bus
- SPI: Serial Peripheral Interface
- I2C: Inter-Integrated Circuit
- JTAG: Joint Test Action Group



UART block diagram

Our target system block diagram

- System with a bus architecture:
  - Master (RISC-V), Slave Memory: SRAM
  - Custom IPs: Multiplier, ALU (single transfer)
  - Input: Camera Interface → Assume that an image is stored in memory
  - Output: LCD Drive
  - CNN accelerator



Lecture plan

- Today you will
  - Understand an image format
  - Understand the digital signals for an image sensor and an LCD drive
  - Model an image sensor and an LCD drive using Verilog HDL
    - Lab 1: Image sensor, display model
    - Lab 2: Brightness adjustment
    - Lab 3: Reversible Color Transform (RCT)
      - Frame memory compression

Road map

Review

Image sensor

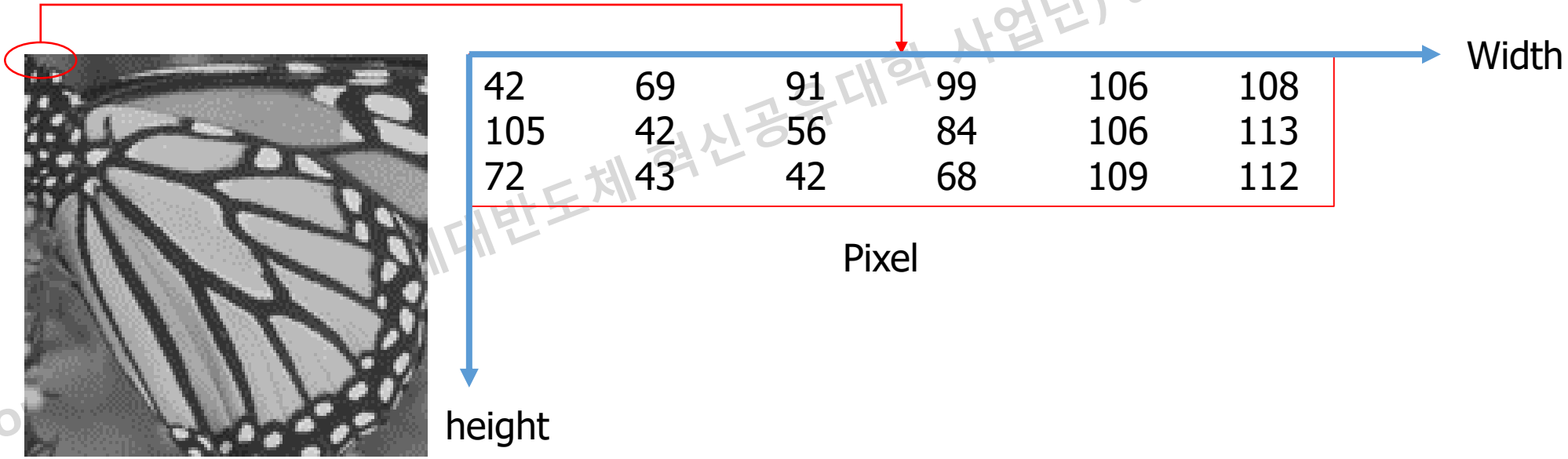
Sensor, display model

Brightness adjustment

Reversible Color Transform (RCT)

Input/output: Image

- What is an image?
  - An image is an artifact that depicts visual perception, such as a photograph or a 2D picture that resembles a subject—usually a physical object—and thus provides a depiction of it.
- In the context of signal processing, an image is a distributed amplitude of color(s)



What is an image?

- An image can be represented by using **three color channels** such as red, green and blue.
  - An RGB pixel: 2^{24} colors for a pixel in {(0, 0, 0), (0, 0, 1), …, (255, 255, 255)}
- Example
  - Matlab: A = **imread**(filename) = A[height][width][channel]
  - Python/C++ OpenCV: cv2.**imread**('banana.jpg', cv2.IMREAD_COLOR)

42	58	85	98	115	120	124	124
41	47	54	72	96	106	113	123
80	52	42	42	59	70	97	114
225	136	38	35	37	62	89	97
108	99	44	27	33	53	78	97
30	33	37	35	33	35	46	90
37	37	38	33	34	34	46	74
45	42	42	37	34	34	50	98

Red

30	43	66	76	89	92	96	97
23	29	38	56	77	83	86	95
62	32	25	35	45	53	74	88
208	118	23	27	25	48	69	72
90	84	34	25	26	45	63	74
17	22	29	33	27	27	28	66
29	33	36	34	32	29	32	53
39	38	38	35	29	29	37	80

Green

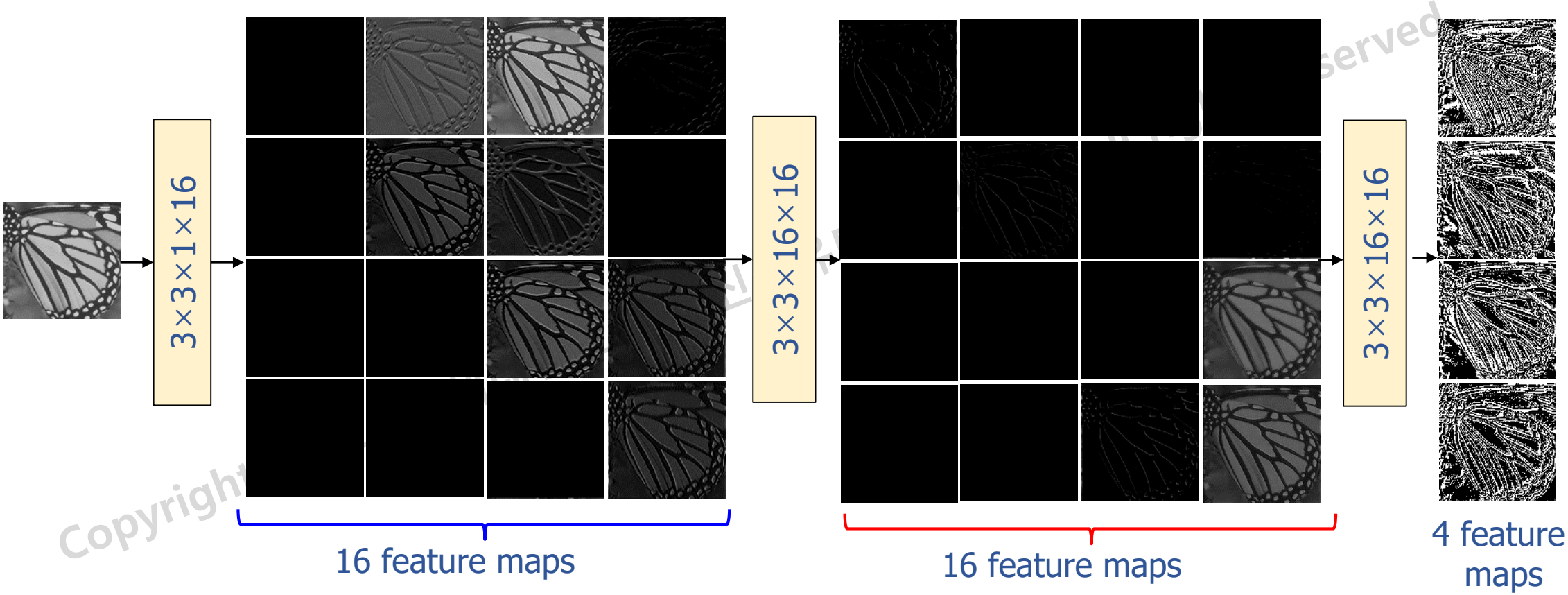
22	26	42	43	53	53	56	54
12	13	19	34	49	51	51	55
46	14	12	23	25	25	42	50
193	102	13	23	15	28	44	37
74	68	24	23	19	32	41	41
6	11	24	30	24	20	10	37
24	26	32	28	27	25	17	26
34	32	34	29	25	25	26	61

Blue

Image

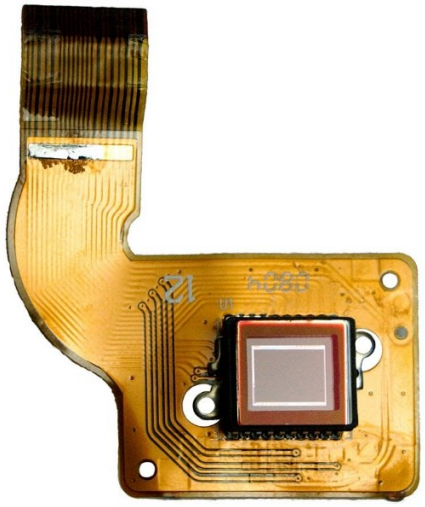
Feature images/maps in a CNN network

- Input and feature maps are also images when doing inference on a CNN network

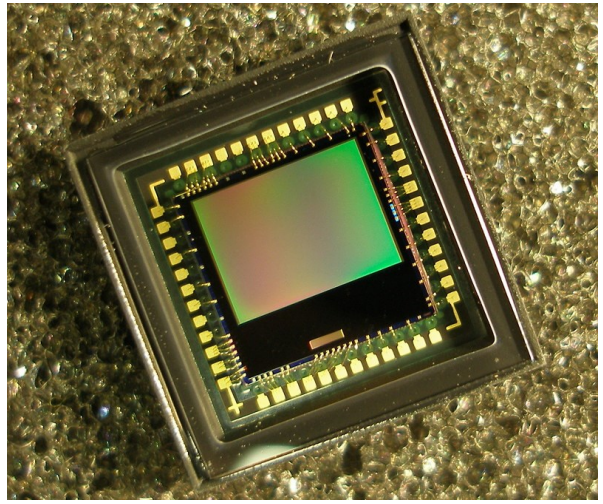


Input: CCD vs CMOS sensors

- An image sensor or imager is a sensor that detects and conveys information used to make an image (from Wikipedia).
  - Invented in 1969 at AT&T Bell Labs by Willard Boyle and George E. Smith.
  - In 2009, they were awarded the Nobel Prize for Physician
- Two main types of digital image sensors
  - Charge-coupled device (CCD).
  - Active-pixel sensor (CMOS sensor).



A CCD image sensor on a flexible circuit board

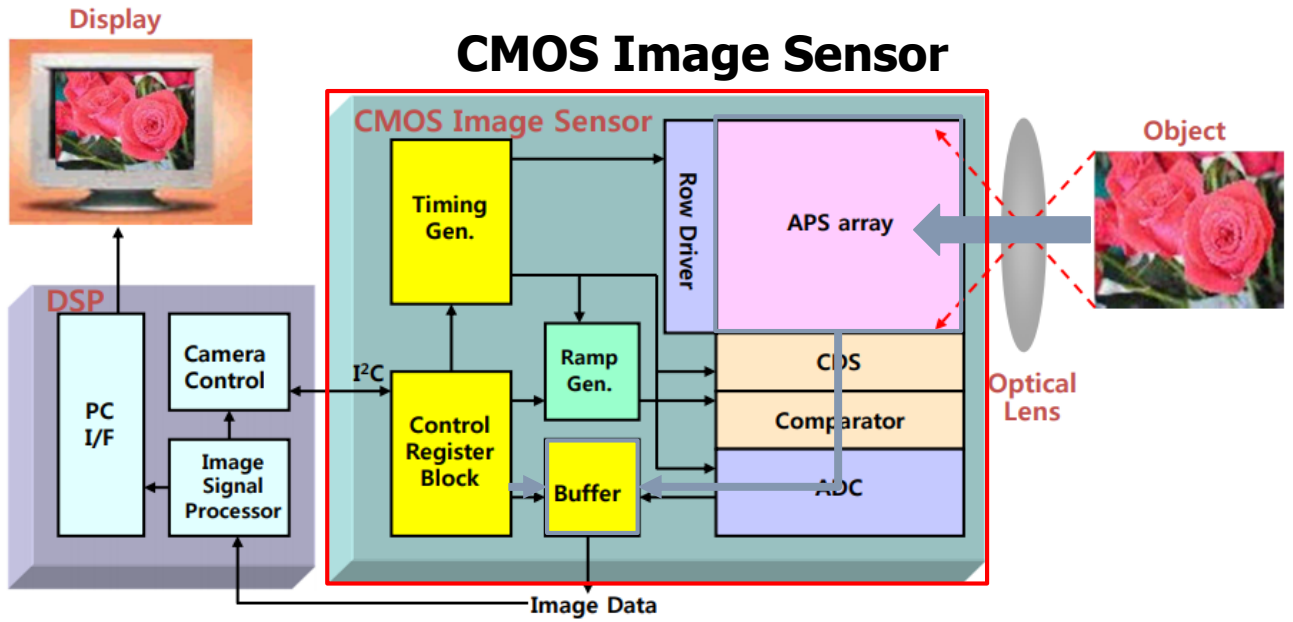


CMOS image sensor

https://www.teledynedalsa.com/en/learn/knowledge-center/ccd-vs-cmos/

CMOS Image Sensor (CIS)

- An object → Optical Lens → CMOS Image Sensor → Image Signal Processor → Display
- It has an active pixel sensor (APS) array that captures an image of an object
  - Each pixel sensor unit cell has a photodetector (i.e., photodiode).
  - A CMOS sensor typically captures a row at a time
- Image data is stored in a buffer



CIS: Function block Diagram

- Capture an image from a sensor
  - Video timing generator
  - Image Array
  - Analog signal processor
  - ADC
  - Digital signal processor
  - Video port
  - I2C/SCCB interface

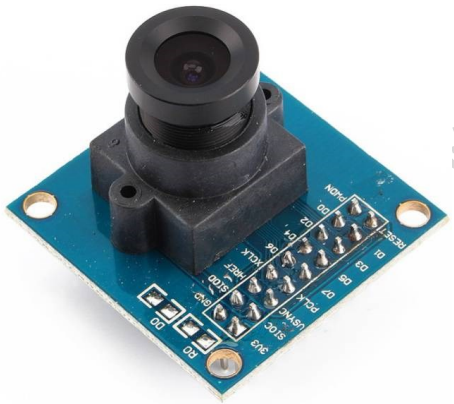


Image Sensor (OV7670)

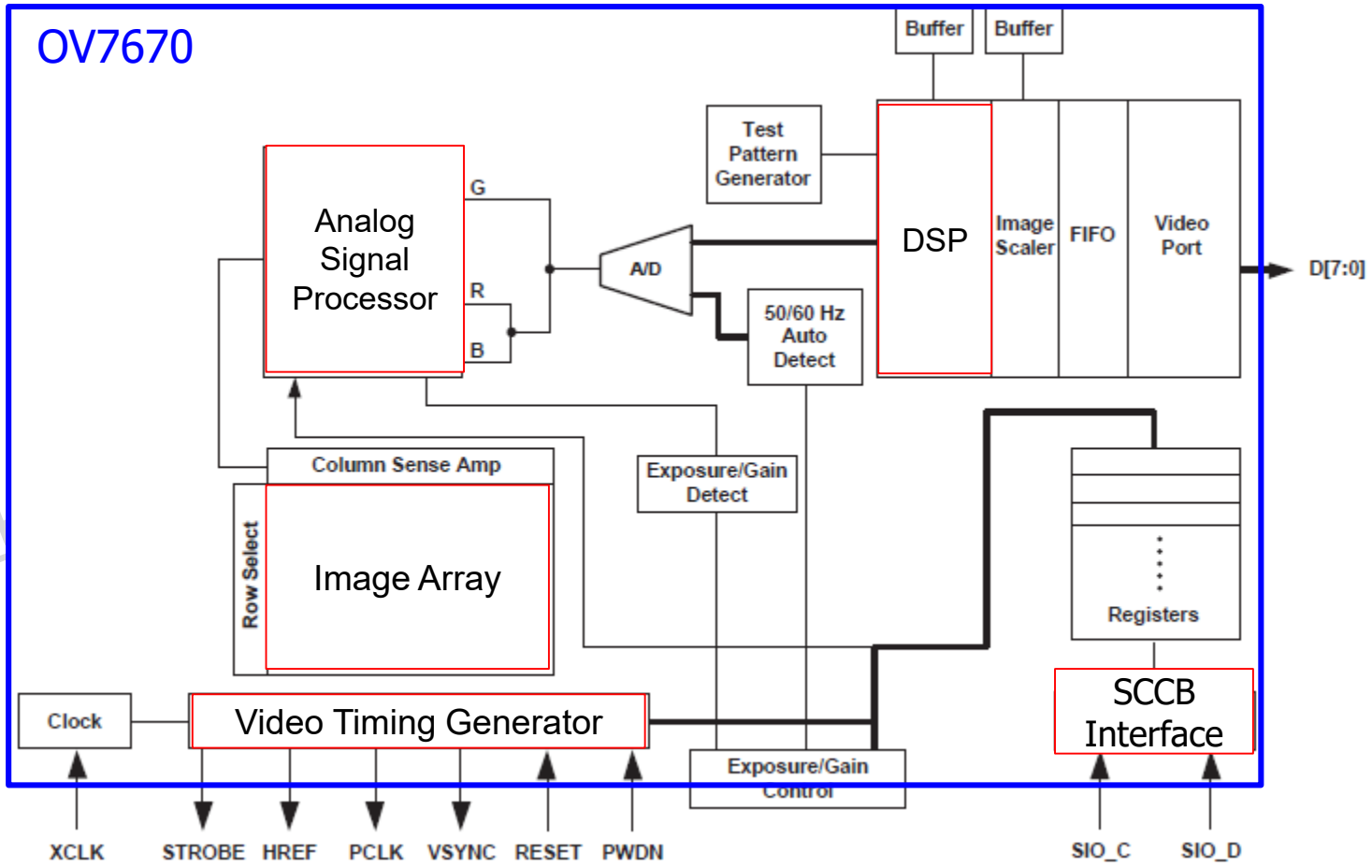


Image sensor timing

- Signals:
  - VSYNC: Frame synchronization
  - HREF: Line/row image synchronization
  - D[7:0] Camera data

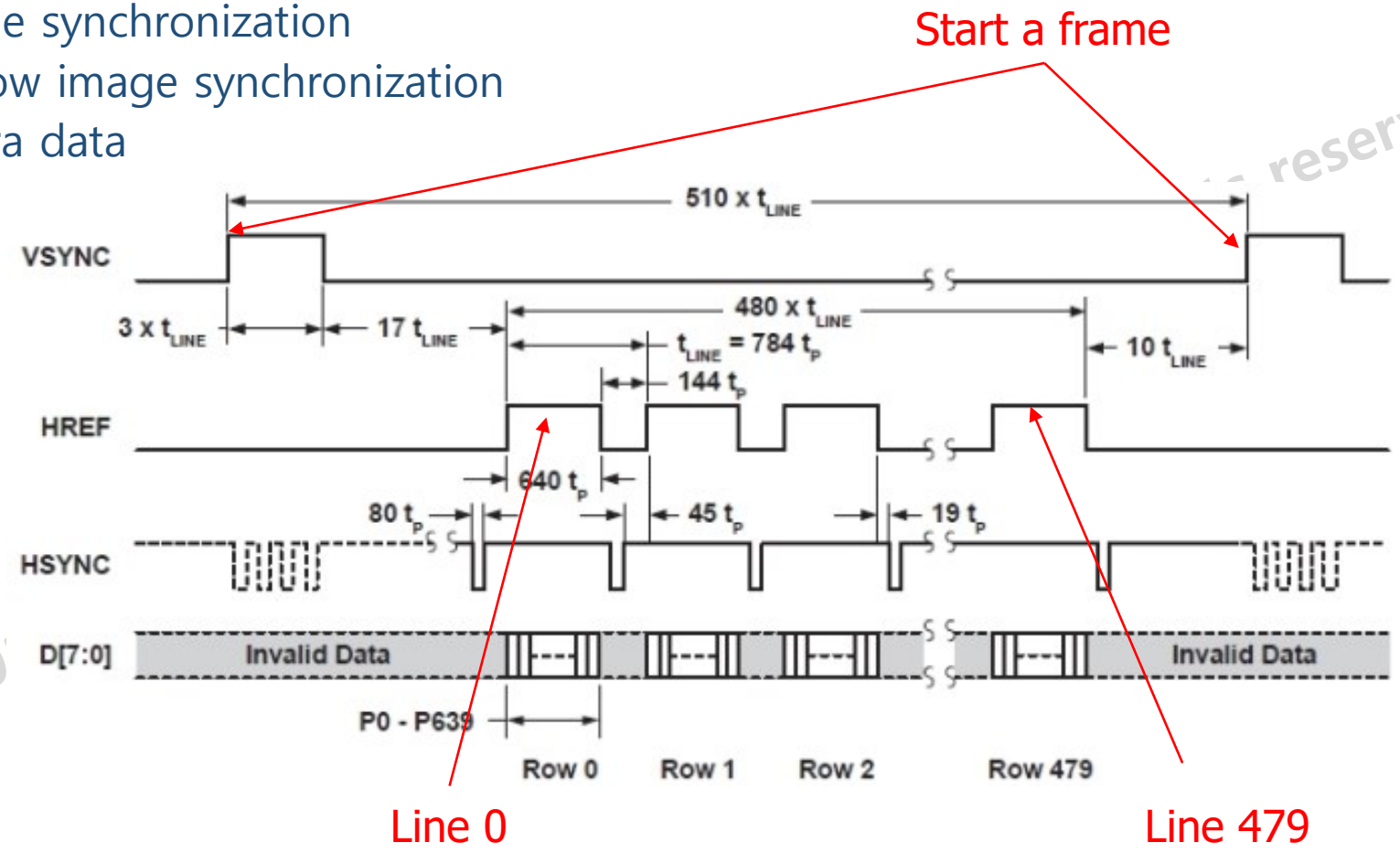


Image Sensor - Arduino

- Image sensor (OV7670): Capture an image from a sensor
- Arduino board: an embedded board
- Schematic to connect an Arduino board and an OV7670 Sensor
  - I2C interfaces: SIOC (SCL or clock), SDIO (SDA or data).
  - VSYNC, HREF, PLCK (camera clock output), XCLK (input clock).
  - Pixel data D[7:0]

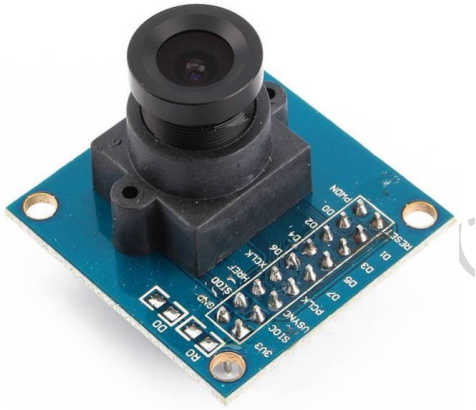
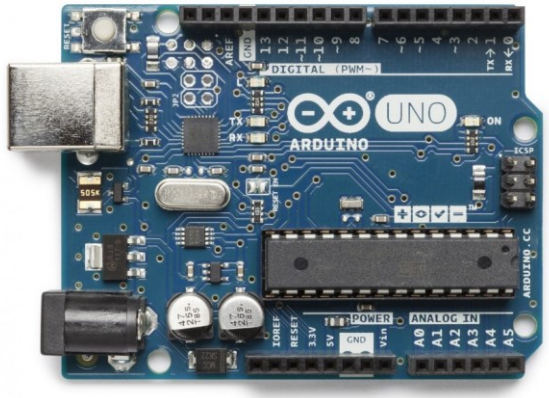
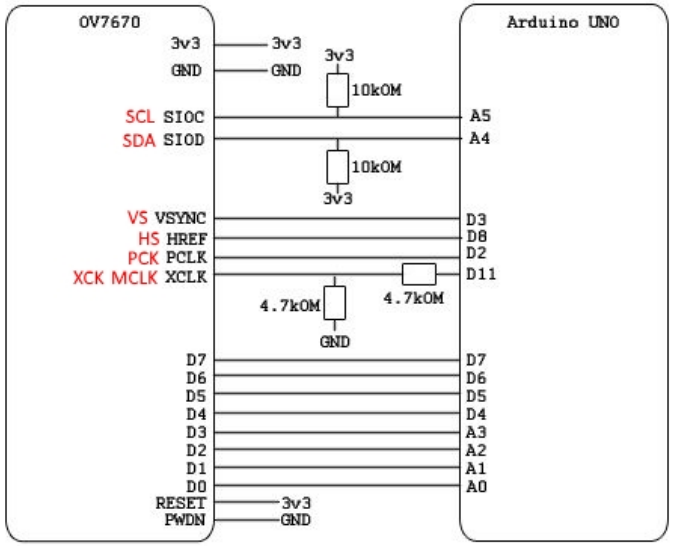


Image Sensor (OV7670)



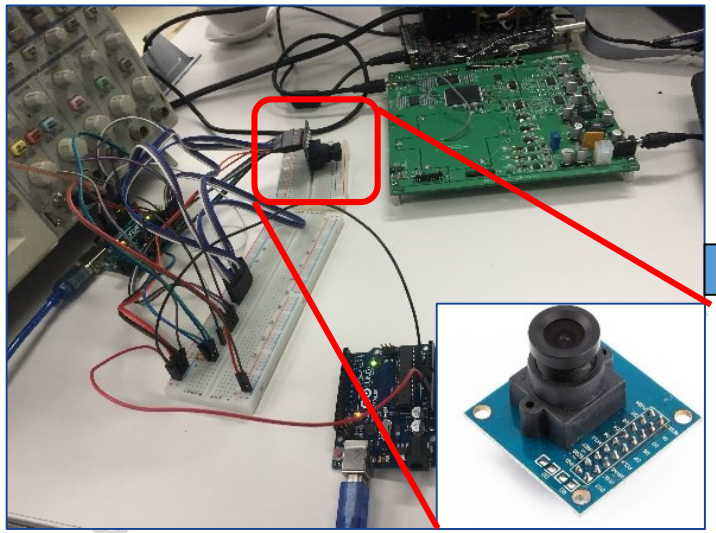
Arduino board



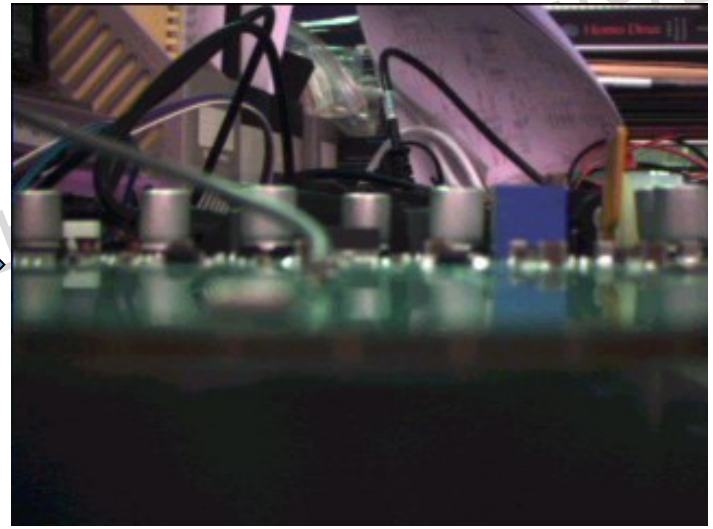
Schematic

Image Sensor - Arduino

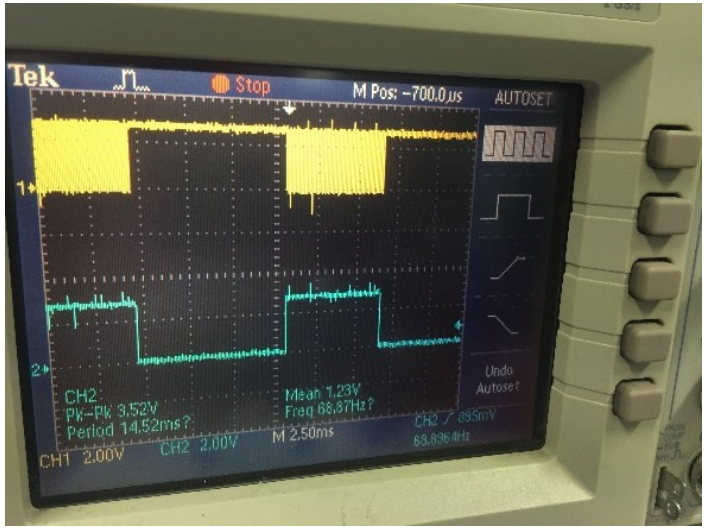
- Arduino
  - Configure image sensor
  - Capture signals from camera.
  - Send data via UART to PC
- PC receives data from an Arduino and saves them as an image.



System setting



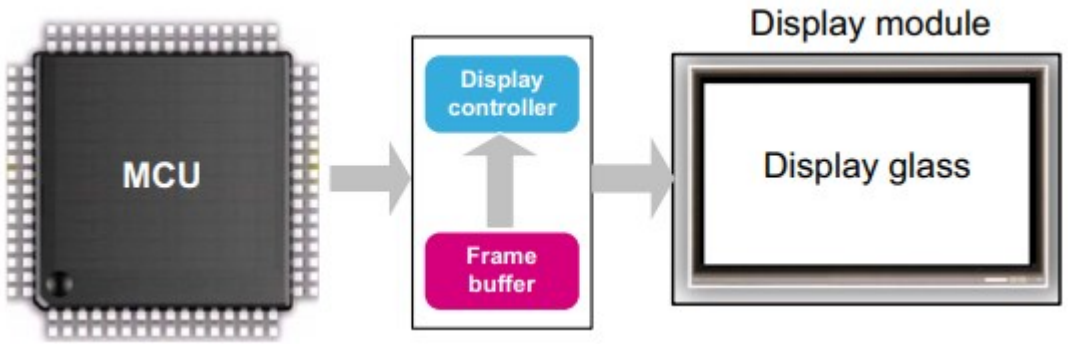
Capture Image



HREF & D[0]

Output: LCD Drive

- The evolution of the mobile, industrial and consumer applications
  - A huge demands of graphical user interfaces (GUIs).
- How to connect to high-resolution display panels directly?
  - Embedded LCD-TFT display controller (LTDC)
  - A basic embedded graphic system
    - An MCU, a framebuffer, a display controller and a display panel.

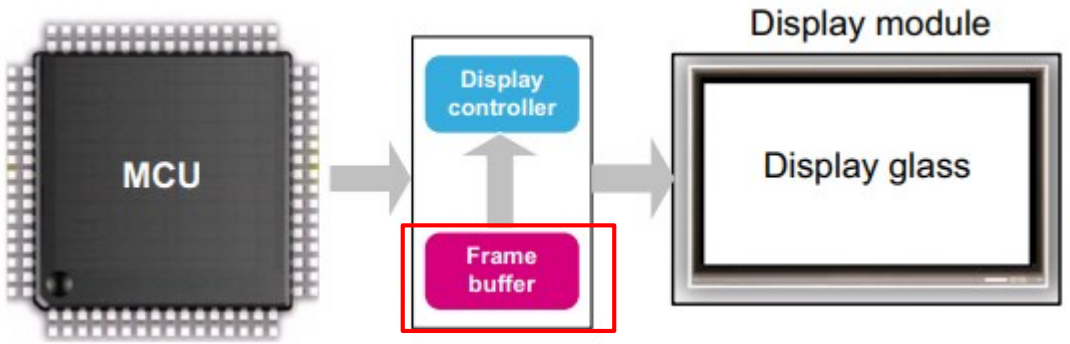


A typical embedded graphic system

Source: LCD-TFT display controller (LTDC) on STM32 MCUs - Application notes

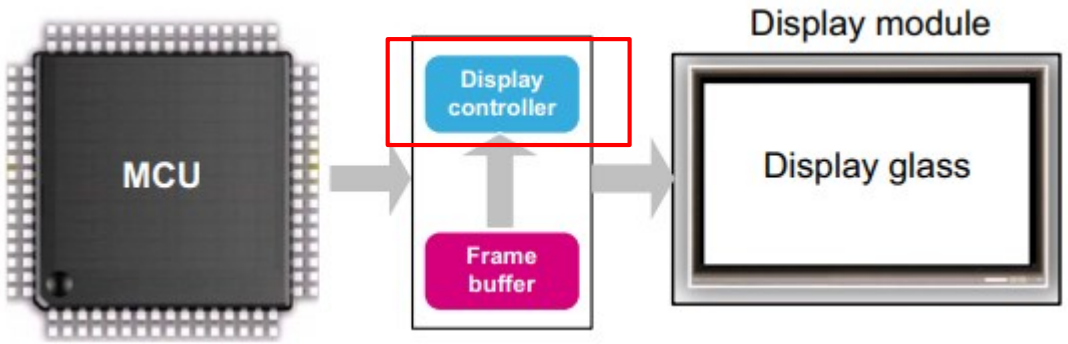
Frame memory buffer

- The framebuffer is used to store pixel data of the image that is displayed
  - The memory is usually called the graphic RAM (GRAM).
  - The MCU computes the image to be displayed in the framebuffer
- Parameters:
  - Framerate: The more often the framebuffer is updated, the higher the frame rate is.
  - Size: depends on the resolution and color depth of the display.



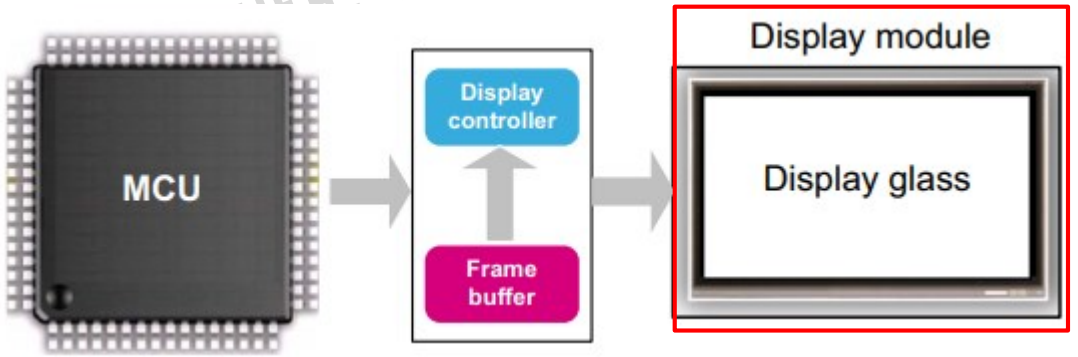
Display Controller

- The display controller continuously does “refreshing” the display
  - Transfer the framebuffer content to the display panel
  - i.e. 60 times per second 60Hz.
- The display controller can be embedded either in MCU or in the display.



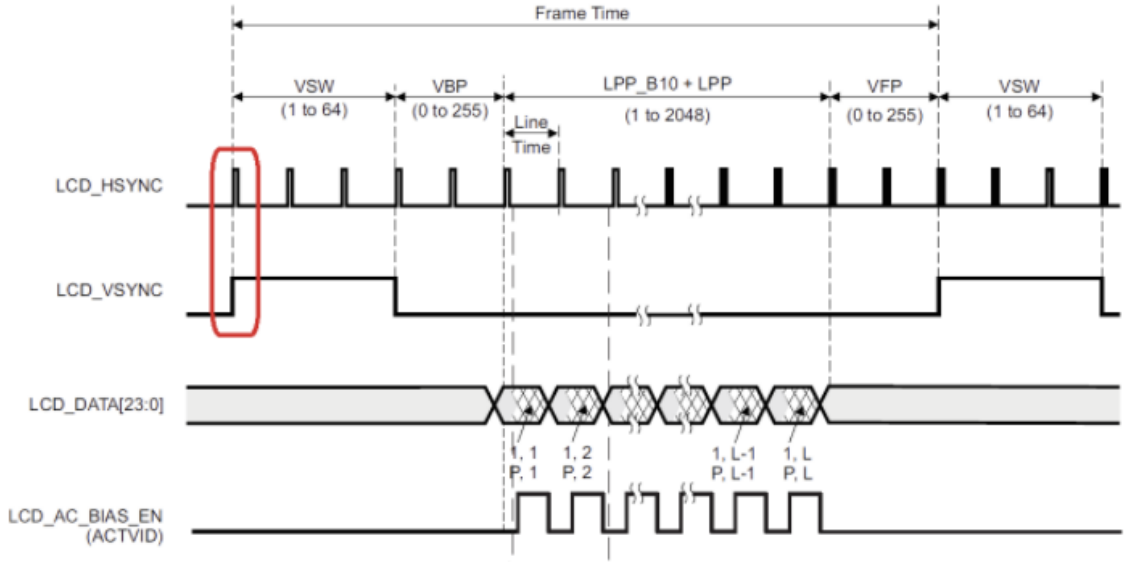
Display Panel

- The display panel is responsible to display the image.
- A display is characterized by:
  - Resolution: the number of pixels
    - Horizontal (pixels number) x Vertical (lines number).
  - Color depth: defined the number of colors
    - It is represented by bits per pixels (bpp)
    - For a color depth of 24 bit per pixel (bpp) (i.e. RGB888): 16,777,216 colors.
  - Refresh rate: the number of times per second that the display panel is refreshed.



Interface signals

- LCD_CLK: Clock
- LCD_VSYNC: the *frame* synchronization signal, manages *vertical* scanning and acts as the frame update strobe.
- LCD_HSYNC: the *line* synchronization signal, manages *horizontal* line scanning and acts as the line display strobe.
- LCD_DE: the DE signal, indicates to the LCD-TFT that the data in RGB bus is valid and must be latched to be drawn.
- Pixel RGB: The LTDC interface can be configured to output more than one color depth.



Note: In the context of this lab, HSYNC is used as a valid signal (DE)*

Road map

- Review
- Image sensor
- Sensor, display model
- Brightness adjustment
- Reversible Color Transform (RCT)

Lab 1a: Sensor model

- Lab 1a: Simulate a frame buffer.
  - Data preparation:
    - Read a BMP file in Matlab
    - Convert it into a hexadecimal file
  - Sensor model
    - Load a hexadecimal file
    - Generate interface signals: VSYNC, HSYNC, R, G, B

Data preparation

- We cannot directly read a bitmap file in Verilog
  - Convert a BMP image to the hexadecimal format.
- This Matlab code converts a bit map image to a hex file.

```
% Input file
src_image_file = 'img/kodim01.bmp';
% Load an input image
img = imread(src_image_file);

% Save pixel values to a buffer
[height,width,nch] = size(img);
buf = zeros(height*width*nch,1);
idx = 1;
for i = height:-1:1
    for j = 1:width
        buf(idx  ) = img(i,j,1); % Red
        buf(idx+1) = img(i,j,2); % Green
        buf(idx+2) = img(i,j,3); % Blue
        idx = idx + 3;
    end
end

% Write a hex file
out_image_file = [src_image_file(1:end-4) '.hex'];
fid = fopen(out_image_file,'wt');
fprintf(fid,'%x\n',buf);
fclose(fid);
```

(1) Load an image

(2) Save all pixels to a buffer.

kodim01.hex | kodim01.hex

1	63	1179640	0
2	63	1179641	0
3	63	1179642	0
4	63	1179643	0
5	63	1179644	0
6	63	1179645	0
7	63	1179646	0
8	63	1179647	0
9	63	1179648	0
10	63	1179649	

(3) Write a hex file.

Sensor model (sensor_model.v)

- Parameters:
  - WIDTH=768, HEIGHT=512
  - INFILE=/img/kodim03.hex
  - Start-up delay (START_UP_DELAY) = 100 (cycles)
    - Before sending data
  - Line delay (HSYNC_DELAY)= 160 (cycles)
- Output signals
  - VSYNC
  - HSYNC
  - RGB data (two pixels)

```
module sensor_model
#(parameter WIDTH    = 768,
            HEIGHT   = 512,
            INFILE   = "./img/kodim03.hex",
            START_UP_DELAY = 100,
            VSYNC_CYCLE = 3,
            VSYNC_DELAY = 3,
            HSYNC_DELAY = 160,
            FRAME_TRANS_DELAY = 200,
            BMP_HEADER_NUM = 54
)
(
    input HCLK,
    input HRESETn,
    output reg VSYNC,
    output reg HSYNC,
    output reg [7:0]  DATA_R0,
    output reg [7:0]  DATA_G0,
    output reg [7:0]  DATA_B0,
    output reg [7:0]  DATA_R1,
    output reg [7:0]  DATA_G1,
    output reg [7:0]  DATA_B1,
    output            ctrl_done
);
```

Sensor model (sensor_model.v)

- For given image data in a file, Verilog supports two commands to read the data file
  - $readmemh: Read a hexadecimal file
  - $readmemb: Read a binary file

```
initial begin
        $readmemh(INFILE, total_memory,0,sizeOfLengthReal_frame-1);
end
// Parse the input pixels
always @ (start) begin
    if(start == 1'b1) begin
        for(i=0; i<WIDTH*HEIGHT*3 ; i=i+1) begin
            temp_BMP[i] = total_memory[i]; //read bmp format image
        end

        for(i=0; i<HEIGHT; i=i+1) begin
            for(j=0; j<WIDTH; j=j+1) begin
                org_R[WIDTH * i + j] = temp_BMP[WIDTH * 3 * (HEIGHT-i-1) + 3*j + 0];
                org_G[WIDTH * i + j] = temp_BMP[WIDTH * 3 * (HEIGHT-i-1) + 3*j + 1];
                ...
            end
        end
    end
end
```

Read data file using $readmemh

The RGB image data are saved into memory.

Finite State Machine (FSM)

- Update the current state (cstate) by the following or next state (nstate)
  - Sequential logic
- Decide the next state based on the current state and other conditions.
  - Combinational logic

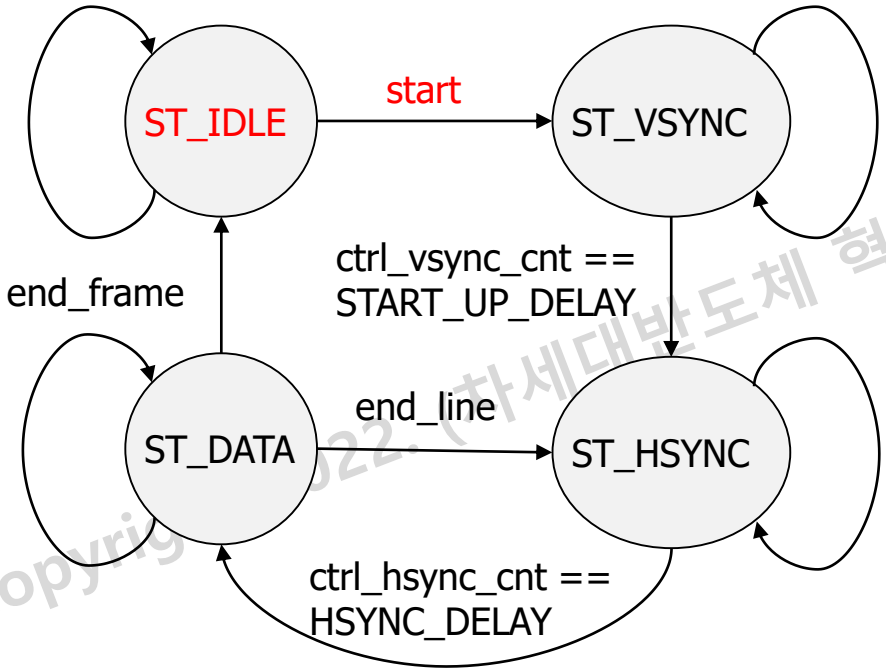
```
always @ (posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        cstate <= ST_IDLE;
    end
    else begin
        cstate <= nstate;
    end
end

always @ (*) begin
    case(cstate)
        ST_IDLE: begin
                if(start)
                    nstate = ST_VSYNC;
                else
                    nstate = ST_IDLE;
        end
        ...
        default: nstate = ST_IDLE;
    endcase
end
```

Update the current state (cstate) by the next state (nstate)

Decide the next state based on the current state and other conditions.

Finite State Machine

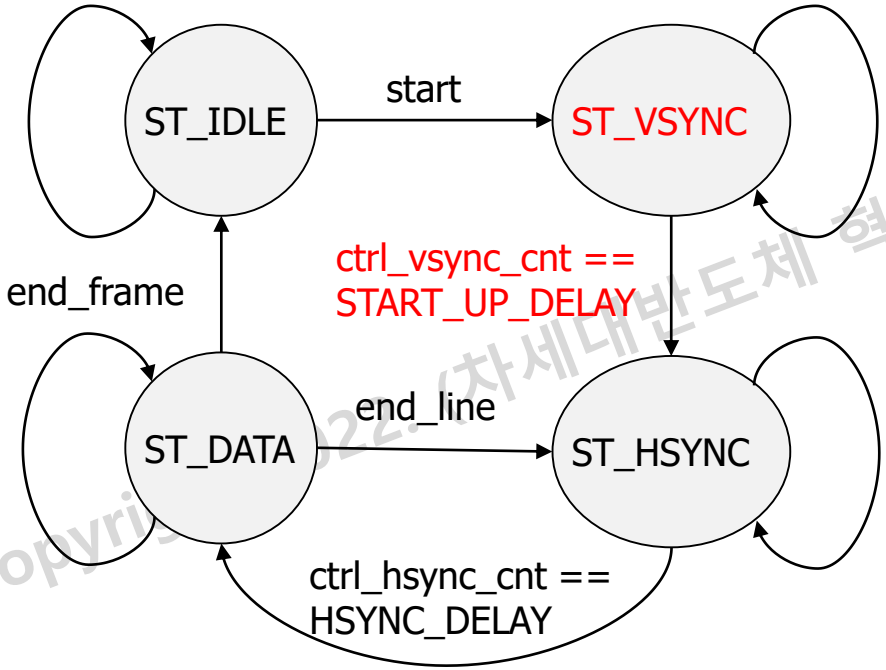
- ST_IDLE:
  - FSM is initialized at ST_IDLE
  - When “start” goes HIGH, FSM moves to ST_VSYNC.




```
always @(*) begin
    case(cstate)
        ST_IDLE: begin
                if(start)
                    nstate = ST_VSYNC;
                else
                    nstate = ST_IDLE;
            end
        ST_VSYNC: begin
                if(ctrl_vsync_cnt == START_UP_DELAY)
                    nstate = ST_HSYNC;
                else
                    nstate = ST_VSYNC;
            end
        ST_HSYNC: begin
                //if(ctrl_hsync_cnt == /* Insert your code here */)
                //    nstate = /* Insert your code here */;
                //else
                //    nstate = ST_HSYNC;
            end
        ST_DATA: begin
                if(ctrl_done) begin   //end of frame
                    //nstate = /* Insert your code here */;
                end
                else begin
                    if(col == WIDTH - 2)    //end of line
                        nstate = ST_HSYNC;
                    else
                        nstate = ST_DATA;
                end
            end
        default: nstate = ST_IDLE;
    endcase
end
```

Finite State Machine

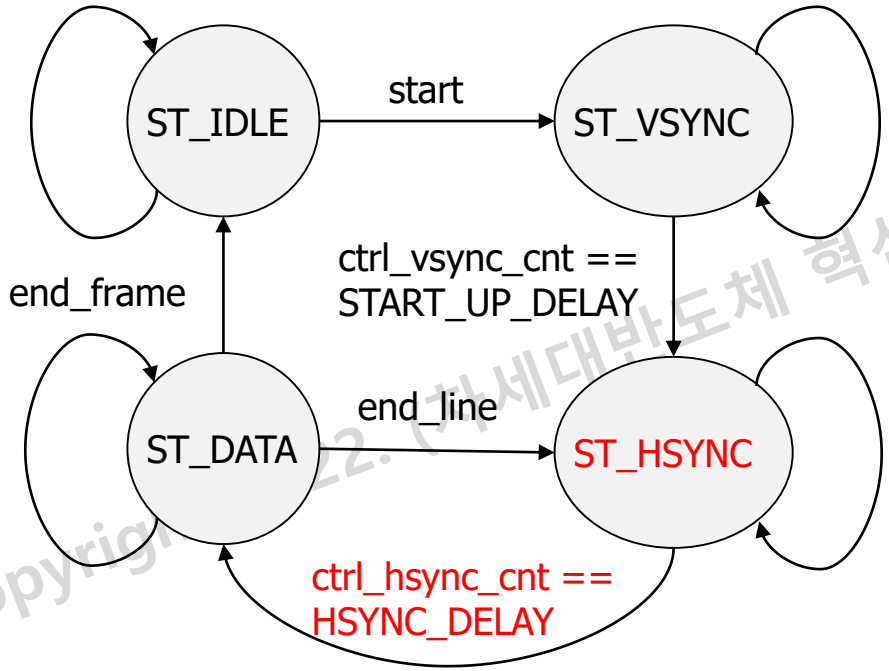
- ST_VSYNC: Frame synchronization
  - Start up for a frame
  - There is a VSYNC counter.
  - When the counter reaches START_UP_DELAY, FSM moves to ST_HSYNC.




```
always @(*) begin
   case(cstate)
      ST_IDLE: begin
            if(start)
               nstate = ST_VSYNC;
            else
               nstate = ST_IDLE;
      end
        ST_VSYNC: begin
                if(ctrl_vsync_cnt == START_UP_DELAY)
                   nstate = ST_HSYNC;
                else
                   nstate = ST_VSYNC;
        end
      ST_HSYNC: begin
            //if(ctrl_hsync_cnt == /* Insert your code here */)
            //    nstate = /* Insert your code here */;
            //else
            //    nstate = ST_HSYNC;
      end
      ST_DATA: begin
            if(ctrl_done) begin   //end of frame
                //nstate = /* Insert your code here */;
            end
            else begin
               if(col == WIDTH - 2)    //end of line
                  nstate = ST_HSYNC;
               else
                  nstate = ST_DATA;
            end
      end
      default: nstate = ST_IDLE;
   endcase
end
```

Finite State Machine

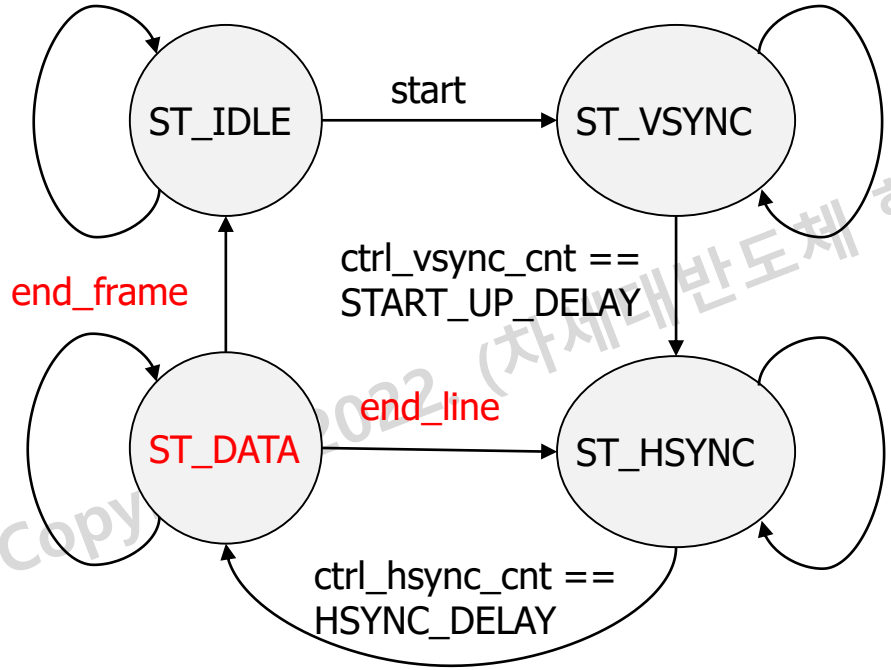
- ST_HSYNC: Line synchronization
  - There is an HSYNC counter.
  - When the counter reaches HSYNC_DELAY, FSM moves to ST_DATA.




```
always @(*) begin
   case(cstate)
      ST_IDLE: begin
            if(start)
               nstate = ST_VSYNC;
            else
               nstate = ST_IDLE;
      end
      ST_VSYNC: begin
            if(ctrl_vsync_cnt == START_UP_DELAY)
               nstate = ST_HSYNC;
            else
               nstate = ST_VSYNC;
      end
        ST_HSYNC: begin
                  //if(ctrl_hsync_cnt == /* Insert your code here */)
                  //    nstate = /* Insert your code here */;
                  //else
                  //    nstate = ST_HSYNC;
        end
      ST_DATA: begin
            if(ctrl_done) begin   //end of frame
                 //nstate = /* Insert your code here */;
            end
            else begin
               if(col == WIDTH - 2)    //end of line
                  nstate = ST_HSYNC;
               else
                  nstate = ST_DATA;
            end
      end
      default: nstate = ST_IDLE;
   endcase
end
```

Finite State Machine

- ST_DATA: Sending pixel data by using two counters
  - Line data counter: if it reaches an **end of a line,** FSM moves to ST_HSYNC.
  - Frame data counter: if it reaches an **end of a frame**, FSM moves to ST_IDLE.




```
always @(*) begin
    case(cstate)
        ST_IDLE: begin
                if(start)
                    nstate = ST_VSYNC;
                else
                    nstate = ST_IDLE;
        end
        ST_VSYNC: begin
                if(ctrl_vsync_cnt == START_UP_DELAY)
                    nstate = ST_HSYNC;
                else
                    nstate = ST_VSYNC;
        end
        ST_HSYNC: begin
                //if(ctrl_hsync_cnt == /* Insert your code here */)
                //    nstate = /* Insert your code here */;
                //else
                //    nstate = ST_HSYNC;
        end
          ST_DATA: begin
                    if(ctrl_done) begin   //end of frame
                          //nstate = /* Insert your code here */;
                    end
                    else begin
                        if(col == WIDTH - 2)    //end of line
                            nstate = ST_HSYNC;
                        else
                            nstate = ST_DATA;
                    end
          end
        default: nstate = ST_IDLE;
    endcase
end
```

Data

- Pixel index is defined by row, col counters
- Each pixel has Red (R), Blue (B), and Green (G) channels. Each channel has 8 bits.
- In this example, we output two pixels per cycle
  - R0, G0, B0
  - R1, G1, B1

```
always @ (*) begin
    VSYNC   = 1'b0;
    HSYNC   = 1'b0;
    DATA_R0 = 0;
    DATA_G0 = 0;
    DATA_B0 = 0;
    DATA_R1 = 0;
    DATA_G1 = 0;
    DATA_B1 = 0;

    if(ctrl_data_run) begin
        VSYNC   = 1'b0;
        HSYNC   = 1'b1;
        DATA_R0 = org_R[WIDTH * row + col   ];
        DATA_G0 = org_G[WIDTH * row + col   ];
        ...
        DATA_R1 = org_R[WIDTH * row + col +1];
        DATA_G1 = org_G[WIDTH * row + col +1];
        ...
        end
end
```

Two output pixels in one cycle.

Test bench (sensor_model_tb.v)

- Test a module sensor_model
  - Clock (HCLK)= 50MHz
  - Reset (HRESETn)
  - Monitor outputs
    - data_R0, data_G0, data_B0
    - data_R1, data_G1, data_B1
    - vsync, hsync

```
sensor_model #(.INFILE(`INPUTFILENAME)) u_sensor_model (
    .HCLK           (HCLK    ),
    .HRESETn        (HRESETn ),
    .VSYNC          (vsync   ),
    .HSYNC          (hsync   ),
    .DATA_R0        (data_R0 ),
    .DATA_G0        (data_G0 ),
    .DATA_B0        (data_B0 ),
    .DATA_R1        (data_R1 ),
    .DATA_G1        (data_G1 ),
    .DATA_B1        (data_B1 ),
    .ctrl_done (enc_done)
);
```

Input file

module instance for test bench

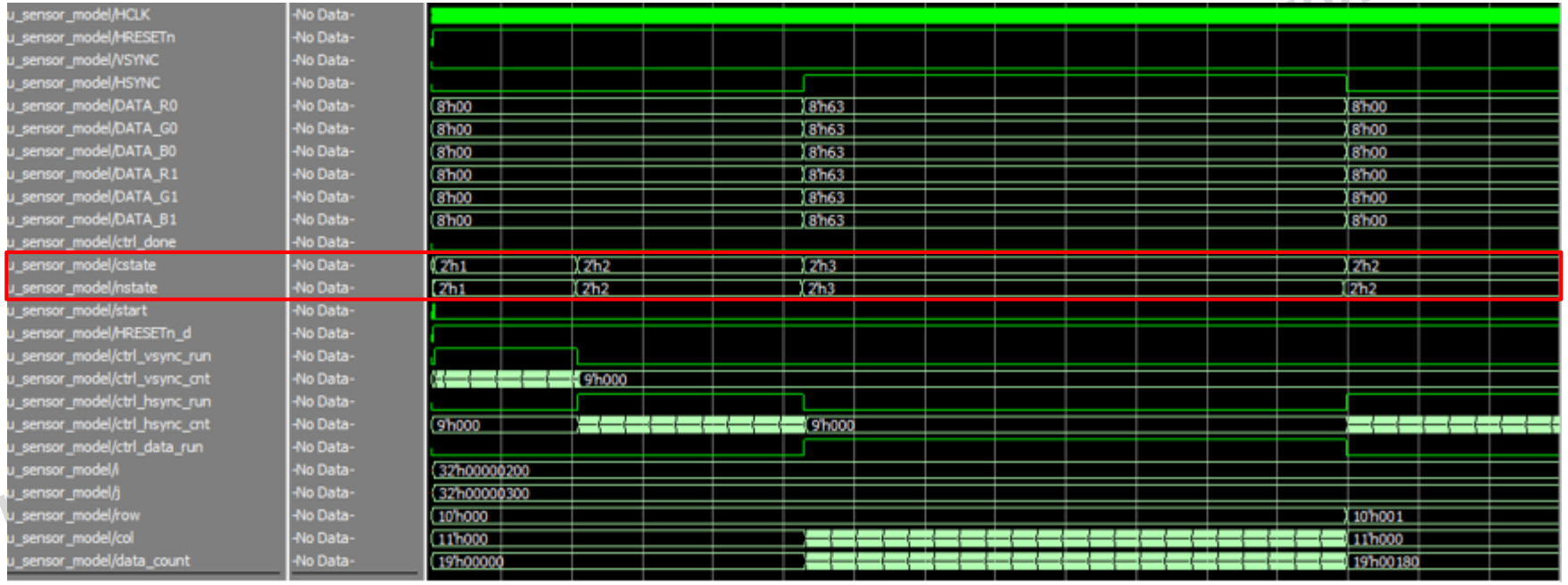
```
initial begin
    HCLK = 0;
    forever #10 HCLK = ~HCLK;
end
```

reset, clock signals generation

```
initial begin
    HRESETn     = 0;
    #25 HRESETn = 1;
end
```

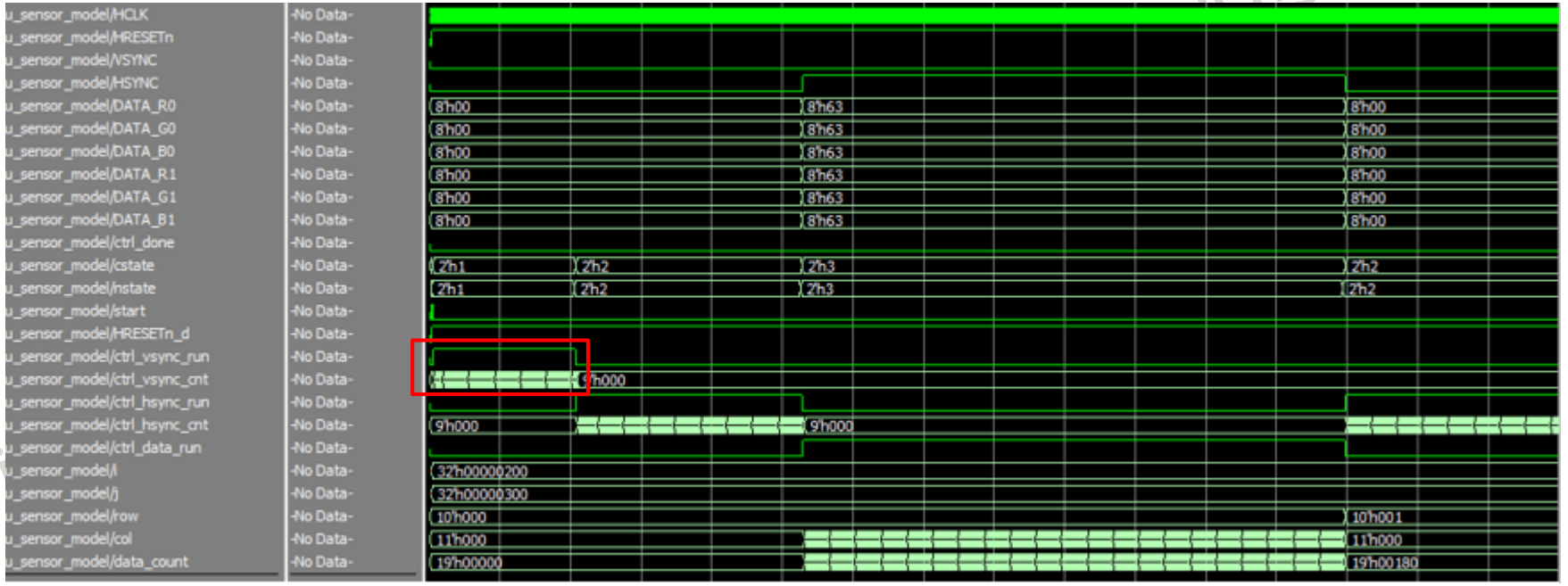
Waveform

- Run a simulation with time = 16,000 ns
- cstate transition: 0 -> 1 -> 2 -> 3 -> 2 ...



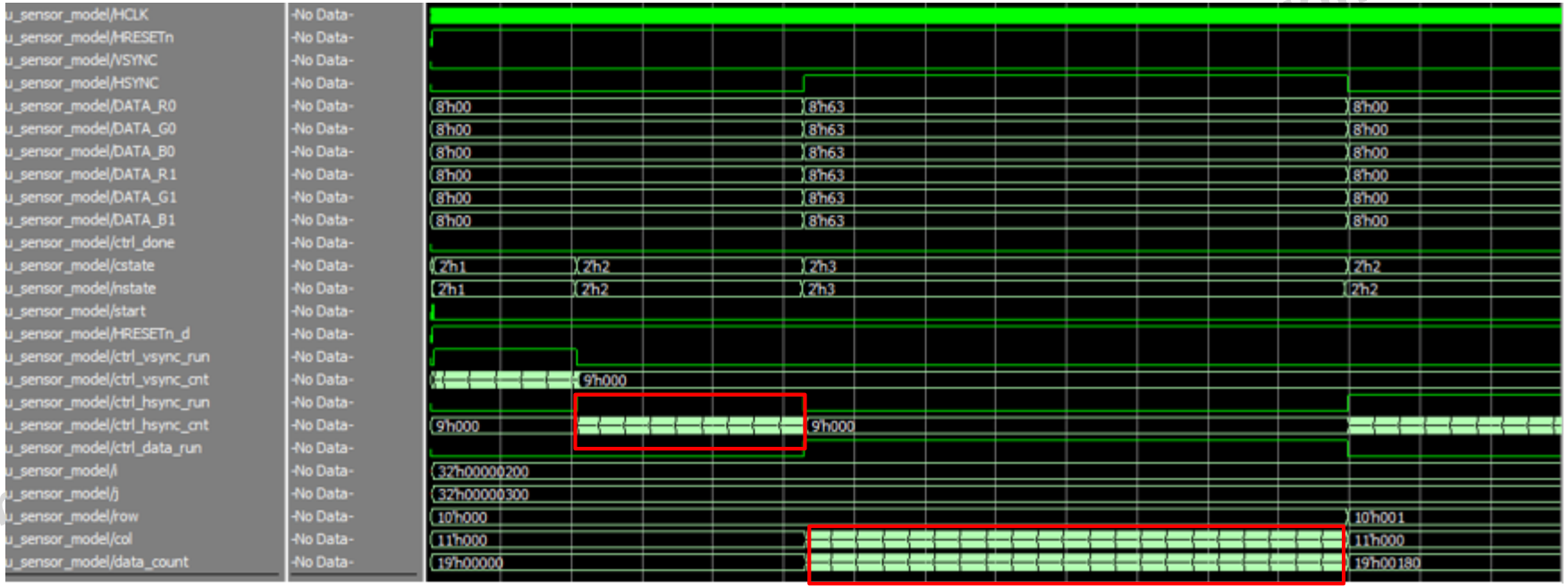
Waveform

- Start a frame
  - When cstate == 1 (VSYNC)
    - ctrl_vsync_cnt increases every cycle
    - After ctrl_vsync_cnt reaches 100, state transition occurs.



Waveform

- Generate pixel data line by line
  - When cstate == 2 (HSYNC)
    - A counter ctrl_hsync_cnt increases every cycle
    - After ctrl_hsync_cnt reaches 160, state transition occurs (line delays)



To do …

- Completing the missing codes (**sensor_model.v**)

```
always @(*) begin
    case(cstate)
        ST_IDLE: begin
                if(start)
                    nstate = ST_VSYNC;
                else
                    nstate = ST_IDLE;
        end
        ST_VSYNC: begin
                if(ctrl_vsync_cnt == START_UP_DELAY)
                    nstate = ST_HSYNC;
                else
                    nstate = ST_VSYNC;
        end
        ST_HSYNC: begin
                if(ctrl_hsync_cnt == /* Insert your code here */)
                    nstate = /* Insert your code here */;
                else
                    nstate = ST_HSYNC;
        end
        ST_DATA: begin
                if(ctrl_done)    //end of frame
                    nstate = /* Insert your code here */;
                else begin
                    if(col == WIDTH - 2)    //end of line
                        nstate = ST_HSYNC;
                    else
                        nstate = ST_DATA;
                    end
        end
        default: nstate = ST_IDLE;
    endcase
end
```

```
// Output
always @(*) begin
    VSYNC   = 1'b0;
    HSYNC   = 1'b0;
    DATA_R0 = 0;
    DATA_G0 = 0;
    DATA_B0 = 0;
    DATA_R1 = 0;
    DATA_G1 = 0;
    DATA_B1 = 0;
    if(ctrl_data_run) begin
        VSYNC   = 1'b0;
        HSYNC   = 1'b1;
        DATA_R0 = org_R[WIDTH * row + col   ];
        DATA_G0 = org_G[WIDTH * row + col   ];
        /* Insert your code here */
        //DATA_B0 ;
        DATA_R1 = org_R[WIDTH * row + col +1];
        DATA_G1 = org_G[WIDTH * row + col +1];
        /* Insert your code here */
        //DATA_B1 ;
        end
end
```

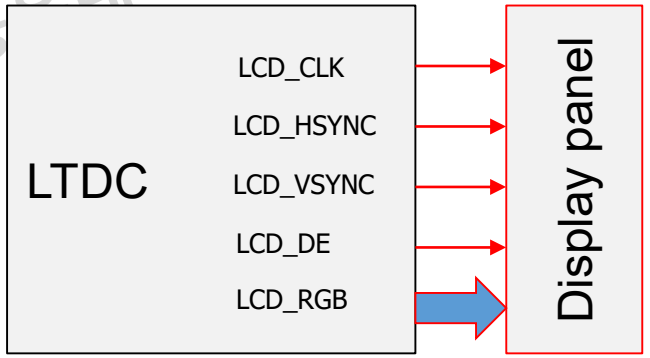
To do …

- Completing the missing codes (**sensor_model.v)**
- Do a simulation with time = 16,000 ns
- Show the output waveform
- How to verify the results?
  - Check state transition
  - Precise verification can be done after Lab1b (using a display panel model)

```
//------------------------------------------------
// Input processing
//------------------------------------------------
//{{{

// Read the input file to memory
initial begin
        $readmemh(INFILE, total_memory,0,sizeOfLengthReal_frame-1);
end
// Parse the input pixels
always@(start) begin
    if(start == 1'b1) begin
        for(i=0; i<WIDTH*HEIGHT*3 ; i=i+1) begin
            temp_BMP[i] = total_memory[i]; //read bmp format image
        end

        for(i=0; i<HEIGHT; i=i+1) begin
            for(j=0; j<WIDTH; j=j+1) begin
                org_R[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)+3*j+0];
                org_G[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)+3*j+1];
                /* Insert your code here */
                //org_B[] = ;
                /*************************/
            end
        end
    end
end
```

Lab 1b: Display panel model

- Lab 1b: Simulate the display panel (display_model.v)
- Scopes
  - Data preparation: Input signals come from a sensor model
  - Bitmap
  - Display panel model
    - Write a bitmap file in Verilog



Bitmap header (1/2)

- A Windows bitmap image starts with 54-byte header data.
- **Image size** = 768*512*3= 1179648 bytes

  BMP header = 54 bytes

  BMP File size = Image size + BMP Header = 1179702 Bytes
- $1179702_{10} = 120036_{hex}$
  - Then 4-byte size of BMP file: 12_{hex}= 18, 36_{hex}= 54

    BMP_header[2] = 54;

    BMP_header[3] = 0 ;

    BMP_header[4] = 18;

    BMP_header[5] = 0 ;

Bitmap header (2/2)

- A Windows bitmap image starts with 54-byte header data.
- **Image width** = 768 => In hexadecimal: 0x0300. The 4 bytes of the image width are 0, 3, 0, 0.

  BMP_header[18] = 0;

  BMP_header[19] = 3;

  BMP_header[20] = 0;

  BMP_header[21] = 0;
- **Image height** = 512 => In hexadecimal: 0x0200. The 4 bytes of the image width are 0, 2, 0, 0.

  BMP_header[22] = 0;

  BMP_header[23] = 2;

  BMP_header[24] = 0;

  BMP_header[25] = 0;
- *Remaining parameters are set to 0.*

Buffer

- Assume that the panel model has an internal buffer to store pixel data.
- For incoming pixel data, the buffer is updated.
- Note that the input is two pixels.

```
always @(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
        for(k=0;k<WIDTH*HEIGHT*3;k=k+1) begin
            out_BMP[k] <= 0;
        end
    end else begin
        if(RECON_VALID) begin
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+2] <= DATA_RECON_R0;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+1] <= DATA_RECON_G0;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m  ] <= DATA_RECON_B0;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+5] <= DATA_RECON_R1;
            ...
        end
    end
end
```

Store two pixels to the buffer

Counters

- Line pixel counter:
  - Increase by one for any incoming data.
  - Reset at the end of line.
- Line counter
  - Increase by one if reaching to the end of line.

```
always @ (posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
        l <= 0;
        m <= 0;
    end else begin
        if(RECON_VALID) begin
            if(m == WIDTH/2-1) begin
                m <= 0;
                l <= l + 1;
            end else begin
                m <= m + 1;
            end
        end
    end
end
```

When reaching end of line, increase l and reset m

Else increase m by 1

Write an bmp image

- When all pixels are received, the display panel model writes an bmp image.

```
initial begin
    // Open file
    fd = $fopen(INFILE, "wb+");
end

always @ (DEC_DONE) begin
    if(DEC_DONE == 1'b1) begin
        // Write header
        for(i=0; i<BMP_HEADER_NUM; i=i+1) begin
            $fwrite(fd, "%c", BMP_header[i][7:0]);
        end

        // Write data
        for(i=0; i<WIDTH*HEIGHT*3; i=i+6) begin
            $fwrite(fd, "%c", out_BMP[i  ][7:0]);
            $fwrite(fd, "%c", out_BMP[i+1][7:0]);
            $fwrite(fd, "%c", out_BMP[i+2][7:0]);
            $fwrite(fd, "%c", out_BMP[i+3][7:0]);
            ...
        end
    end
end
```

Step 1: Open a file for writing.

Step 2: Write the header.

Step 3: Write the pixel data.

Test bench (fmc_top_tb.v)

- Top module
  - A sensor model
  - A display model
  - Outputs of sensor_model is connected to inputs of the display model

```
assign recon_valid = hsync;
assign recon_data_R0 = data_R0;
assign recon_data_G0 = data_G0;
assign recon_data_B0 = data_B0;
assign recon_data_R1 = data_R1;
assign recon_data_G1 = data_G1;
assign recon_data_B1 = data_B1;
```

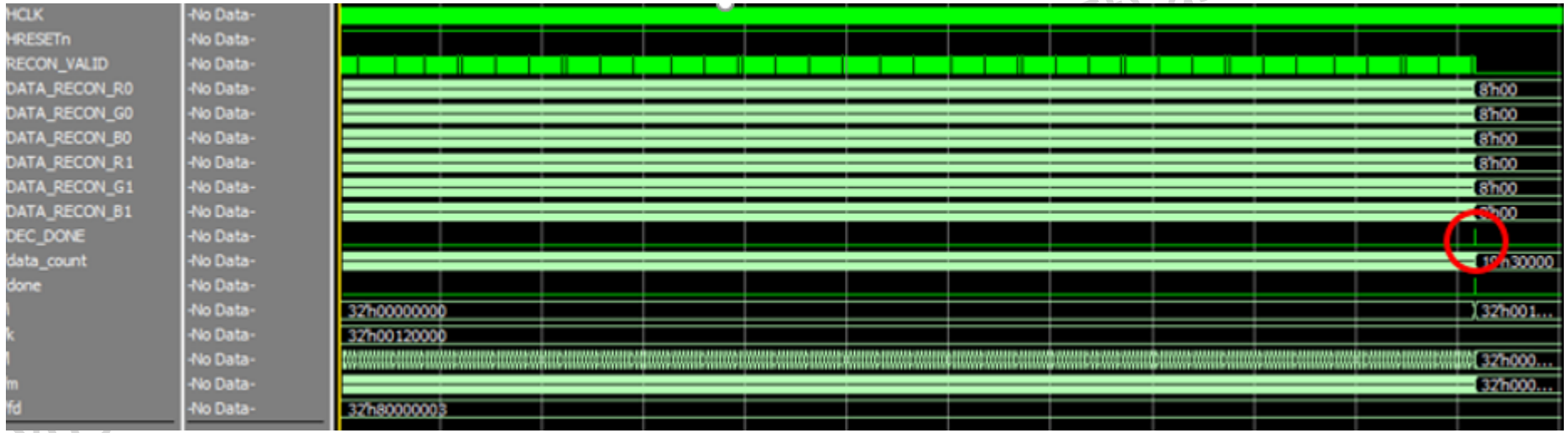
```
sensor_model #(.INFILE(`INPUTFILENAME))   Input file
u_sensor_model (
    .HCLK           (HCLK    ),
    .HRESETn        (HRESETn ),
    .VSYNC          (vsync   ),
    .HSYNC          (hsync   ),
    .DATA_R0        (data_R0 ),
    .DATA_G0        (data_G0 ),
    .DATA_B0        (data_B0 ),
    .DATA_R1        (data_R1 ),
    .DATA_G1        (data_G1 ),
    .DATA_B1        (data_B1 ),
    .ctrl_done (enc_done)
);
```

module instance for test bench

```
display_model #(.INFILE(`OUTPUTFILENAME))
u_display_model(
    ./*input */       HCLK(HCLK),
    ./*input */       HRESETn(HRESETn),
    ./*input */       RECON_VALID(recon_valid),
    ./*input [7:0]  */DATA_RECON_R0(recon_data_R0),
    ./*input [7:0]  */DATA_RECON_G0(recon_data_G0),
    ./*input [7:0]  */DATA_RECON_B0(recon_data_B0),
    ./*input [7:0]  */DATA_RECON_R1(recon_data_R1),
    ./*input [7:0]  */DATA_RECON_G1(recon_data_G1),
    ./*input [7:0]  */DATA_RECON_B1(recon_data_B1),
    ./*output       */DEC_DONE()
);
```

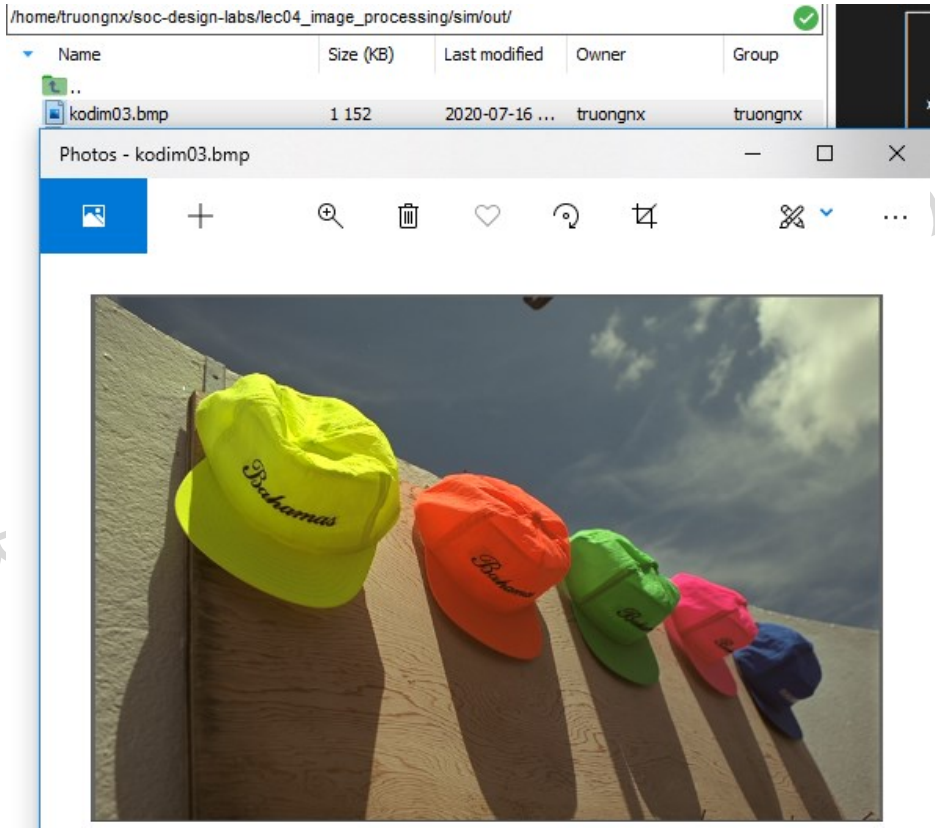
Output result

- Run 6ms
  - Check the output file
    - Output file is at ./out/kodim03.bmp by default



Waveform

- Run 6 milliseconds
  - DEC_DONE is triggered at the end.



To do …

- Complete the missing codes
- Do a simulation with time = 6 ms
- Verify the output file

```
// Update the internal buffers.
always@(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
        for(k=0;k<WIDTH*HEIGHT*3;k=k+1) begin
            out_BMP[k] <= 0;
        end
    end else begin
        if(RECON_VALID) begin
            out_BMP[WIDTH*3*(HEIGHT-1-l)+6*m+2] <= DATA_RECON_R0;
            out_BMP[WIDTH*3*(HEIGHT-1-l)+6*m+1] <= DATA_RECON_G0;
            out_BMP[WIDTH*3*(HEIGHT-1-l)+6*m  ] <= DATA_RECON_B0;
            out_BMP[WIDTH*3*(HEIGHT-1-l)+6*m+5] <= DATA_RECON_R1;
            /*Insert your code*/
            //out_BMP[] <= DATA_RECON_G1;
            //out_BMP[] <= DATA_RECON_B1;
            /******************/
        end
    end
end
```

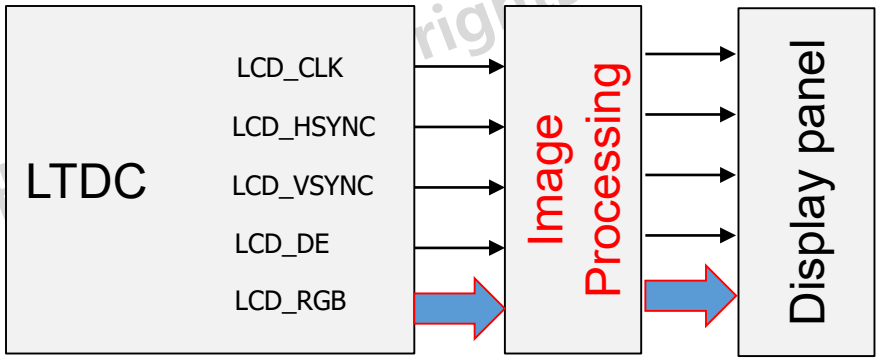
```
// Write the output file
//{{{
initial begin
    // Open file
    fd = $fopen(INFILE, "wb+");
end

always@(DEC_DONE) begin
    if(DEC_DONE == 1'b1) begin
        // Write header
        for(i=0; i<BMP_HEADER_NUM; i=i+1) begin
            $fwrite(fd, "%c", BMP_header[i][7:0]);
        end

        // Write data
        for(i=0; i<WIDTH*HEIGHT*3; i=i+6) begin
            $fwrite(fd, "%c", out_BMP[i  ][7:0]);
            $fwrite(fd, "%c", out_BMP[i+1][7:0]);
            $fwrite(fd, "%c", out_BMP[i+2][7:0]);
            $fwrite(fd, "%c", out_BMP[i+3][7:0]);
            /* Insert your code */
            //$fwrite(fd, "%c", out_BMP[][]);
            //$fwrite(fd, "%c", out_BMP[][]);
            /*********************/
        end


    end
end
//}}}
```

Road map

- Review
- Image sensor
- Sensor, display model
- **Brightness adjustment**
- Reversible Color Transform (RCT)

Lab 2: Brightness adjustment

- We aim to simulate two image-processing applications
- Scopes
  - Change the RGB outputs
  - Two applications
    - Brightness adjustment
    - (Reversible) color transform (RCT)



Brightness adjustment

- Sometimes, we want to control the brightness of the monitor.
- Two modes:
  - Brighter: Increase the pixel values
  - Darker: Decrease the pixel values

Lab 3a: brightness_adjustment.v

- Inputs
  - Clock, reset
  - Input pixels r0, g0, b0, r1, g1, b1 and a valid signal (in_valid)
  - mode, value[7:0]
- Outputs: out_r0, out_g0, out_b0, out_r1, out_g1, out_b1 with an output valid signal (out_valid)

```
module brightness_adjustment
#(
    parameter IMG_PIX_W  = 8,
              WAVE_PIX_W = 10
)
(
    input clk,
    input rst_n,
    input in_valid,
    input mode,
    input [IMG_PIX_W-1:0] value,
    input [IMG_PIX_W-1:0] r0, g0, b0,
    input [IMG_PIX_W-1:0] r1, g1, b1,
    output reg out_valid,
    output reg [IMG_PIX_W-1:0] out_r0, out_g0, out_b0, out_r1, out_g1, out_b1
);
```

Brightness adjustment

- Mode: one bit, 1: brighter, 0: darker
- Value: the amount of color change
  - Mode = 1
    - Output pixel = input pixel + value
    - If output pixel is larger than 255
      - Assign it to 255
  - Mode = 0
    - Output pixel = input pixel – value
    - If output pixel is smaller than 0
      - Assign it to zero

```
always @(posedge clk, negedge rst_n) begin
    if(~rst_n) begin
        out_valid <= 1'b0;
        out_r0 <= 0;
        out_g0 <= 0;
        out_b0 <= 0;
        out_r1 <= 0;
        out_g1 <= 0;
        out_b1 <= 0;
    end
    else begin
        out_valid <= in_valid;
        if (mode == 1'b1) begin   //brighter
            out_r0 <= ({1'b0, r0} + {1'b0, value} > 255) ? 255 : r0 + value;
            out_g0 <= ({1'b0, g0} + {1'b0, value} > 255) ? 255 : g0 + value;
            out_b0 <= ({1'b0, b0} + {1'b0, value} > 255) ? 255 : b0 + value;
            out_r1 <= ({1'b0, r1} + {1'b0, value} > 255) ? 255 : /* Insert your code here*/;
            out_g1 <= ({1'b0, g1} + {1'b0, value} > 255) ? 255 : /* Insert your code here*/;
            out_b1 <= ({1'b0, b1} + {1'b0, value} > 255) ? 255 : /* Insert your code here*/;
        end
        else begin                //darker
            out_r0 <= (r0 < value) ? 0 : r0 - value;
            out_g0 <= (g0 < value) ? 0 : g0 - value;
            out_b0 <= (b0 < value) ? 0 : b0 - value;
            out_r1 <= (r1 < value) ? 0 : /* Insert your code here*/;
            out_g1 <= (g1 < value) ? 0 : /* Insert your code here*/;
            out_b1 <= (b1 < value) ? 0 : /* Insert your code here*/;
        end
    end
end
```

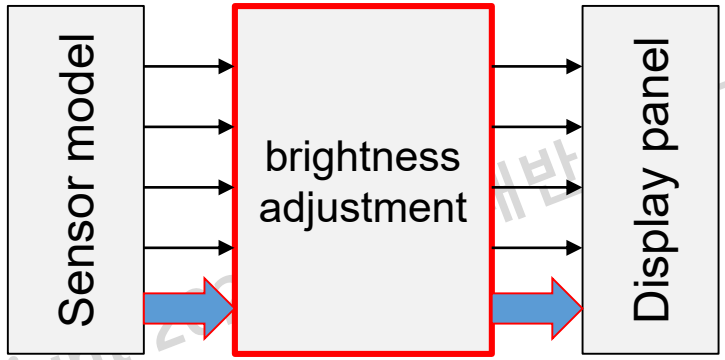
To do …

- Complete the missing code
- Run simulation with time = 6ms
- Show the results

```
always @(posedge clk, negedge rst_n) begin
    if(~rst_n) begin
        out_valid <= 1'b0;
        out_r0 <= 0;
        out_g0 <= 0;
        out_b0 <= 0;
        out_r1 <= 0;
        out_g1 <= 0;
        out_b1 <= 0;
    end
    else begin
        out_valid <= in_valid;                                    Clipping
        if (mode == 1'b1) begin    //brighter
            out_r0 <= ({1'b0, r0} + {1'b0, value} > 255) ? 255 : r0 + value;
            out_g0 <= ({1'b0, g0} + {1'b0, value} > 255) ? 255 : g0 + value;
            out_b0 <= ({1'b0, b0} + {1'b0, value} > 255) ? 255 : b0 + value;
            out_r1 <= ({1'b0, r1} + {1'b0, value} > 255) ? 255 : /* Insert your code here*/;
            out_g1 <= ({1'b0, g1} + {1'b0, value} > 255) ? 255 : /* Insert your code here*/;
            out_b1 <= ({1'b0, b1} + {1'b0, value} > 255) ? 255 : /* Insert your code here*/;
        end
        else begin                 //darker
            out_r0 <= (r0 < value) ? 0 : r0 - value;
            out_g0 <= (g0 < value) ? 0 : g0 - value;
            out_b0 <= (b0 < value) ? 0 : b0 - value;
            out_r1 <= (r1 < value) ? 0 : /* Insert your code here*/;
            out_g1 <= (g1 < value) ? 0 : /* Insert your code here*/;
            out_b1 <= (b1 < value) ? 0 : /* Insert your code here*/;
        end
    end
end
```

Test bench (fmc_top_tb.v)

- A “brightness_adjustment” instance is in between the sensor and the display
  - Inputs come from the sensor
  - Outputs go to the display



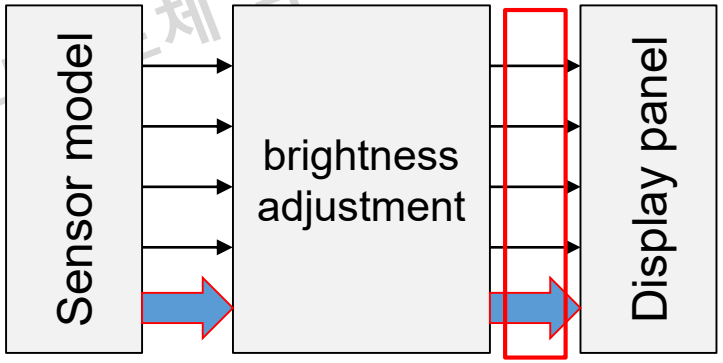

```
brightness_adjustment #(.IMG_PIX_W(IMG_PIX_W))
u_brightness_adjustment(
    /*input*/ .clk(HCLK),
    /*input*/ .rst_n(HRESETn),
    /*input*/ .in_valid(hsync),
    /*input*/ .mode(br_mode),
    /*input [IMG_PIX_W-1:0]*/ .value(br_value),
    /*input [IMG_PIX_W-1:0]*/ .r0(data_R0),
    /*input [IMG_PIX_W-1:0]*/ .g0(data_G0),
    /*input [IMG_PIX_W-1:0]*/ .b0(data_B0),
    ...
    /*output [IMG_PIX_W-1:0]*/ .out_g1(br_data_G1),
    /*output [IMG_PIX_W-1:0]*/ .out_b1(br_data_B1)
);
```

To do …

- Reuse the codes in Lab 1
- Modify the connections to connect a brightness adjustment module to a display model

```
/* connect sensor module to display module */
assign recon_valid = hsync;
assign recon_data_R0 = data_R0;
assign recon_data_G0 = data_G0;
assign recon_data_B0 = data_B0;
assign recon_data_R1 = data_R1;
assign recon_data_G1 = data_G1;
assign recon_data_B1 = data_B1;
```

```
/* connect brightness adjustment module to display module*/
assign recon_valid = br_out_valid;
assign recon_data_R0 = br_data_R0;
assign recon_data_G0 = br_data_G0;
assign recon_data_B0 = br_data_B0;
assign recon_data_R1 = br_data_R1;
assign recon_data_G1 = br_data_G1;
assign recon_data_B1 = br_data_B1;
```



To do …

- Test with different modes and store the results
  - **Brighter** mode when
    - Value = 100
    - Value = 50
  - **Darker** mode when
    - Value = 100
    - Value = 50

```
/* Insert your code here */
//assign br_mode = ;     //1: brighter, 0: darker
//assign br_value = ;    //amount of adjustment

brightness_adjustment
#(.IMG_PIX_W(IMG_PIX_W),
  .WAVE_PIX_W(WAVE_PIX_W))
        u_brightness_adjustment
(
    /*input*/ .clk(HCLK),
    /*input*/ .rst_n(HRESETn),
    /*input*/ .in_valid(hsync),
    /*input*/ .mode(br_mode),
    /*input [IMG_PIX_W-1:0]*/ .value(br_value),
    /*input [IMG_PIX_W-1:0]*/ .r0(data_R0),
    /*input [IMG_PIX_W-1:0]*/ .g0(data_G0),
    /*input [IMG_PIX_W-1:0]*/ .b0(data_B0),
    /*input [IMG_PIX_W-1:0]*/ .r1(data_R1),
    /*input [IMG_PIX_W-1:0]*/ .g1(data_G1),
    /*input [IMG_PIX_W-1:0]*/ .b1(data_B1),
    /*output*/ .out_valid(br_out_valid),
    /*output [IMG_PIX_W-1:0]*/ .out_r0(br_data_R0),
    /*output [IMG_PIX_W-1:0]*/ .out_g0(br_data_G0),
    /*output [IMG_PIX_W-1:0]*/ .out_b0(br_data_B0),
    /*output [IMG_PIX_W-1:0]*/ .out_r1(br_data_R1),
    /*output [IMG_PIX_W-1:0]*/ .out_g1(br_data_G1),
    /*output [IMG_PIX_W-1:0]*/ .out_b1(br_data_B1)
);
```

Results

- Integrate the model with the controller model and the display model
- Do a simulation with a time = 6 ms and show the results at different options

Original image

darker, 50

darker, 100

brighter, 50

brighter, 100

Road map

Review

Image sensor

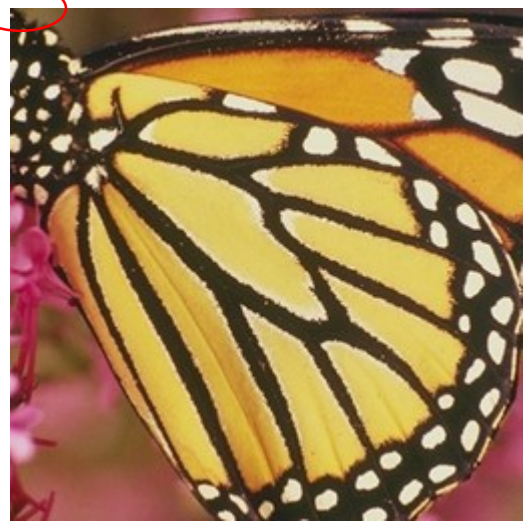
Sensor, display model

Brightness adjustment

Reversible Color Transform (RCT)

Image format

- An image can be represented using **three color channels** such as red, green, and blue.
  - An RGB pixel: 2^{24} colors for a pixel in {(0, 0, 0), (0, 0, 1), …, (255, 255, 255)}
- Observations
  - Color channels are likely to be similar
  - Pixels in a small block have similar values



42	58	85	98	115	120	124	124
41	47	54	72	96	106	113	123
80	52	42	42	59	70	97	114
225	136	38	35	37	62	89	97
108	99	44	27	33	53	78	97
30	33	37	35	33	35	46	90
37	37	38	33	34	34	46	74
45	42	42	37	34	34	50	98

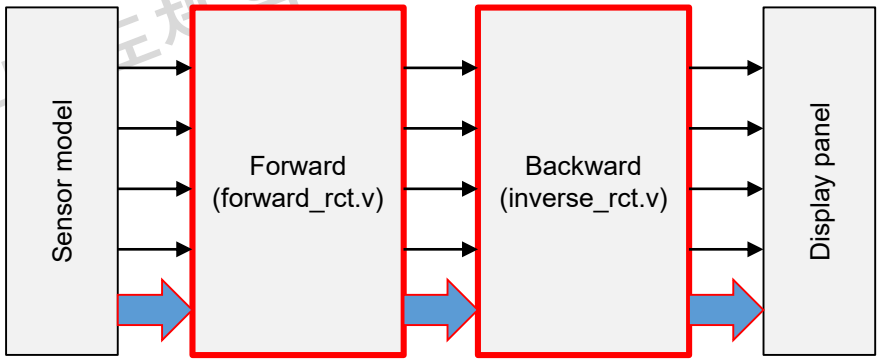
30	43	66	76	89	92	96	97
23	29	38	56	77	83	86	95
62	32	25	35	45	53	74	88
208	118	23	27	25	48	69	72
90	84	34	25	26	45	63	74
17	22	29	33	27	27	28	66
29	33	36	34	32	29	32	53
39	38	38	35	29	29	37	80

22	26	42	43	53	53	56	54
12	13	19	34	49	51	51	55
46	14	12	23	25	25	42	50
193	102	13	23	15	28	44	37
74	68	24	23	19	32	41	41
6	11	24	30	24	20	10	37
24	26	32	28	27	25	17	26
34	32	34	29	25	25	26	61

Image Red Green Blue

Lab 3: Reversible color transform (RCT)

- Lab 3: Implement a reversible color transform (RCT)
  - Color channels are usually similar
  - In image compression, RCT is used to convert between two color domains
    - Forward RCT: RGBRGB to YCbCr
    - Backward RCT: YCbCr to RGB



Forward RCT (rct_forward.v)

- Convert pixels from the RGB domain to the YCbCr domain
- Inputs
  - Clock, reset
  - Input pixels r0, g0, b0, r1, g1, b1 and a valid signal (in_valid)
- Outputs: y0, y1, cb0, cb1 and an output valid signal (out_valid)

```
module forward_rct
#(
    parameter IMG_PIX_W  = 8,
              WAVE_PIX_W = 10
)
(
    input clk,
    input rst_n,
    input in_valid,
    input [IMG_PIX_W-1:0] r0, g0, b0,
    input [IMG_PIX_W-1:0] r1, g1, b1,
    output reg out_valid,
    output reg [WAVE_PIX_W-1:0] y0, y1, cb0, cr0
);
```

Forward RCT (rct_forward.v)

- Convert pixels from the RGB domain to the YCbCr domain
- $\begin{bmatrix} y \\ cb \\ cr \end{bmatrix} = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 0 & -1 & 1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix}$

- Sampling:
  - Ignore cb1, cr1

```
/*Signals*/
wire [WAVE_PIX_W-1:0] tmp_y0 = {2'b00, r0} + {1'b0, g0,1'b0} + {2'b00, b0};//r0 + (g0<<<1) + b0
wire [WAVE_PIX_W-1:0] tmp_y1 = {2'b00, r1} + {1'b0, g1,1'b0} + {2'b00, b1};//r1 + (g1<<<1) + b1
wire [WAVE_PIX_W-1:0] tmp_cb = {2'b00, b0} - {2'b00, g0};//b0-g0
wire [WAVE_PIX_W-1:0] tmp_cr = {2'b00, r0} - {2'b00, g0};//r0-g0
/*Sequential logic*/
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        y0  <= 0;
        y1  <= 0;
        cb0 <= 0;
        cr0 <= 0;
        out_valid <= 0;
    end else begin
        if(in_valid) begin
            y0  <= {2'b0,tmp_y0[9:2]};
            //y1  <= /*Insert your code here*/;
            cb0 <= tmp_cb;
            //cr0 <= /*Insert your code here*/;
            out_valid <= 1;
        end else begin
            y0  <= 0;
            y1  <= 0;
            cb0 <= 0;
            cr0 <= 0;
            out_valid <= 0;
        end
    end
end
```

Backward/inverse RCT (rct_inverse.v)

- Convert pixels from the YCbCr domain to the RGB domain
- Inputs
  - Clock, reset
  - Input pixels y0, cb0, cr0, y1, cb1, cr1 and a valid signal (in_valid)
- Outputs: r0, g0, b0, r1, g1, b1 and an output valid signal (out_valid)

```
module inverse_rct
#(
    parameter IMG_PIX_W  = 8,
              WAVE_PIX_W = 10
)
(
    input clk,
    input rst_n,
    input in_valid,
    input signed [WAVE_PIX_W-1:0] y0, cb0, cr0,
    input signed [WAVE_PIX_W-1:0] y1, cb1, cr1,
    //input [WAVE_PIX_W-1:0] y0, cb0, cr0,
    //input [WAVE_PIX_W-1:0] y1, cb1, cr1,
    output reg out_valid,
    output [IMG_PIX_W-1:0] r0, g0, b0,
    output [IMG_PIX_W-1:0] r1, g1, b1
);
```

RCT backward/inverse (rct_inverse.v)

- Convert pixels from the RGB domain to the YCbCr domain

$$tmp = Y - (Cb + Cr) \gg 2$$
$$R = Cr + tmp$$
$$G = tmp$$
$$B = Cb + tmp$$

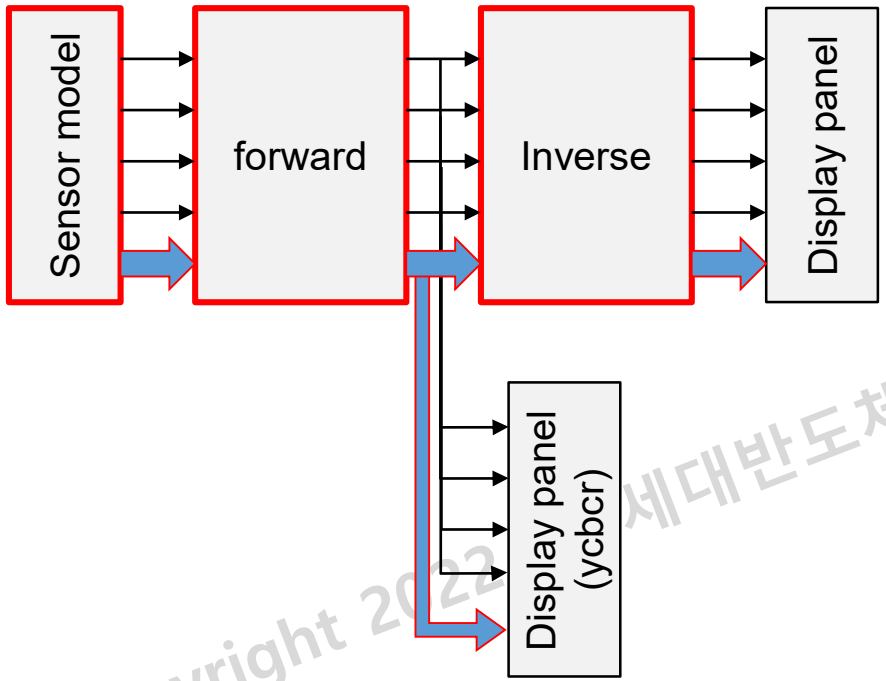
- Clipping
  - If the pixel is larger than 255, assign it to 255
  - If the pixel is smaller than 0, assign it to zero

```
assign r0 = (r0_tmp[WAVE_PIX_W-1])? 0 : (r0_tmp[WAVE_PIX_W-2])? 255 : r0_tmp[IMG_PIX_W-1:0];
assign g0 = (g0_tmp[WAVE_PIX_W-1])? 0 : (g0_tmp[WAVE_PIX_W-2])? 255 : g0_tmp[IMG_PIX_W-1:0];
assign b0 = (b0_tmp[WAVE_PIX_W-1])? 0 : (b0_tmp[WAVE_PIX_W-2])? 255 : b0_tmp[IMG_PIX_W-1:0];
assign r1 = (r1_tmp[WAVE_PIX_W-1])? 0 : (r1_tmp[WAVE_PIX_W-2])? 255 : r1_tmp[IMG_PIX_W-1:0];
assign g1 = (g1_tmp[WAVE_PIX_W-1])? 0 : (g1_tmp[WAVE_PIX_W-2])? 255 : g1_tmp[IMG_PIX_W-1:0];
assign b1 = (b1_tmp[WAVE_PIX_W-1])? 0 : (b1_tmp[WAVE_PIX_W-2])? 255 : b1_tmp[IMG_PIX_W-1:0];
```

```
/*Combinational logic*/
assign tmp0 = y0 - ((cb0 + cr0) >>> 2);
assign tmp1 = y1 - ((cb1 + cr1) >>> 2);

/*Sequential logic*/
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        r0_tmp <= 0;
        g0_tmp <= 0;
        b0_tmp <= 0;
        r1_tmp <= 0;
        g1_tmp <= 0;
        b1_tmp <= 0;
        out_valid <= 0;
    end else begin
        if(in_valid) begin
            r0_tmp <= cr0 + tmp0;
            g0_tmp <= tmp0;
            b0_tmp <= cb0 + tmp0;
            //r1_tmp <= /*Insert your code here*/;
            //g1_tmp <= /*Insert your code here*/;
            //b1_tmp <= /*Insert your code here*/;
            out_valid <= 1;
        end else begin
            r0_tmp <= 0;
            g0_tmp <= 0;
            b0_tmp <= 0;
            r1_tmp <= 0;
            g1_tmp <= 0;
            b1_tmp <= 0;
            out_valid <= 0;
        end
    end
end
```

Test bench (fmc_rct_top_tb.v)

- Modules: Sensor model, Forward RCT, Inverse RCT and Two displays



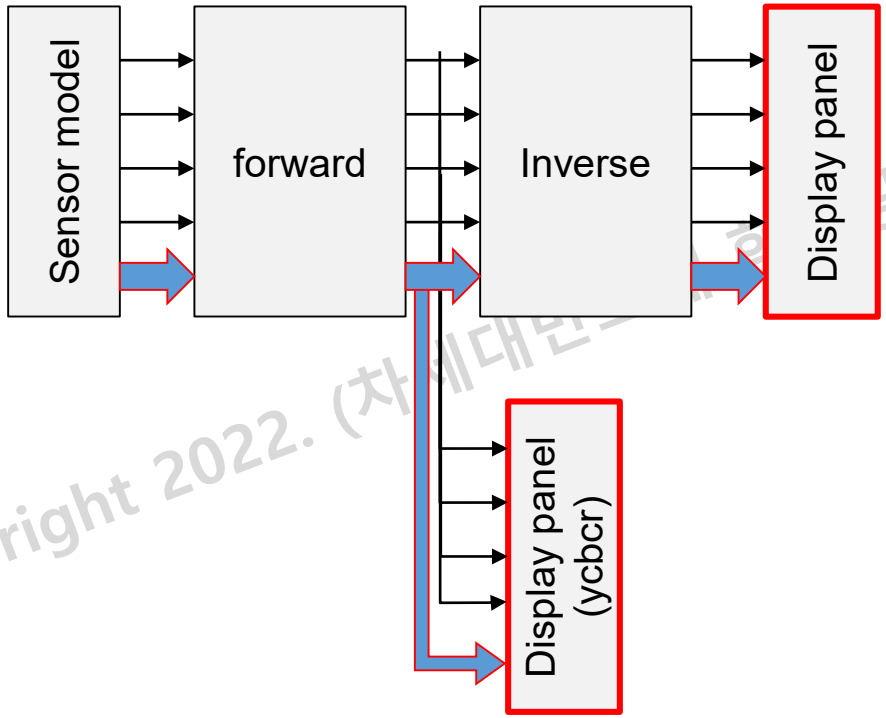

```
sensor_model
#(.INFILE(`INPUTFILENAME))
u_sensor_model (
    .HCLK           (HCLK    ),
    .HRESETn        (HRESETn ),
    .VSYNC          (vsync   ),
    .HSYNC          (hsync   ),
    .DATA_R0        (data_R0 ),
    .DATA_G0        (data_G0 ),
    .DATA_B0        (data_B0 ),
    ...
    .ctrl_done      (enc_done)
);
```

```
forward_rct u_forward_rct (
    /* input*/.clk      (HCLK    ),
    /* input*/.rst_n    (HRESETn ),
    /* input*/.in_valid (hsync   ),
    /* input*/.r0       (data_R0 ),
    /* input*/.g0       (data_G0 ),
    /* input*/.b0       (data_B0 ),
    ...
);
```

```
inverse_rct u_inverse_rct (
    ./*input */clk      (HCLK),
    ./*input */rst_n    (HRESETn),
    ./*input */in_valid (rct_o_valid),
    ./*input*/y0        (rct_dout_y0),
    ./*input */cb0      (rct_dout_cb),
    ./*input*/cr0       (rct_dout_cr),
...
);
```

Test bench (fmc_rct_top_tb.v)

- Use two display instances to visualize two images
  - RCT transformed image
  - Reconstructed image

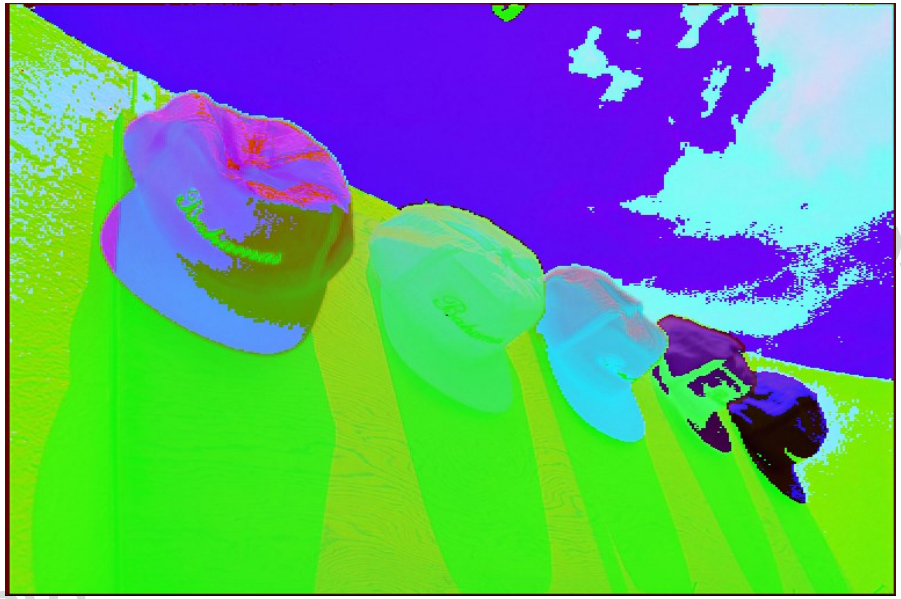



```
camera_sensor #(.INFILE(`OUTPUTFILENAME))
u_camera_sensor(
    ./*input */HCLK(HCLK),
    ./*input */ HRESETn(HRESETn),
    ./*input */ RECON_VALID(recon_valid),
    ./*input [7:0]  */ DATA_RECON_R0(recon_data_R0),
    ./*input [7:0]  */ DATA_RECON_G0(recon_data_G0),
    ./*input [7:0]  */ DATA_RECON_B0(recon_data_B0),
    ...
    ./*output */DEC_DONE()
);
```

```
camera_sensor#(.INFILE(`OUTPUTFILENAME2))
u_camera_sensor2(
    ./*input */HCLK(HCLK),
    ./*input */ HRESETn(HRESETn),
    ./*input */ RECON_VALID(rct_o_valid),
    ./*input [7:0]  */ DATA_RECON_R0(rct_dout_y0),
    ./*input [7:0]  */ DATA_RECON_G0(rct_dout_cb),
    ./*input [7:0]  */ DATA_RECON_B0(rct_dout_y1),
    ...
    ./*output */DEC_DONE()
);
```

To do …

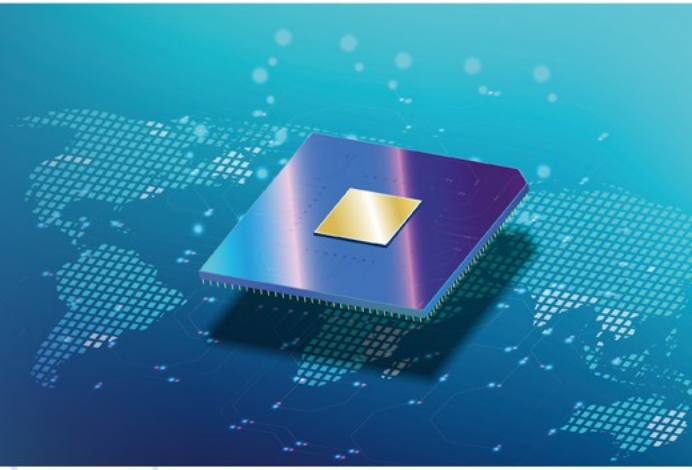
- Implement RCT forward and backward modules by completing the missing codes
- Do a simulation with time = 6 ms (fmc_rct_top_tb.v)
- Check the output images



./out/kodim03_ycbcr.bmp

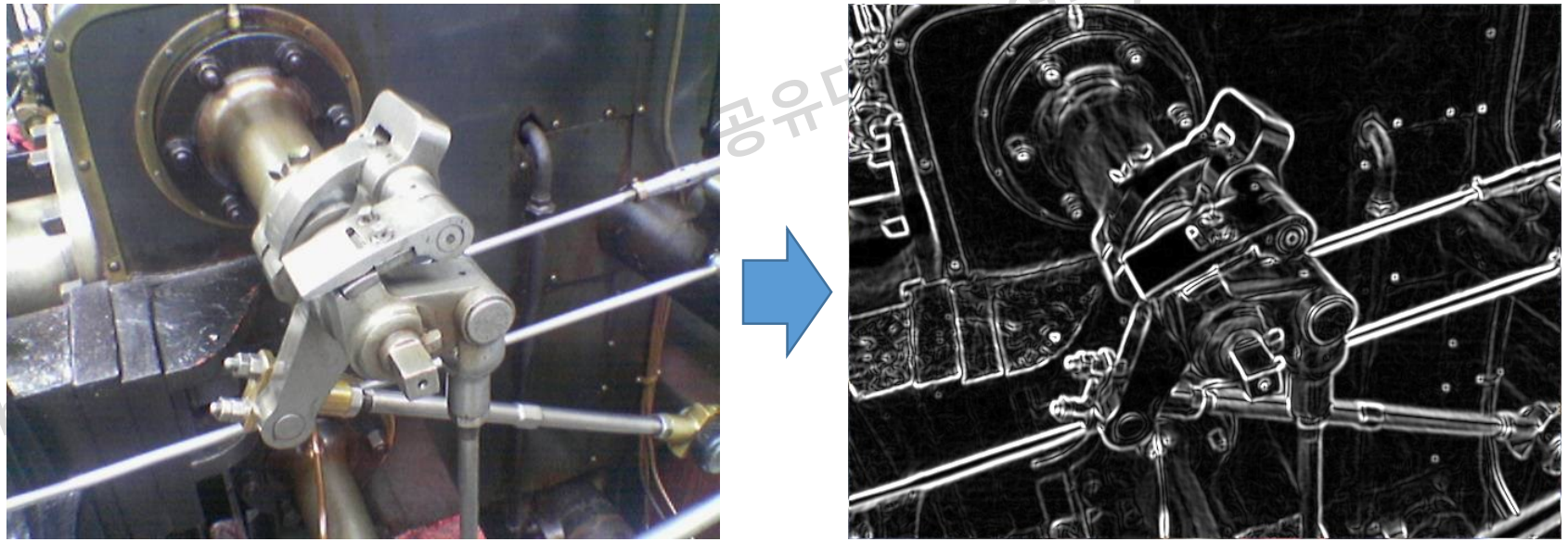
./out/kodim03.bmp

Backup slides
Sobel filter



Sobel operator

- The **Sobel filter** is used in image processing and computer vision, particularly within **edge detection** algorithms where it creates an image emphasizing edges.
- The operator uses two 3×3 kernels, which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes and one for vertical.



https://en.wikipedia.org/wiki/Sobel_operator

Sobel operator

- If we define **A** as the source image and $\mathbf{G}_x$ and $\mathbf{G}_y$ are two images which at each point contain the horizontal and vertical derivative approximations, respectively, the computations are as follows:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A, \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

where * here denotes the 2-dimensional signal processing *convolution* operation.

- The *x*-coordinate is defined here as increasing in the "right"-direction
- The *y*-coordinate is defined as increase in the "down"-direction.
- An approximated gradient magnitude at each image point:

$$G = \sqrt{G_x^2 + G_y^2}$$

Sobel operator: Decomposition

- Efficient implementation:
  - Only eight image points around a point are needed to compute the corresponding result
  - Only integer arithmetic is needed to compute the gradient vector approximation.

- Furthermore, the two discrete filters described above are both separable:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \end{bmatrix}$$

⇒ Count the number of addition/subtraction and shift operations