

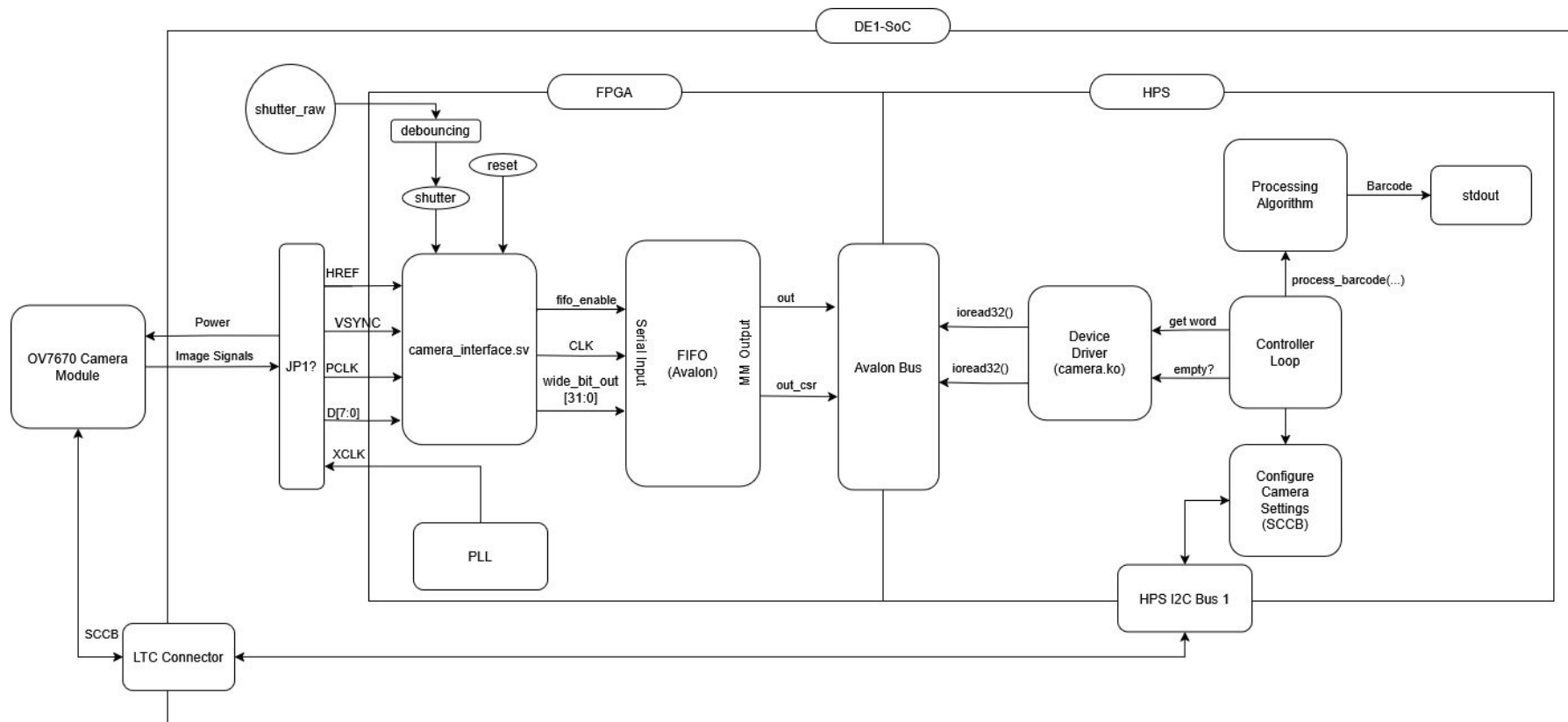
EAN-13 Barcode Decoder with FPGA and VGA Display

Matthew Modi, Helen Bovington, Ananya Haritsa,
Rahul Pulidindi, Kamil Zajkowski

Background

- **Purpose:** Read EAN-13 Barcode via OV7670 camera → decode on HPS → display GTIN on VGA
- Leverages EAN-12/GS1-US standard used on billions of retail items
- Demonstrates true HW/SW design: FPGA for real-time pixel capture + CPU for image decoding
- Low-cost prototype (OV7670 + De1-SoC) that mimics commercial barcode scanners
- **High-level Flow:** Camera → FGPA Fabric → HPS Software → VGA

System Block Diagram



SCCB

Demo!

SCCB is the bus we use to configure the camera's internal memory (settings).

We use a userspace program to write data to the second I2C bus on the DE1-SoC.

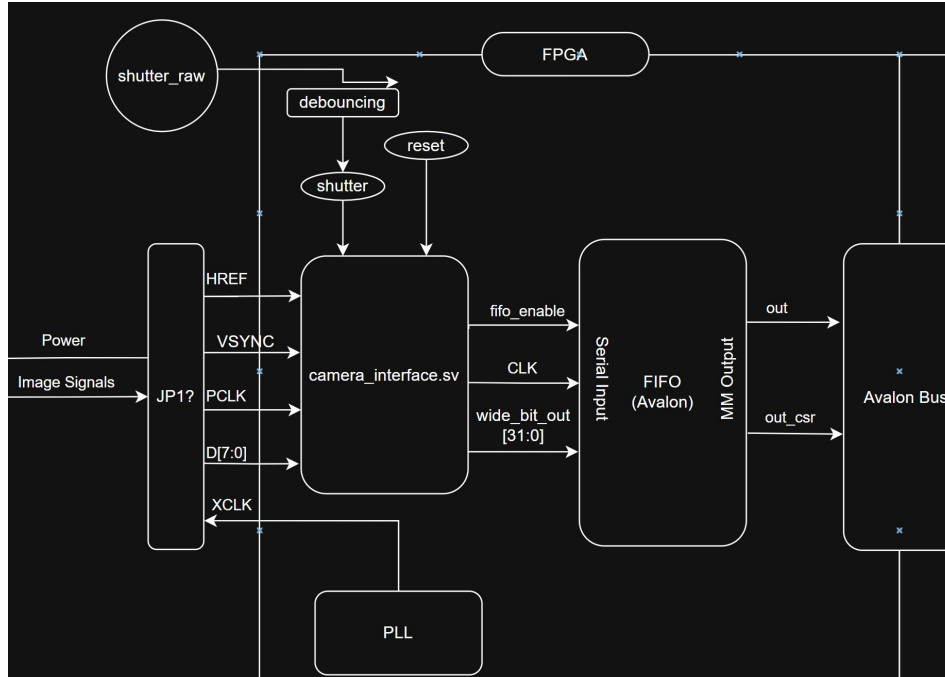
Hardware Design (Camera → FPGA Fabric → Avalon Bus)

- Main Tasks:

- Wait for capture flag & VSYNC → find middle row
- Stream exactly 640 pixels into FIFO
- Assert data ready handshake to HPS

Use	Connections	Name	Description	Export
✓		clk_0 clk_in clk_in_reset clk clk_reset	Clock Source Clock Input Reset Input Clock Output Reset Output	clk reset <i>Double-click to</i> <i>Double-click to</i>
✓		fifo_0 clk_in reset_in clk_out reset_out in out out_csr	Avalon FIFO Memory Intel FPG... Clock Input Reset Input Clock Input Reset Input Avalon Streaming Sink Avalon Memory Mapped Slave Avalon Memory Mapped Slave	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>
✓		hps_0 h2f_user1_clock memory hps_io h2f_reset h2f_axi_clock h2f_axi_master f2h_axi_clock f2h_axi_slave h2f_lw_axi_clock h2f_lw_axi_master	Arria V/Cyclone V Hard Proce... Clock Output Conduit Conduit Reset Output Clock Input AXI Master Clock Input AXI Slave Clock Input AXI Master	<i>Double-click to</i> hps_ddr3 hps <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>
✓		camera_interface_0 reset clock_sink camera_data camera_interfac...	CAMERA INTERFACE Reset Input Clock Input Conduit Avalon Streaming Source	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>
✓		pll_0 refclk reset outclk0	PLL Intel FPGA IP Clock Input Reset Input Clock Output	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>
✓		camera_clk_0 clock_input pclk_io pclk_fpga xclk_io	CAMERA_CLK Clock Input Clock Input Clock Output Clock Output	<i>Double-click to</i> pclk_io <i>Double-click to</i> xclk_io
✓		camera_io_0 camera_io camera_data reset_fake clk_fake	CAMERA_IO Conduit Conduit Reset Input Clock Input	camera_io <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>

camera_interface.sv



States:

- BLOCK
- RESET
- SHUTTER
- WRITE

Write every 4th clk cycle

FIFO 520 bit depth

The Device Driver

Register Map

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

```
#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"
```

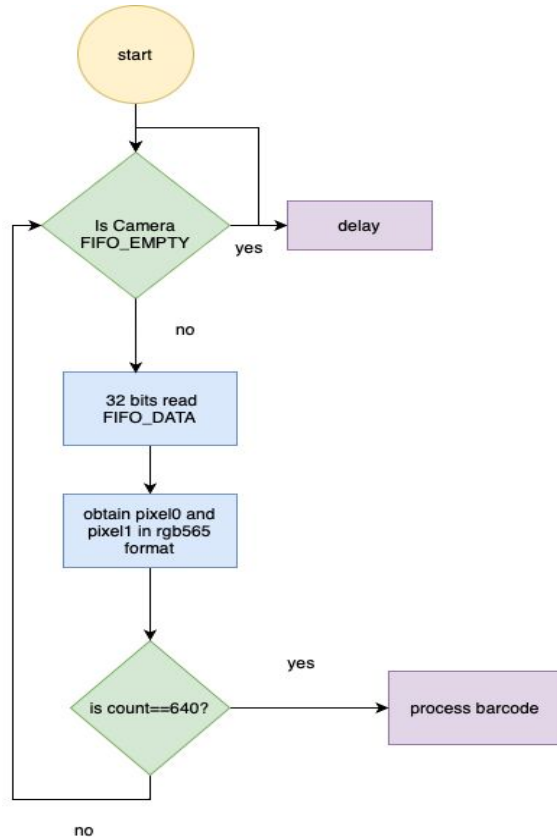
```
static long camera_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
        case CAMERA_READ_WORD: {
            uint32_t word = ioread32(dev.scanline_base);
            if (copy_to_user((uint32_t *)arg, &word, sizeof(uint32_t)))
                return -EFAULT;
            break;
        }

        case CAMERA_FIFO_EMPTY: {
            uint32_t 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;
}
```

Software Flow Control (the controller)



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

```
// 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++;  
}
```

Barcode Decoding

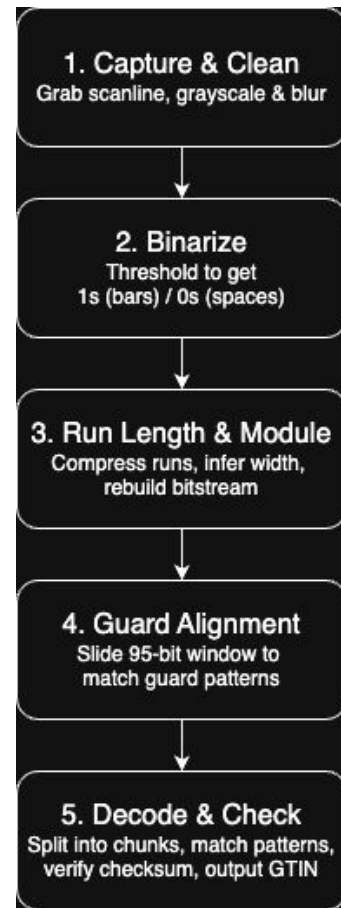
Demo!



```
[0 0 0 ... 0 0 0]  
Shift used: 0  
Checksum valid? True  
UPC-A Barcode Number: 049022895270  
Average Confidence: 1.00  
Hamming Dist: 0
```

Barcode Decoding Algorithm

1. Grayscale & Smooth
 - Convert RGB565 pixels → 8-bit gray + apply 5-point box blur
2. Binary Spaces
 - Threshold blurred values into 1 (bar) or 0 (space)
3. Run-Length Encode
 - Count consecutive 1s / 0s to get run lengths
4. Module & Bitstream
 - Find median run → bar-unit width → expand runs into flat “1010...” string
5. Guard Alignment
 - Slide 95-bit window to best match start/middle/end patterns
6. Decode & Output
 - Split into 12x7-bit groups
 - Map each to a digit
 - Return 12-digit code + ‘\0’ or NULL on failure



Conclusion

- **Complete Pipeline:**
 - Real-time EAN-13 decoding from a single 24 MHz scanline on OV7670 + DE1-SoC
- **Hardware/Software Co-Design:**
 - FPGA handles precise pixel capture and buffering
 - HPS runs robust decode & display logic
- **Performance & Accuracy:**
 - Consistently decodes 95-bit windows with checksum verification in < 3 s

Challenges:

- **OV7670 Module:**
 - Did not output correctly