CSEE4840 Embedded Systems Barcode Scanner Report

Matthew Modi (mem2382), Ananya Haritsa (ah4308), Helen Bovington (hab2175), Rahul Pulidindi (rp3254), Kamil Zajkowski (kmz2123)
Columbia University | Spring 2025

Table of Contents

Overview

EAN-13 Background

In the United States, EAN-13 barcodes are commonly used on retail goods and books. Retail goods are marked as described by the UPC-A standard with a leading “0” followed by a 12 digit identifier. Books use the ISBN standard which is a subset of the GTIN standard, where the country code is marked as 978 or 979 and commonly referred to as “Bookland”.

Each EAN-13 barcode consists of:

- A 3-digit GS1 prefix (country or organization code)
- A manufacturer code
- A product code
- A checksum digit, which is calculated using a modulo-10 algorithm for error detection.

EAN-13 is designed for optical scanning, and the encoded information is not stored as characters but as a sequence of bar widths and spacings, with strict rules for start, middle, and end guards, and left/right digit parity patterns. This makes it ideal for real-time decoding from images or video frames, such as in our system. Our project leverages this predictable encoding structure to decode bar widths from a single scanline of a barcode image, allowing us to extract the 13-digit GTIN from visual data captured by an OV7670 camera. This approach mimics the working principle of physical barcode scanners and offers a hands-on demonstration of digital image processing, signal sampling, and hardware/software co-design.

Our Approach

In our group's final project for Embedded Systems, we made a barcode scanner using the OV7670 camera module. Our system reads UPC-A barcodes using the camera and displays 12-digit UPC (Universal Product Code) at the output of the barcode scanning algorithm code. EAN-12 is a barcode symbology defined by GS1 US, the American branch (🦅) of the global GS1 organization responsible for developing and maintaining barcode standards. Outside of the United States, there is also an international 13-digit barcode standard maintained by GS1.

Each EAN-13 barcode consists of:

- A 3-digit GS1 prefix (country or organization code)
- A manufacturer code
- A product code
- A checksum digit, which is calculated using a modulo-10 algorithm for error detection.

Due to the limited availability of international products, our group specified our data processing for the 12 digit barcode, but the system could be easily adapted to account for the 13 digit barcode scanner. This project implements a hybrid hardware-software system for decoding EAN-13 barcodes, using the OV7670 camera and the DE1-SoC development board. The system leverages both the programmable logic (FPGA) and

the integrated Hard Processor System (HPS) on the Cyclone V SoC. After data is received and processed on the FPGA, individual pixels are sent to the algorithm, implemented in C++, which decodes the data into the original barcode as taken by the camera.

Hardware

This project requires extensive hardware and software development to fully implement. At a high level, the camera takes a photo of the barcode and turns each of the pixels it receives into 2 bytes of data, in 565 Red-Green-Blue format. The camera frame is 480 rows long and 640 pixels (1280 bytes) wide. After the shutter of the camera is pressed, a verilog module applies its logic to the several output waveforms of the camera until it passes the middle row of data, theoretically the most informationally robust, to a FIFO. The FIFO passes the information to an Avalon bus that allows the data bits to be read by the device driver, and finally processed by the algorithm. We chose this project because of the intensive hardware-software interface development needed and unique challenge of processing data collected externally to the software. It offers a hands-on demonstration of digital image processing, signal sampling, and hardware/software co-design.

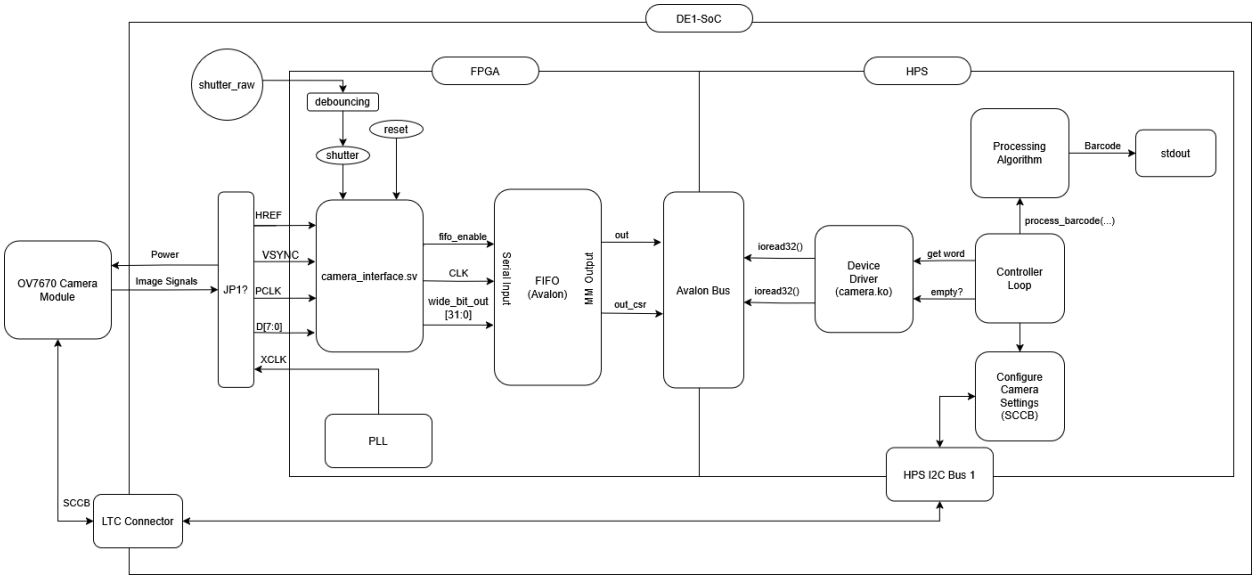
Software & Design Justification

We implemented a custom device driver that enables the transfer of RGB color data from the FPGA to the HPS. Specifically, the driver exposes the RGB values of each pixel in the middle row of the captured image frame. We offloaded as much image processing as possible to software running on the HPS for the following reasons:

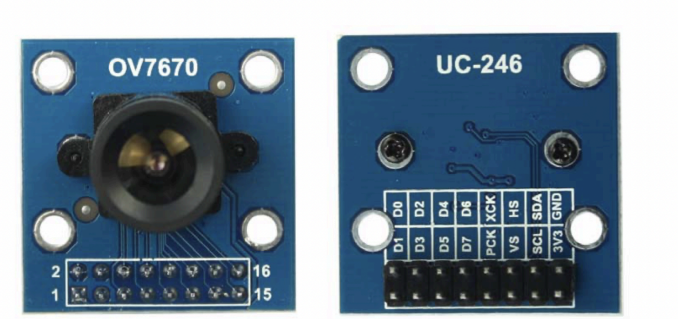
1) Performing most of the data processing on software allows for faster iteration. Developing and testing processing algorithms in software allows for significantly faster iteration. Compiling software changes typically takes under a minute, while recompiling FPGA fabric can take 15 minutes or more.
2) Software also allows for greater flexibility. Software provides more adaptability to changing conditions. For example, if ambient lighting changes or we need to fine-tune denoising, thresholding, or filtering parameters, we can update the software without requiring time-consuming hardware recompilation.

This hybrid approach enables us to leverage the FPGA for efficient data acquisition while retaining the flexibility and rapid development cycle of software-based processing.

Block Diagram



OV7670 Camera Module



Interfacing OV7670 camera

- [Module Specification]
- [Chip Specification]
- [SCCB Protocol Specification]

To capture barcode image data, our project uses the OV7670, a compact, low-power CMOS image sensor that outputs 8-bit VGA video data at up to 30 frames per second. It communicates with a host device through a semi-proprietary Serial Camera Control Bus (SCCB) for configuration, similar to I2C. The camera outputs formatted image data in RGB565 format, which requires assembling two 8-bit data values per pixel using the PCLK signal. The timing and data synchronization rely on HREF (line valid) and VSYNC (frame sync) signals, which the camera_interface.sv verilog module on the programmed FPGA will monitor to capture one full horizontal row of pixel data per barcode scan.

Ideally, the captured row is the center row to allow for the most robust receiving of data. The synchronization of the signals PCLK, VSYNC, HREF, and DATA[7:0] are shown in the screenshot below [1]:

Figure 5 Horizontal Timing

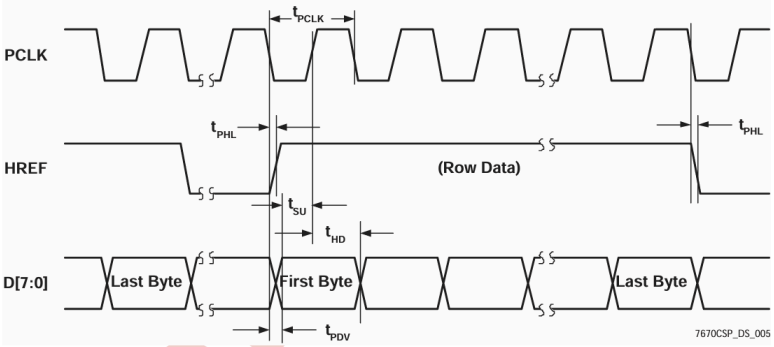
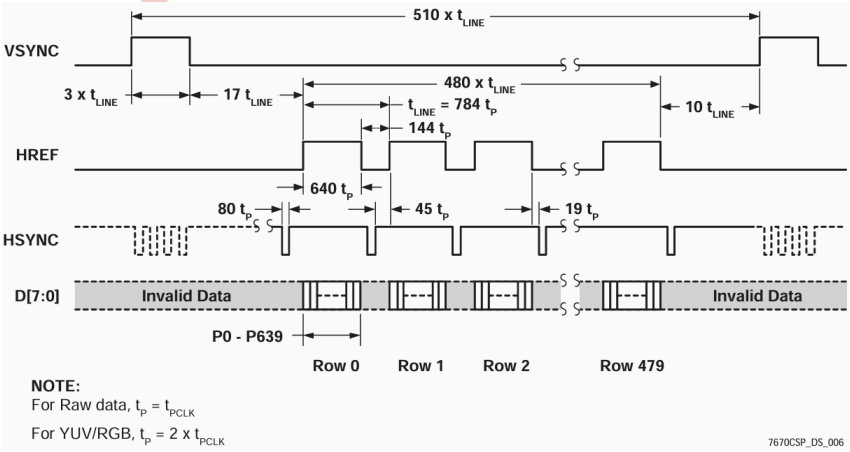
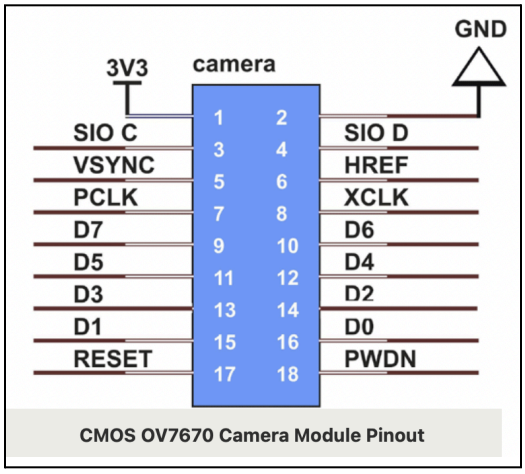


Figure 6 VGA Frame Timing



Additionally, the pinout of the camera is shown below:

[1] OV7670_DS (1.4).fm



CMOS OV7670 Camera Module Pinout

We reviewed the OV7670 datasheet: (OV7670 Camera Module Datasheet (Rev. C, PDF)), which outlines key capabilities such as exposure control, gamma correction, white balance, color saturation, and hue control, all configurable through SCCB. These settings allow us to fine-tune the image quality if needed, especially for robust performance under varied lighting conditions — which is critical for accurate barcode decoding. After the SCCB protocol was set up and the signal synchronization understood, camera_interface.sv was implemented to allow for data transmission from the camera. The camera_interface.sv code is discussed in the next section.

Camera Interface (***camera_interface.sv***)

At a high level, the FPGA fabric is designed to perform the following functionality:

1) Capture button of camera being pressed.
   When an outside user presses the button to the camera, a flag is set signaling to the FPGA to prepare to capture the data collected from the photo.
2) Read the middle row of pixels.
   For the duration of the next frame coming into the FPGA fabric, wait for the middle row of pixels.
3) Save the middle row of pixels.
   Upon determining the end of the middle row, set the flag back to its original status, indicating that the FPGA should not save the remainder of the incoming data.
4) Interface with the HPS.
   A mutex or handshake signal may be used to pause the HPS from reading shared memory during capture, ensuring data integrity. Extract one row of pixels when a button is pressed. The HPS can then safely access the captured row for further software-side decoding of the barcode.

The camera_interface.sv module is engineered to interface with the OV7670 camera, facilitating the capture and processing of pixel data for subsequent digital applications. It employs a Mealy-style finite state machine (FSM) comprising four distinct states: RESET, SHUTTER, WRITE, and BLOCK.

Upon reset, which is triggered by the shutter, the FSM initializes counters and prepares the system for data acquisition. The camera is continuously transmitting data, and each frame is barred on either side of the data transmission by VSYNC barring high for 4704 clk cycles. We are able to determine with specificity that VSYNC bars high for 4704 clk cycles by the timing diagram which shows VSYNC high for three t_{LINE}.

$$N\times\ t_{CLK} = (3\ \times t_{LINE})\ \times \frac{784\times t_P}{t_{LINE}} \times\ \frac{2\times t_{CLK}}{t_P} = 4704\ \times t_{CLK}$$

The timing diagram for the output of the camera is incredibly detailed, which allowed us to come to similarly detailed understanding of the other camera output signals. After the shutter is pressed, the verilog module waits for the next rising edge of VSYNC, which transitions the FSM into the SHUTTER state. In the SHUTTER state, it monitors the HREF signal to increment column and row counters, determining the position within the frame. When the middle row (row 239) is reached, the FSM transitions to the WRITE state, where it begins assembling 32-bit words from consecutive 8-bit pixel values. These words are constructed over four clock cycles and output via wide_bit_out[32:0]. Although the data arrives from the camera in 8 bit chunks, it needs to leave the camera_interface.sv module in 32 bit chunks because the FIFO requires a 32 bit input. It would be possible to pad the 24 MSB of the FIFO input with zeros and only feed in 8 bits at a time, but doing so would be wasteful and it is a more elegant solution to deliver the data to the FIFO in 32 bit chunks. A fifo_enable signal is asserted to indicate the availability of valid data for downstream processing every four clock cycles, or every time four 8-bit packets of camera data have been packaged into a 32 bit chunk. Once the desired data is captured and transmitted to the FIFO, the FSM enters the BLOCK state, blocking any other data transmission until another valid shutter is captured.

To manage the pixel data, the module instantiates eight flip flop submodules, each acting as a D-type flip-flop for one bit of the 8-bit data bus (d[7:0]). These flip-flops are synchronized with the pixel clock and controlled by the href signal, ensuring accurate data capture.

Additionally, a debounce_better_version module processes the shutter_raw input to generate a clean shutter signal, mitigating the effects of mechanical switch bouncing. Debouncing the shutter signal is key to successful digital logic, as any bouncing in the shutter press could interfere with downstream logic. Our solution for shutter debouncing

is described below, and requires four verilog modules[2]:

Unset

debounce_better_version.sv

```
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects
// Verilog code for button debouncing on FPGA
// debouncing module without creating another clock domain
// by using clock enable signal
module debounce_better_version(input pb_1,clk,output pb_out);
wire slow_clk_en;
wire Q1,Q2,Q2_bar,Q0;
clock_enable u1(clk,slow_clk_en);

my_dff_en d0(clk,slow_clk_en,pb_1,Q0);


my_dff_en d1(clk,slow_clk_en,Q0,Q1);
my_dff_en d2(clk,slow_clk_en,Q1,Q2);
assign Q2_bar = ~Q2;
//assign pb_out = Q1 & Q2_bar;
assign pb_out = Q1 & Q2_bar & slow_clk_en;

endmodule
```

Unset

flipflop.sv

```
//async reset flip flop module
module flipflip(clk, rst, en, d, q);
    input logic clk;
    input logic rst;
    input logic en;
    input logic d;
    output logic q;

    always_ff@(posedge clk or posedge rst) begin
        if (rst)
            q <= 1'b0;
        else if (en) begin
            q <= d;
```

[2] Verilog code for debouncing buttons on FPGA - FPGA4student.com

```
        end
    end
endmodule
```

Unset

clock_enable.sv

```
// Slow clock enable for debouncing button
module clock_enable(input Clk,output slow_clk_en);
    reg [26:0]counter=0;
    always @(posedge Clk_100M)
    begin
      counter <= (counter>=249999)?0:counter+1;
    end

    assign slow_clk_en = (counter == 249999)?1'b1:1'b0;
endmodule
*/

// Generates a 1-clock-cycle pulse every x clock cycles
module clock_enable (
    input  logic clk,
    output logic slow_clk_en
);
    reg [24:0] counter = 0; // 4-bit counter is enough for values 0–9
    always_ff @(posedge clk) begin
        if (counter == 25000000)
            counter <= 0;
        else
            counter <= counter + 1;
    end
    assign slow_clk_en = (counter == 25000000);
endmodule
```

Unset

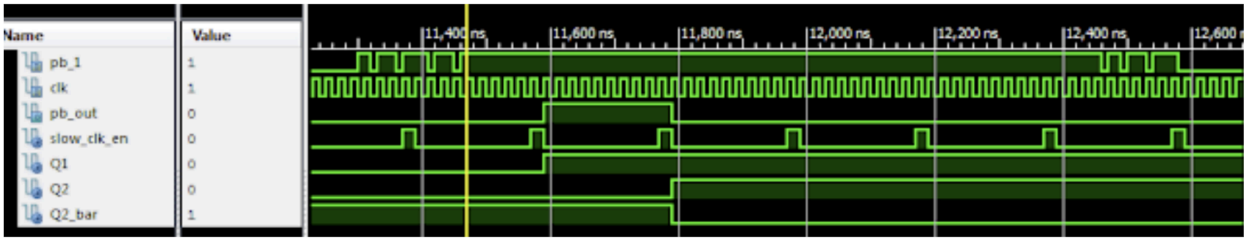
flipflip.sv

```
//async reset flip flop module
module flipflip(clk, rst, en, d, q);
```

```
    input logic clk;
    input logic rst;
    input logic en;
    input logic d;
    output logic q;

    always_ff@(posedge clk or posedge rst) begin
        if (rst)
            q <= 1'b0;
        else if (en) begin
            q <= d;
        end
    end
endmodule
```

The timing diagram representing the functionality of the debouncing is also shown:



Mechanical switches, such as the shutter button, are prone to signal bouncing, which can lead to multiple unintended triggers. To mitigate this, the module incorporates a debounce_better_version submodule that processes the raw shutter_raw input to generate a clean shutter signal.

The debouncing mechanism operates by generating a slower clock enable signal (slow_clk_en) with a heavily unbalanced duty cycle, high for only one clock period and low for the remaining cycles. This design ensures that the likelihood of sampling the bouncing signal during its unstable phase is minimal. Once a stable high signal is detected during the slow_clk_en high phase, the shutter signal is latched and processed through additional flip-flops to produce a single-cycle pulse, synchronized with the main clock.

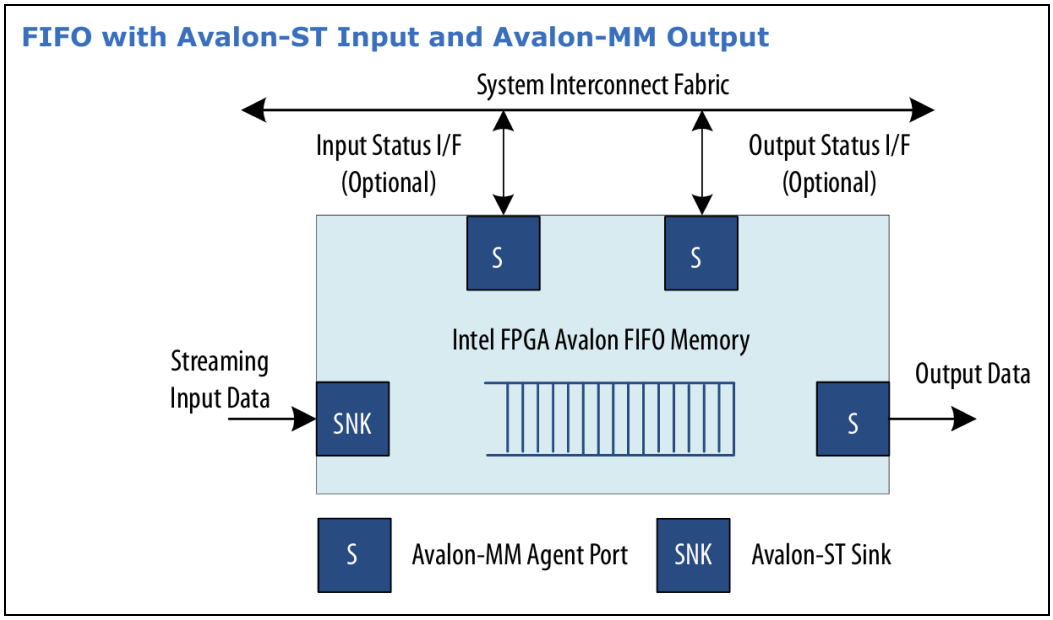
For instance, assuming a 50 MHz main clock, the slow_clk_en can be configured to be high for one cycle every second (i.e., high for 1 cycle and low for 49,999,999 cycles). This configuration requires the user to press and hold the shutter button for at least one second to ensure reliable detection.

The camera_interface.sv module effectively captures and processes pixel data from a CMOS image sensor, organizing it into 32-bit words suitable for further digital processing or storage. Its design addresses key challenges, including synchronization of incoming data, efficient data aggregation for FIFO buffering, and reliable detection of mechanical shutter inputs through robust debouncing techniques.

FIFO Avalon® -ST Sink to Avalon® -MM Read Agent

After the camera_module.sv file, the data gets passed into a FIFO which is pre-configured by Platform Designer. Intel gives four different possible configurations for the FIFO, depending on if you want the input to be serial or memory mapped. If the interfaces should be serial, the interface is labeled to be either “ST Sinks” or “ST Source”. If the interface is memory mapped, the interface is labeled to be either “MM Read Agent” or “MM Write Agent”. For our implementation, we want the input of the FIFO to be a serial because the data is being continuously received from the camera and it needs to be stored in such a way that the output can be read in a calm and controlled fashion. The output of the FIFO, therefore, is memory mapped because we want control of reading data out of the FIFO.

In this configuration, according to the Intel documentation, the only allowable interface width between the input and the output of the FIFO is 32-bit. After exiting the camera_interface.sv module, the 32-bit wide data is streamed through the Avalon-ST sink interface. The FIFO core performs endian conversion as necessary to align with the output interface protocol. As the input data is sunk into the FIFO, it gets read to memory mapped registers. The mapped registers remain in place until read enable configuration is set and the information can be read out of the FIFO. There are many configurable settings when generating the FIFO instance in Platform Designer including backpressure, ready, valid, and wait request. The signal most pertinent to our project would be “backpressure” as it is the signal which notifies the interface on the output of the FIFO that the FIFO contains information. Although this signal could be helpful, our group chose to implement a timed inquiry to the FIFO to check for data instead of the backpressure pin. Therefore, our FIFO is rather bare bones. The diagram for the Sink-MM FIFO from the Intel data sheet is shown below.



The datasheet also includes helpful information about how the output of the FIFO can be interacted with. The use of separate register maps allows for flexibility in how each bus interacts with the FIFO, potentially enabling different data widths, access protocols, or control mechanisms. At a high level, there are two buses interacting with the FIFO, both with their own register map, meaning the base address of both buses can be considered zero.

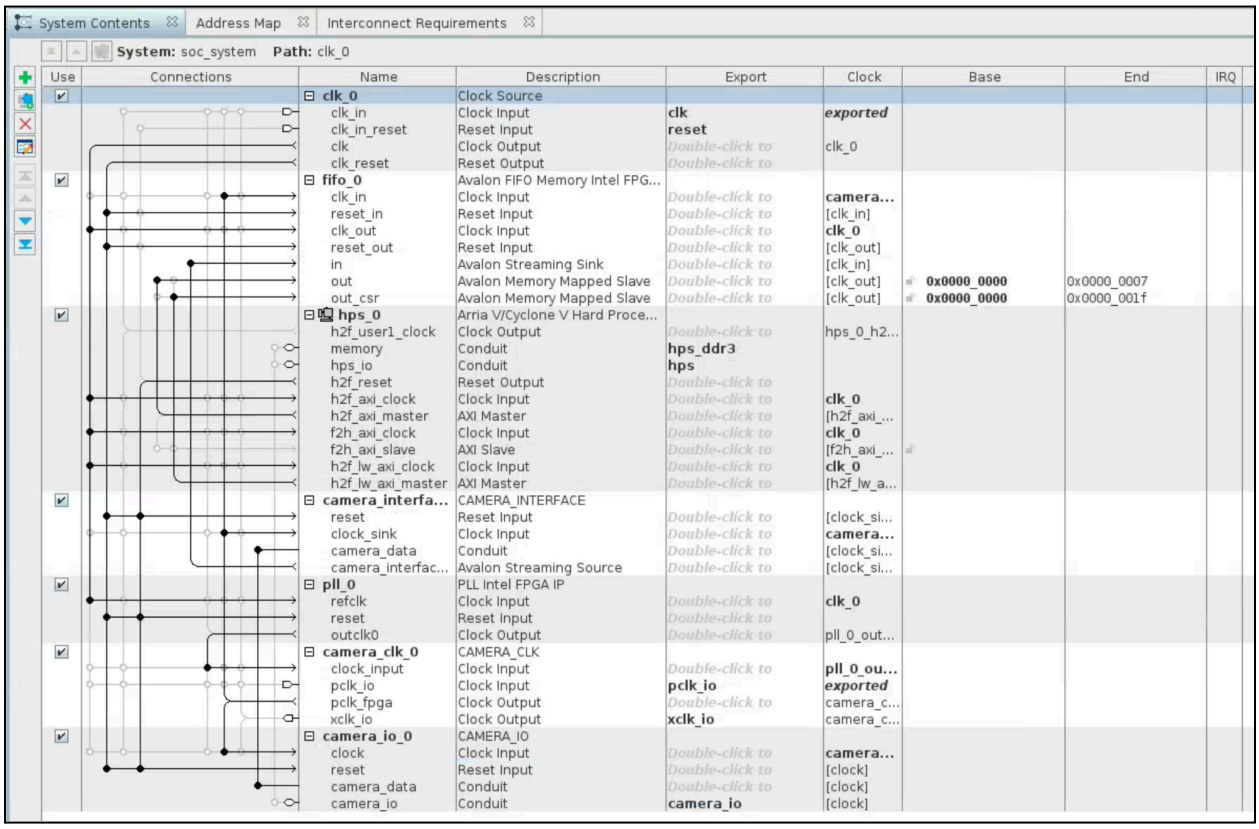
As shown in the Platform Designer GUI and documentation, the output buses for the data and status registers of the FIFO have specific locations and configurations. A screenshot of the fully connected Platform Designer hardware for our project is shown below[3]. The signal labeled “out_cpr” is six, 32 bit registers which are mapped to the following signals:

offset	31	24	23	16	15	8	7	6	5	4	3	2	1	0
base	fill_level													
base + 1									i_status					
base + 2									event					
base + 3									interrupt enable					
base + 4	almostfull													
base + 5	almostempty													

[3] 24.2.4. Avalon® -ST Sink to Avalon® -MM Read Agent

As shown, the lowest 6 bits of the base+1 register contain status signals about the FIFO MM output, bit one of which signals the EMPTY bit.

Bit(s)	Name	Description
0	FULL	Has a value of 1 if the FIFO is currently full.
1	EMPTY	Has a value of 1 if the FIFO is currently empty.
2	ALMOSTFULL	Has a value of 1 if the fill level of the FIFO is equal or greater than the `almostfull` value.
3	ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO is less or equal than the `almostempty` value.
4	OVERFLOW	Is set to 1 for 1 cycle every time the FIFO overflows. The FIFO overflows when an Avalon® write host writes to a full FIFO. OVERFLOW is only valid when **Allow backpressure** is off.
5	UNDERFLOW	Is set to 1 for 1 cycle every time the FIFO underflows. The FIFO underflows when an Avalon® read host reads from an empty FIFO. UNDERFLOW is only valid when **Allow backpressure** is off.



From the EMPTY and DATA buses, data can be read out of the FIFO. Key to reading the data, however, is the device driver. The device driver will be discussed in detail in the next session.

Device Driver (*camera.ko*)

The camera device driver is implemented as a Linux kernel module that interfaces with a scanline camera over an Avalon-MM FIFO interface. It exposes a character device

(/dev/camera) and provides two ioctl-based operations: CAMERA_READ_WORD, which returns a 32-bit word containing two packed RGB565 pixels, and CAMERA_FIFO_EMPTY, which reports whether the FIFO is currently empty. Internally, the driver maps the FIFO data register at offset 0x000 and the status register (i_status) at offset 0x004, where the EMPTY condition is indicated by bit 1. The driver uses the ioread32() interface to access hardware registers and translates low-level FIFO status into simple integer responses for userspace. By delegating polling and control logic to userspace, the driver maintains a lightweight, low-overhead design while enabling reliable pixel stream access for real-time barcode decoding applications.

Address 0x000 : FIFO_DATA (32-bit)

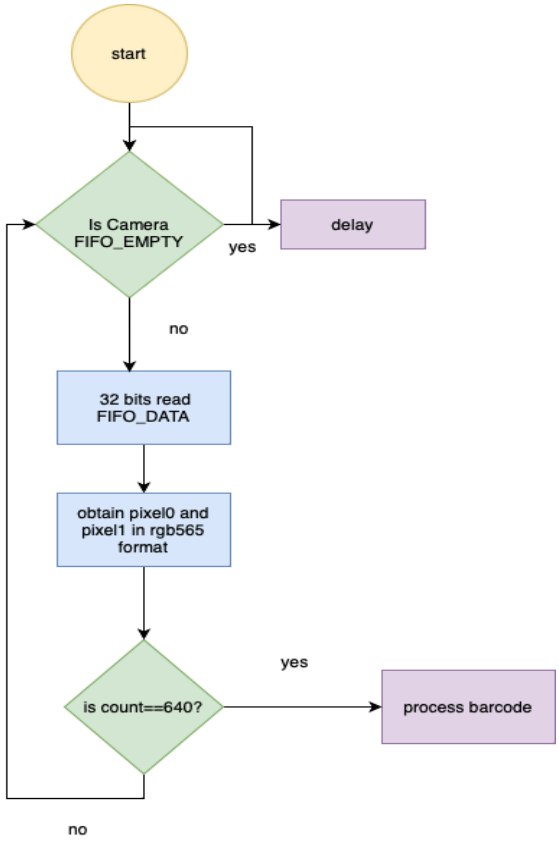
31-27	26-21	20-16	15-11	10-5	0-4
R1	G1	B1	R0	G0	B0

Address 0x004 : FIFO_STATUS (32-bit)

...	5	4	3	2	1	0
STUFF WE DONT NEED	UNDERFLOW	OVERFLOW	ALMOST EMPTY	ALMOST FULL	EMPTY	FULL

Controller Loop

The userspace controller, implemented in `read_scanline.c`, is responsible for retrieving a stream of RGB565 pixels from the camera driver and passing them to a barcode decoding routine. It opens the `/dev/camera` device and polls the FIFO by repeatedly invoking the `CAMERA_FIFO_EMPTY` ioctl until data becomes available. Once ready, the program reads 32-bit words from the FIFO using the `CAMERA_READ_WORD` ioctl. Each word contains two 16-bit RGB565 pixels, which are unpacked and stored in a structured array using the `rgb565_t` type. This struct uses bit fields to represent red, green, and blue components as `uint8_t r:5`, `g:6`, and `b:5` respectively, matching the hardware pixel format. After collecting a complete scanline of 640 pixels, the controller calls `process_barcode()`, which decodes the image into a 12-digit UPC-A code if possible. This modular design separates low-level polling and pixel unpacking from high-level barcode recognition, enabling clarity and reusability.



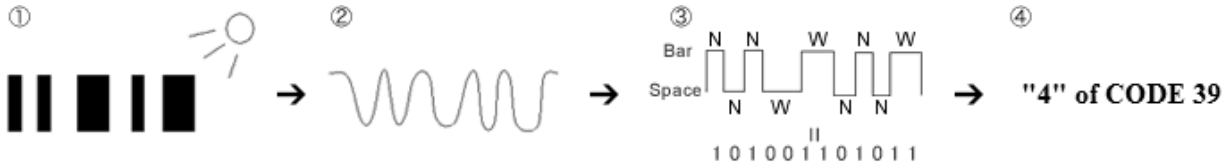
This struct uses bit fields to represent red, green, and blue components as `uint8_t r:5`, `g:6`, and `b:5` respectively, matching the hardware pixel format. After collecting a complete scanline of 640 pixels, the controller calls `process_barcode()`, which decodes the image into a 12-digit UPC-A code if possible. This modular design separates low-level polling and pixel unpacking from high-level barcode recognition, enabling clarity and reusability.

```C/C++
typedef struct {
       uint8_t r : 5;
       uint8_t g : 6;
       uint8_t b : 5;
} rgb565_t;

char *process_barcode(rgb565_t *pixels, int len);
```

Processing Algorithm

The barcode_decoder module implements a complete software pipeline to extract UPC-A barcodes from a horizontal scanline of RGB565 pixels. It begins by converting each pixel to grayscale using a weighted sum of red, green, and blue values, then applies adaptive thresholding to binarize the scanline based on local brightness. The resulting binary sequence is run-length encoded to measure consecutive bar and space widths, which are normalized to unit widths based on the expected barcode structure. The decoder then searches for UPC-A guard patterns and splits the normalized data into digit-encoded segments, using separate lookup tables for left- and right-hand digits. A final checksum is computed and compared against the trailing digit for validation. If successful, the function returns a null-terminated string containing the 12-digit UPC-A code; otherwise, it returns NULL. This modular pipeline is robust to noise and lighting variation, and integrates cleanly with the scanline controller.



(Barcode processing stages[4])

Configure Camera Settings (SCCB)

To configure the camera's onboard memory, an SCCB interface is used. It includes serial clock and serial data pins, and has a protocol very similar to I2C. There were two options to implement register configuration for our system: create FPGA hardware to communicate over I2C or configure the HPS to communicate over I2C.

Creating FPGA hardware was a safer option due to the known process of configuring GPIO pins to interface with the hardware. However, it presented a large amount of complexity in either recreating the two-way ACK-based communication of I2C in System Verilog. On the other hand, connecting an I2C device to the HPS was a risky option since there are no known examples of this for the DE1-SoC board. Still, writing user-level C-code to interface with the standard <linux/gpio.h> library for I2C communications was far easier to implement and faster to iterate on than writing a custom hardware design in System Verilog. For this reason we chose to use the HPS I2C interface.

The Cyclone-V SoC HPS uses an IP Block from Synopsys as an I2C interface. This IP Block contains 4 I2C interfaces, each addressable from the configuration files. Only 2 of these interfaces are used on the DE1-SoC board and only one is exposed to a physical header.

[4] https://www.denso-wave.com/en/adcd/fundamental/barcode/scan/index.html

The (HPS_I2C1 / i2c-0) bus is on-board only. It goes to a mux which allows either the HPS or FPGA to access it but only goes to on-chip components. The (HPS_I2C2 / i2c-1) bus is connected to the LTC Connector and designed to be used with a Linear Technologies expansion cards. This second bus is disabled by default. To enable it, we modified the system configuration:

soc_system_board_info.xml

```
Unset
<!-- Before -->
<DTAppend name="status" type="string" parentlabel="hps_0_i2c1" val="disabled"/>

<!-- After -->
<DTAppend name="speed-mode" type="number" parentlabel="hps_0_i2c1" val="0"/>
```

The next step was running **`make dtb`** to rebuild the device tree. This also outputs a human readable device tree:

soc_system.dts

```
Unset
/* Before */
hps_0_i2c1: i2c@0xffc05000 {
              compatible = "snps,designware-i2c-21.1", "snps,designware-i2c";
              reg = <0xffc05000 0x00000100>;
              interrupt-parent = <&hps_0_arm_gic_0>;
              interrupts = <0 159 4>;
              clocks = <&l4_sp_clk>;
              emptyfifo_hold_master = <1>;      /* embeddedsw.dts.params.... */
              status = "disabled";       /* appended from boardinfo */
       }; //end i2c@0xffc05000 (hps_0_i2c1)

/* After */
hps_0_i2c1: i2c@0xffc05000 {
              compatible = "snps,designware-i2c-21.1", "snps,designware-i2c";
              reg = <0xffc05000 0x00000100>;
              interrupt-parent = <&hps_0_arm_gic_0>;
              interrupts = <0 159 4>;
              clocks = <&l4_sp_clk>;
              emptyfifo_hold_master = <1>;      /* embeddedsw.dts.params.... */
              status = "okay";    /* embeddedsw.dts.params.status type STRING */
              speed-mode = <0>;   /* appended from boardinfo */
       }; //end i2c@0xffc05000 (hps_0_i2c1)
```

Finally, the device tree (***soc_system.dtb***) could be installed in the usual way by moving it to **`/dev/mmcblk0p1`** and rebooting.

To verify that the second i2c bus is active and loaded, one can run **`cat /proc/iomem.`**
If both i2c buses are in operation, two entries will be present

`i2c@0xffc04000`
`i2c@0xffc05000`

To verify that the camera is connected and operational, one can install **`i2c-tools`** .
Run **`dpkg -L i2c-tools`** to verify its installation.
Four new programs should appear

`/usr/sbin/i2cdump`
`/usr/sbin/i2cdetect`
`/usr/sbin/i2cget`
`/usr/sbin/i2cset`

Running **`i2cdetect -y -r 1`** will display all devices on the second i2c bus.

Optical Effects

To properly instruct users of the barcode scanning system, we plan to experiment with and improve the limitations of the device as we iterate the image processing parameters. This way, although there may remain significant limitations regarding the flexibility of the device usability, we will create the most usable product possible. We have determined the key limitations that will guide our design process and affect the usability of the scanner.

Distance

How far can the user be from the barcode? This limitation is derived from the resolution of the camera and its ability to distinguish narrow and wide bars. We will decide an acceptable number of pixels for a narrow bar based on testing (likely no less than 2-3 since the pixel borders will never be perfectly aligned with the barcode stripes). The maximum distance will assume that the camera is aligned horizontally to the barcode since that is the worst case (smallest bar width). Once, upon our testing, the camera consistently misidentifies the barcode, we will know we are too far.

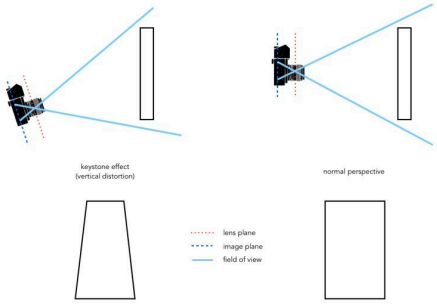
Similarly, we need to determine how close the camera can be to the barcode? This limitation is based on the camera's minimum focus distance and field of view.

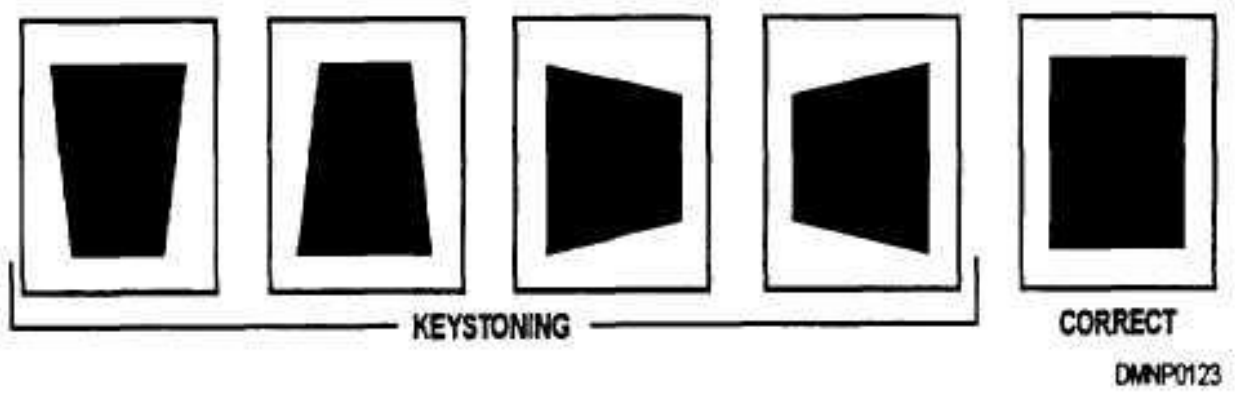
Alignment (Camera Orientation)

There are three angles which define the alignment of the camera to the barcode: pitch, yaw, and roll. For our purposes, pitch and yaw together can be classified as “angle of incidence”, representing how aligned the camera sensor plane and barcode plane are. Roll

Angle of Incidence

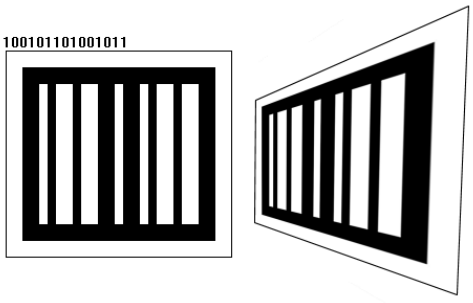
Pitch and yaw represent the alignment of the camera sensor plane to the plane of the barcode. If the camera is normal to the surface, the angle of incidence is 0°. The more oblique (misaligned) the camera is from the surface, the smaller the projected height and width of the barcode. However, while the visual size of the barcode shrinks as it is projected at a high angle of incidence, the two axes have different effects.





Since barcodes are horizontally symmetric, the pitch of the camera does not alter the processing, since selecting a single pixel row will mitigate keystoning on the horizontal axis. The only limitation imposed by camera pitch is that the center row of pixels remains within the bounds of the barcode. This equates to the user moving the scanner up and down to align the center row of pixels within the height of the barcode.

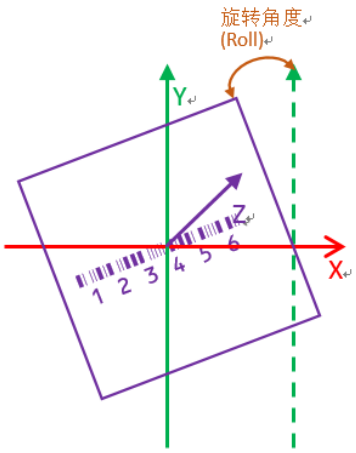
On the other hand, since barcodes encode data along their horizontal axis, the yaw of the camera causes meaningful keystoning of the bars. Keystoning means that the closer side of the barcode will appear larger than the farther side. This equates to a non-uniform stretching of the bar widths. This effect must be addressed in the software algorithm. Our plan is to use the left and right barcode identifiers to calibrate the most and least stretched sides of the barcode, then linearly interpolate between the two scales as we parse narrow and wide bars in the middle.



For angle of incidence tolerance, ±72° has been achieved in commercial scanners[5] in both tilt and skew angle.

Roll

The camera roll is similar to the pitch since it does not require a software algorithm to account for. Since we will be selecting a single row of horizontal pixels from the camera, the user will be responsible for orienting the camera to the correct roll. However, this does not mean that the camera must have 0° of roll. As long as the full width of the barcode is intersected by the X axis of the camera, the data will be read. The amount of roll allowed by the camera is dependent on the height of the barcode. A taller barcode would allow for more roll while still scanning. The one additional consideration with roll is that the horizontal sample of the barcode would stretch as the roll increases. This is already accounted for in the algorithm.



A 360° roll tolerance has been achieved in commercial scanners[6].

Scene Brightness

The camera will have auto exposure, however scenes which are too dim or bright will compress the dynamic range of the image. To solve this we will experiment with using an LED above the camera to enable a constant brightness in more lighting conditions.

Image Noise

The image will have natural noise due to thermal and electrical effects on the image sensor, especially in dim conditions. Raising the brightness with an LED may solve this problem if the noise is strong enough to affect the white/black thresholds for bar colors.

Background

The barcode will not always be on a pure white background. The patterns to the left and right of the barcode may resemble the white/black patterns of the bars. To allow the system to recognize the barcode itself, the EAN-13 specification implements standard “guards” on the left and right sides of the encoded barcode data. These consistent patterns allow the processing system to identify the barcode location.

[5] https://www.lmppos.com/product/2D-Wireless-Barcode-Scanner.html
[6] https://www.lmppos.com/product/2D-Wireless-Barcode-Scanner.html

Full Code

GitHub Repo https://github.com/matthew-modi/embedded-project.git

camera.c

```
Unset
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include "camera.h"

// Define locally if not using HAL includes
#define ALTERA_AVALON_FIFO_DATA_REG           0
#define ALTERA_AVALON_FIFO_STATUS_REG         1
#define ALTERA_AVALON_FIFO_STATUS_EMPTY_MASK  (1 << 1)

#define SCANLINE_OFFSET      (ALTERA_AVALON_FIFO_DATA_REG * 4)
#define FIFO_EMPTY_OFFSET    (ALTERA_AVALON_FIFO_STATUS_REG * 4)
#define DRIVER_NAME "camera"

// #define SCANLINE_OFFSET       0x000  // FIFO read port
// #define FIFO_EMPTY_OFFSET     0x004  // FIFO istatus

struct camera_dev {
    struct resource res;
    void __iomem *virtbase;
    void __iomem *scanline_base;
    void __iomem *fifo_empty_base;
} dev;

static long camera_ioctl(struct file *file, unsigned int cmd, unsigned long
arg)
{
    switch (cmd) {
    case CAMERA_READ_WORD: {
        u32 word = ioread32(dev.scanline_base);
```

```
        if (copy_to_user((u32 *)arg, &word, sizeof(u32)))
            return -EFAULT;
        break;
    }

    case CAMERA_FIFO_EMPTY: {
        u32 status = ioread32(dev.fifo_empty_base);
        int empty = (status >> 1) & 0x1;  // Bit 1 is EMPTY flag
        if (copy_to_user((int *)arg, &empty, sizeof(int)))
            return -EFAULT;
        break;
    }

    default:
        return -EINVAL;
    }

    return 0;
}

static const struct file_operations camera_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = camera_ioctl,
};

static struct miscdevice camera_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DRIVER_NAME,
    .fops  = &camera_fops,
};

static int __init camera_probe(struct platform_device *pdev)
{
    int ret;

    ret = misc_register(&camera_misc_device);
    if (ret)
        return ret;

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret)
        goto fail_deregister;
```

```
    if (!request_mem_region(dev.res.start, resource_size(&dev.res),
DRIVER_NAME)) {
        ret = -EBUSY;
        goto fail_deregister;
    }

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (!dev.virtbase) {
        ret = -ENOMEM;
        goto fail_release;
    }

    dev.scanline_base = dev.virtbase + SCANLINE_OFFSET;
    dev.fifo_empty_base = dev.virtbase + FIFO_EMPTY_OFFSET;

    pr_info(DRIVER_NAME ": probe successful\n");
    return 0;

fail_release:
    release_mem_region(dev.res.start, resource_size(&dev.res));
fail_deregister:
    misc_deregister(&camera_misc_device);
    return ret;
}

static int camera_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&camera_misc_device);
    return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id camera_of_match[] = {
    { .compatible = "csee4840,camera-1.0" },  /* your custom node */
    { .compatible = "ALTR,fifo-21.1" },       /* default FIFO core */
    { .compatible = "ALTR,fifo-1.0" },        /* fallback older format */
    { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, camera_of_match);
#endif

static struct platform_driver camera_driver = {
```

```
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(camera_of_match),
    },
    .remove = __exit_p(camera_remove),
};

static int __init camera_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&camera_driver, camera_probe);
}

static void __exit camera_exit(void)
{
    platform_driver_unregister(&camera_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(camera_init);
module_exit(camera_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ananya Haritsa, Columbia University");
MODULE_DESCRIPTION("Camera driver for polling and stream reading");
```

camera.h

```
C/C++
#ifndef _CAMERA_H
#define _CAMERA_H

#include <linux/ioctl.h>
#include <linux/types.h>

#define CAMERA_MAGIC 'q'

#define CAMERA_READ_WORD    _IOR(CAMERA_MAGIC, 1, u32 *)
#define CAMERA_FIFO_EMPTY   _IOR(CAMERA_MAGIC, 2, int *)

#endif
```

read_scanline.c

```C/C++
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
#include "camera.h"
#include "barcode_decoder.h"

#define DEVICE_PATH   "/dev/camera"
#define POLL_DELAY_US 1000    // 1 ms between polls
#define PIXEL_COUNT   640     // Number of 16-bit RGB565 pixels to collect

// Must match the bit-field in barcode_decoder.h
typedef struct {
    uint8_t b : 5;
    uint8_t g : 6;
    uint8_t r : 5;
} rgb565_t;

int main(void) {
    int fd, count = 0;
    rgb565_t pixels[PIXEL_COUNT];

    // Open camera device
    fd = open(DEVICE_PATH, O_RDONLY);
    if (fd < 0) {
        perror("Failed to open /dev/camera");
        return EXIT_FAILURE;
    }

    // Wait until FIFO has data
    printf("Waiting for FIFO to have data…\n");
    int empty = 1;
```

```
do {
    if (ioctl(fd, CAMERA_FIFO_EMPTY, &empty) < 0) {
        perror("ioctl CAMERA_FIFO_EMPTY failed");
        close(fd);
        return EXIT_FAILURE;
    }
    usleep(POLL_DELAY_US);
} while (empty);

printf("FIFO has data. Reading scanline…\n");

// Read until we've collected 640 RGB565 pixels
while (count < PIXEL_COUNT) {
    uint32_t word;
    if (ioctl(fd, CAMERA_READ_WORD, &word) < 0) {
        perror("ioctl CAMERA_READ_WORD failed");
        close(fd);
        return EXIT_FAILURE;
    }

    // lower 16 bits
    uint16_t p0 = word & 0xFFFF;
    pixels[count].r = (p0 >> 11) & 0x1F;
    pixels[count].g = (p0 >> 5)  & 0x3F;
    pixels[count].b =  p0        & 0x1F;
    count++;

    // upper 16 bits (if still room)
    if (count < PIXEL_COUNT) {
        uint16_t p1 = word >> 16;
        pixels[count].r = (p1 >> 11) & 0x1F;
        pixels[count].g = (p1 >> 5)  & 0x3F;
        pixels[count].b =  p1        & 0x1F;
        count++;
    }
}

close(fd);

// Process the RGB565 scanline—returns a malloc'd 13-byte string or NULL
char *upc = process_barcode(pixels, PIXEL_COUNT);
if (upc) {
    printf("Decoded UPC-A: %s\n", upc);
    free(upc);
```

```
    } else {
        printf("Failed to decode barcode\n");
    }

    return EXIT_SUCCESS;
}
```

barcode_decoder.c

```C/C++
#include "barcode_decoder.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PIXEL_COUNT 640
#define MAX_BITS     1024

// (Keep your L_CODES, R_CODES, hamming_distance(), upca_checksum(),
//  and decode_upca() exactly as before.)

static uint8_t rgb565_to_gray(rgb565_t p) {
    // Expand bitfields back into full 8-bit channels
    uint8_t r8 = (p.r << 3) | (p.r >> 2);
    uint8_t g8 = (p.g << 2) | (p.g >> 4);
    uint8_t b8 = (p.b << 3) | (p.b >> 2);
    // Weighted grayscale
    return (77 * r8 + 150 * g8 + 29 * b8) >> 8;
}

char *process_barcode(rgb565_t *pixels, int len) {
    if (len < PIXEL_COUNT) return NULL;

    // 1) Convert to grayscale
    uint8_t scanline[PIXEL_COUNT];
    for (int i = 0; i < PIXEL_COUNT; i++)
        scanline[i] = rgb565_to_gray(pixels[i]);

    // 2) Smooth (5-point box)
    uint8_t smoothed[PIXEL_COUNT];
    for (int i = 0; i < PIXEL_COUNT; i++) {
        int sum = 0, cnt = 0;
        for (int j = -2; j <= 2; j++) {
```

```
            int idx = i + j;
            if (idx >= 0 && idx < PIXEL_COUNT) {
                sum += scanline[idx];
                cnt++;
            }
        }
        smoothed[i] = sum / cnt;
    }

    // 3) Threshold to binary
    uint8_t binary[PIXEL_COUNT];
    int mn = 255, mx = 0;
    for (int i = 0; i < PIXEL_COUNT; i++) {
        if (smoothed[i] < mn) mn = smoothed[i];
        if (smoothed[i] > mx) mx = smoothed[i];
    }
    int thr = (mn + mx) >> 1;
    for (int i = 0; i < PIXEL_COUNT; i++)
        binary[i] = (smoothed[i] < thr);

    // 4) RLE
    int rle[MAX_BITS], rle_len = 0;
    uint8_t cur = binary[0];
    int cnt = 1;
    for (int i = 1; i < PIXEL_COUNT; i++) {
        if (binary[i] == cur) cnt++;
        else {
            rle[rle_len++] = cnt;
            cnt = 1; cur = binary[i];
        }
    }
    rle[rle_len++] = cnt;

    // 5) Module width via median of first 20
    int sorted[20], runs = rle_len < 20 ? rle_len : 20;
    memcpy(sorted, rle, runs * sizeof(int));
    for (int i = 0; i < runs-1; i++)
      for (int j = i+1; j < runs; j++)
        if (sorted[j] < sorted[i]) {
          int t = sorted[i]; sorted[i] = sorted[j]; sorted[j] = t;
        }
    int mw = sorted[runs/2];
    if (mw < 1) mw = 1; if (mw > 10) mw = 10;
```

```
    // 6) Build bitstream
    char bs[MAX_BITS];
    int bs_len = 0;
    cur = binary[0];
    for (int i = 0; i < rle_len; i++) {
        int w = rle[i] / mw;
        if (w < 1) w = 1;
        for (int k = 0; k < w; k++)
            bs[bs_len++] = cur + '0';
        cur = 1 - cur;
    }
    bs[bs_len] = '\0';
    if (bs_len < 95) return NULL;

    // 7) Align to best 95-bit window
    int best_s = -1, best_hd = 1e9;
    for (int s = 0; s <= bs_len - 95; s++) {
        int hd = hamming_distance(  bs + s,     "101",   3)
               + hamming_distance(bs + s+45, "01010", 5)
               + hamming_distance(bs + s+92, "101",   3);
        if (hd < best_hd) { best_hd = hd; best_s = s; if (!hd) break; }
    }
    if (best_s < 0) return NULL;

    // 8) Decode
    char *digits = malloc(13);
    float confs[12];
    if (!digits) return NULL;

    if (decode_upca(bs + best_s, digits, confs) != 0) {
        free(digits);
        return NULL;
    }

    // Optionally check checksum here...
    return digits;
}
```

barcode_decoder.h

```
C/C++
#ifndef BARCODE_DECODER_H
```

```
#define BARCODE_DECODER_H

#include <stdint.h>

typedef struct {
    uint8_t b : 5;
    uint8_t g : 6;
    uint8_t r : 5;
} rgb565_t;

// Returns a malloc'd 13-byte string (12 digits + '\0'), or NULL on failure.
// Caller must free() it.
char *process_barcode(rgb565_t *pixels, int len);

#endif // BARCODE_DECODER_H
```

Camera_interface.sv

```
Unset
module camera_interface (
       //mock input
       input logic pclk,
       input reset,

       //camera inputs
       input logic href,
       input logic vsync,
       input logic [7:0] d,

       //other inputs
       input logic shutter_raw, //assume shutter is active low

       //outputs
       output logic fifo_enable,
       output logic [31:0] wide_bit_out
       );

       typedef enum logic [1:0] {
           RESET,
           SHUTTER,
           WRITE,
```

```
    BLOCK
} state_t;

state_t state;

logic [10:0] col_count;
logic [8:0] row_count;
logic [2:0] clk_count;
logic [7:0] q;

logic shutter;
logic write_enable;
logic prev_vsync;
logic curr_vsync;
logic write_trigger;
logic rst;

assign curr_vsync = vsync;

assign write_enable = (href && (row_count == 239)) ? 1 : 0;

assign write_trigger = write_enable;

always_ff @(posedge pclk) begin

      if (reset) begin
            state      <= RESET;
            col_count  <= 0;
            row_count  <= 0;
            prev_vsync <= 0;
            rst <= 1;
      end else begin
            if ((shutter) && ((row_count < 1) || (row_count > 240)))
                  state <= RESET;
            else case (state)
                  RESET : begin
                        if (vsync) begin
                              col_count <= 11'b0;
                              row_count <= 9'b0;
                              state <= SHUTTER;
                              rst <= 1;
                        end
                  end
                  SHUTTER : begin
```

```
                                rst <= 0;
                                if ((col_count == 11'd0) && (href))
                                        row_count <= row_count + 1;
                                if ((col_count < 11'd1279) && (href))
                                        col_count <= col_count + 1;
                                if ((col_count == 11'd1279) && (href)) begin
                                        col_count <= 0;
                                end
                                if ((write_trigger) && (href))
                                        state <= WRITE;
                        end
                        WRITE : begin
                                if ((col_count < 11'd1279) && (href))
                                        col_count <= col_count + 1;
                                if ((col_count == 11'd1279) && (href)) begin
                                        col_count <= 0;
                                        row_count <= row_count + 1;
                                end
                                if ((row_count==240) && (href))
                                        state <= BLOCK;
                        end
                        BLOCK : begin
                                prev_vsync <= curr_vsync;
                        end
                        default: state <= BLOCK;
                endcase
        end
end

always_ff @(posedge pclk) begin
        if ((write_enable) && (clk_count == 3'd3) && (row_count == 239))
begin
                clk_count <= 3'd0;
        end else if ((clk_count < 3'd4) && (write_enable)) begin
                clk_count <= clk_count + 1;
        end
end

always_ff @(negedge pclk) begin

        if ((write_enable) && (clk_count == 3'd0)) begin
                wide_bit_out[15:8] <= q[7:0];
                fifo_enable <= 0;
        end
```

```
            else if ((write_enable) && (clk_count == 3'd1)) begin
                  wide_bit_out[7:0] <= q[7:0];
                  fifo_enable <= 0;
            end
            else if ((write_enable) && (clk_count == 3'd2)) begin
                  wide_bit_out[31:24] <= q[7:0];
                  fifo_enable <= 0;
            end
            else if ((write_enable) && (clk_count == 3'd3)) begin
                  wide_bit_out[23:16] <= q[7:0];
                  fifo_enable <= 1;
            end
      end

      flipflip uut0 (.clk(pclk), .rst(rst), .en(href), .d(d[0]), .q(q[0]));
      flipflip uut1 (.clk(pclk), .rst(rst), .en(href), .d(d[1]), .q(q[1]));
      flipflip uut2 (.clk(pclk), .rst(rst), .en(href), .d(d[2]), .q(q[2]));
      flipflip uut3 (.clk(pclk), .rst(rst), .en(href), .d(d[3]), .q(q[3]));
      flipflip uut4 (.clk(pclk), .rst(rst), .en(href), .d(d[4]), .q(q[4]));
      flipflip uut5 (.clk(pclk), .rst(rst), .en(href), .d(d[5]), .q(q[5]));
      flipflip uut6 (.clk(pclk), .rst(rst), .en(href), .d(d[6]), .q(q[6]));
      flipflip uut7 (.clk(pclk), .rst(rst), .en(href), .d(d[7]), .q(q[7]));

      debounce_better_version uut8(.pb_1(shutter_raw), .clk(pclk),
.pb_out(shutter));


endmodule
```