Lecture #2 – Digital Camera Example
Outline

• Introduction to a simple digital camera
• Designer’s perspective
• Requirements specification
• Design
  • Four implementations

Introduction

• Putting it all together
  • General-purpose processor
  • Single-purpose processor
  • Memory
  • Interfacing

• Knowledge applied to designing a simple digital camera
  • General-purpose vs. single-purpose processors
  • Partitioning of functionality among different processor types
Digital Camera Introduction

- Captures images
- Stores images in digital format
  - No film
  - Multiple images stored in camera
    - Number depends on amount of memory and bits used per image
- Downloads images to PC
- Only recently possible
  - Systems-on-a-chip
    - Multiple processors and memories on one IC
  - High-capacity flash memory
- Very simple description used for example
  - Many more features with real digital camera
    - Variable size images, image deletion, digital stretching, zooming in and out, etc.
Designer’s Perspective

- Two key tasks
  - Processing images and storing in memory
    - When shutter pressed:
      - Image captured
      - Converted to digital form by charge-coupled device (CCD)
      - Compressed and archived in internal memory
  - Uploading images to PC
    - Digital camera attached to PC
    - Special software commands camera to transmit archived images serially
Charge-Coupled Device (CCD)

- Special sensor that captures an image
- Light-sensitive silicon solid-state device composed of many cells

When exposed to light, each cell becomes electrically charged. This charge can then be converted to a 8-bit value where 0 represents no exposure while 255 represents very intense exposure of that cell to light.

Some of the columns are covered with a black strip of paint. The light-intensity of these pixels is used for zero-bias adjustments of all the cells.

The electromechanical shutter is activated to expose the cells to light for a brief moment.

The electronic circuitry, when commanded, discharges the cells, activates the electromechanical shutter, and then reads the 8-bit charge value of each cell. These values can be clocked out of the CCD by external logic through a standard parallel bus interface.
Zero-Bias Error

- Manufacturing errors cause cells to measure slightly above or below actual light intensity
- Error typically same across columns, but different across rows
- Some of left most columns blocked by black paint to detect zero-bias error
  - Reading of other than 0 in blocked cells is zero-bias error
  - Each row is corrected by subtracting the average error found in blocked cells for that row

<table>
<thead>
<tr>
<th>Covered cells</th>
<th>Zero-bias adjustment</th>
</tr>
</thead>
<tbody>
<tr>
<td>136 170 155 140 144 115 112 248</td>
<td>-13</td>
</tr>
<tr>
<td>145 146 168 123 120 117 119 147</td>
<td>-11</td>
</tr>
<tr>
<td>144 153 168 117 121 127 118 135</td>
<td>-9</td>
</tr>
<tr>
<td>176 183 161 111 186 130 132 133</td>
<td>0</td>
</tr>
<tr>
<td>144 156 161 133 192 153 138 139</td>
<td>-7</td>
</tr>
<tr>
<td>122 131 128 147 206 151 131 127</td>
<td>2</td>
</tr>
<tr>
<td>121 155 164 185 254 165 138 129</td>
<td>4</td>
</tr>
<tr>
<td>173 175 176 183 188 184 117 129</td>
<td>5</td>
</tr>
</tbody>
</table>

Before zero-bias adjustment

<table>
<thead>
<tr>
<th>Covered cells</th>
<th>Zero-bias adjustment</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 157 142 127 131 102 99 235</td>
<td>-13</td>
</tr>
<tr>
<td>134 135 157 112 109 106 108 136</td>
<td>-11</td>
</tr>
<tr>
<td>135 144 159 108 112 118 109 126</td>
<td>-9</td>
</tr>
<tr>
<td>176 183 161 111 186 130 132 133</td>
<td>0</td>
</tr>
<tr>
<td>137 149 154 126 185 146 131 132</td>
<td>-7</td>
</tr>
<tr>
<td>121 130 127 146 205 150 130 126</td>
<td>-1</td>
</tr>
<tr>
<td>117 151 160 181 250 161 134 125</td>
<td>-4</td>
</tr>
<tr>
<td>168 170 171 178 183 179 112 124</td>
<td>-5</td>
</tr>
</tbody>
</table>

After zero-bias adjustment
Compression

- Store more images
- Transmit image to PC in less time
- JPEG (Joint Photographic Experts Group)
  - Popular standard format for representing digital images in a compressed form
  - Provides for a number of different modes of operation
  - Mode used in this chapter provides high compression ratios using DCT (discrete cosine transform)
  - Image data divided into blocks of 8 x 8 pixels
  - 3 steps performed on each block
    - DCT
    - Quantization
    - Huffman encoding
DCT Step

- Transforms original 8 x 8 block into a cosine-frequency domain
  - Upper-left corner values represent more of the essence of the image
  - Lower-right corner values represent finer details
    - Can reduce precision of these values and retain reasonable image quality

- FDCT (Forward DCT) formula
  - $C(h) = \begin{cases} 1/\sqrt{2} & \text{if } h = 0 \\ 1.0 & \text{else} \end{cases}$
    - Auxiliary function used in main function $F(u,v)$
  - $F(u,v) = \frac{1}{4} \times C(u) \times C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} D_{xy} \times \cos(\pi(2x + 1)u/16) \times \cos(\pi(2y + 1)v/16)$
    - Gives encoded pixel at row $u$, column $v$
    - $D_{xy}$ is original pixel value at row $x$, column $y$

- IDCT (Inverse DCT)
  - Reverses process to obtain original block (not needed for this design)
Quantization Step

- Achieve high compression ratio by reducing image quality
  - Reduce bit precision of encoded data
    - Fewer bits needed for encoding
    - One way is to divide all values by a factor of 2
      - Simple right shifts can do this
  - Dequantization would reverse process for decompression

<table>
<thead>
<tr>
<th>1150</th>
<th>39</th>
<th>-43</th>
<th>-10</th>
<th>26</th>
<th>-83</th>
<th>11</th>
<th>41</th>
</tr>
</thead>
<tbody>
<tr>
<td>-81</td>
<td>-3</td>
<td>115</td>
<td>-73</td>
<td>-6</td>
<td>-2</td>
<td>22</td>
<td>-5</td>
</tr>
<tr>
<td>14</td>
<td>-11</td>
<td>1</td>
<td>-42</td>
<td>26</td>
<td>-3</td>
<td>17</td>
<td>-38</td>
</tr>
<tr>
<td>2</td>
<td>-61</td>
<td>-13</td>
<td>-12</td>
<td>36</td>
<td>-23</td>
<td>-18</td>
<td>5</td>
</tr>
<tr>
<td>44</td>
<td>37</td>
<td>-4</td>
<td>10</td>
<td>-21</td>
<td>7</td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>-11</td>
<td>-9</td>
<td>-4</td>
<td>20</td>
<td>-28</td>
<td>-21</td>
<td>14</td>
</tr>
<tr>
<td>-19</td>
<td>-7</td>
<td>21</td>
<td>-6</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>-21</td>
</tr>
<tr>
<td>-5</td>
<td>-13</td>
<td>-11</td>
<td>-17</td>
<td>-4</td>
<td>-1</td>
<td>7</td>
<td>-4</td>
</tr>
</tbody>
</table>

After being decoded using DCT

<table>
<thead>
<tr>
<th>144</th>
<th>5</th>
<th>-5</th>
<th>-1</th>
<th>3</th>
<th>-10</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>0</td>
<td>14</td>
<td>-9</td>
<td>-1</td>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>0</td>
<td>-5</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>-5</td>
</tr>
<tr>
<td>0</td>
<td>-8</td>
<td>-2</td>
<td>-2</td>
<td>5</td>
<td>-3</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>-1</td>
<td>1</td>
<td>-3</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>-4</td>
<td>-3</td>
<td>2</td>
</tr>
<tr>
<td>-2</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>-3</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Divide each cell’s value by 8

After quantization
Huffman Encoding Step

- Serialize 8 x 8 block of pixels
  - Values are converted into single list using zigzag pattern

- Perform Huffman encoding
  - More frequently occurring pixels assigned short binary code
  - Longer binary codes left for less frequently occurring pixels
  - Each pixel in serial list converted to Huffman encoded values
  - Much shorter list, thus compression
Huffman Encoding Example

- Pixel frequencies on left
  - Pixel value –1 occurs 15 times
  - Pixel value 14 occurs 1 time
- Build Huffman tree from bottom up
  - Create one leaf node for each pixel value and assign frequency as node’s value
  - Create an internal node by joining any two nodes whose sum is a minimal value
    - This sum is internal nodes value
  - Repeat until complete binary tree
- Traverse tree from root to leaf to obtain binary code for leaf’s pixel value
  - Append 0 for left traversal, 1 for right traversal
- Huffman encoding is reversible
  - No code is a prefix of another code
Archive Step

- Record starting address and image size
  - Can use linked list
- One possible way to archive images
  - If max number of images archived is N:
    - Set aside memory for N addresses and N image-size variables
    - Keep a counter for location of next available address
    - Initialize addresses and image-size variables to 0
    - Set global memory address to N x 4
      - Assuming addresses, image-size variables occupy N x 4 bytes
    - First image archived starting at address N x 4
    - Global memory address updated to N x 4 + (compressed image size)
- Memory requirement based on N, image size, and average compression ratio
Uploading to PC

- When connected to PC and upload command received
  - Read images from memory
  - Transmit serially using UART
  - While transmitting
    - Reset pointers, image-size variables and global memory pointer accordingly
Requirements Specification

• System’s requirements – what system should do
  • Nonfunctional requirements
    • Constraints on design metrics (e.g., “should use 0.001 watt or less”)
  • Functional requirements
    • System’s behavior (e.g., “output X should be input Y times 2”)
• Initial specification may be very general and come from marketing dept.
  • E.g., short document detailing market need for a low-end digital camera that:
    • captures and stores at least 50 low-res images and uploads to PC,
    • costs around $100 with single medium-size IC costing less than $25,
    • has as long as possible battery life,
    • has expected sales volume of 200,000 if market entry < 6 months,
    • 100,000 if between 6 and 12 months,
    • insignificant sales beyond 12 months
Nonfunctional Requirements

- Design metrics of importance based on initial specification
  - **Performance**: time required to process image
  - **Size**: number of elementary logic gates (2-input NAND gate) in IC
  - **Power**: measure of avg. electrical energy consumed while processing
  - **Energy**: battery lifetime (power x time)
- Constrained metrics
  - Values **must** be below (sometimes above) certain threshold
- Optimization metrics
  - Improved as much as possible to improve product
  - Metric can be both constrained and optimization
Nonfunctional Requirements (cont.)

- **Performance**
  - Must process image fast enough to be useful
  - 1 sec reasonable constraint
    - Slower would be annoying
    - Faster not necessary for low-end of market
  - Therefore, constrained metric

- **Size**
  - Must use IC that fits in reasonably sized camera
  - Constrained and optimization metric
    - Constraint may be 200,000 gates, but smaller would be cheaper

- **Power**
  - Must operate below certain temperature (cooling fan not possible)
  - Therefore, constrained metric

- **Energy**
  - Reducing power or time reduces energy
  - Optimized metric: want battery to last as long as possible
Informal Functional Specification

- Flowchart breaks functionality down into simpler functions
- Each function’s details could then be described in English
- Low quality image has resolution of 64 x 64
- Mapping functions to a particular processor type not done at this stage
Refined Functional Specification

- Refine informal specification into one that can actually be executed
- Can use C/C++ code to describe each function
  - Called system-level model, prototype, or simply model
  - Also is first implementation
- Can provide insight into operations of system
  - Profiling can find computationally intensive functions
- Can obtain sample output used to verify correctness of final implementation
CCD Module

- Simulates real CCD
- *CcdInitialize* is passed name of image file
- *CcdCapture* reads “image” from file
- *CcdPopPixel* outputs pixels one at a time

```c
#include <stdio.h>
define SZ_ROW 64
define SZ_COL (64 + 2)
static FILE *imageFileHandle;
static char buffer[SZ_ROW][SZ_COL];
static unsigned rowIndex, colIndex;

void CcdInitialize(const char *imageFileName) {
    imageFileHandle = fopen(imageFileName, "r");
    rowIndex = -1;
    colIndex = -1;
}

void CcdCapture(void) {
    int pixel;
    rewind(imageFileHandle);
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            if( fscanf(imageFileHandle, "%i", &pixel) == 1 ) {
                buffer[rowIndex][colIndex] = (char)pixel;
            }
        }
    }
    rowIndex = 0;
    colIndex = 0;
}
```

```c
char CcdPopPixel(void) {
    char pixel;
    pixel = buffer[rowIndex][colIndex];
    if( ++colIndex == SZ_COL ) {
        colIndex = 0;
        if( ++rowIndex == SZ_ROW ) {
            colIndex = -1;
            rowIndex = -1;
        }
    }
    return pixel;
}
```
CCDPP Module

- Performs zero-bias adjustment
- **CcdppCapture** uses **CcdCapture** and **CcdPopPixel** to obtain image
- Performs zero-bias adjustment after each row read in

```c
#define SZ_ROW 64
#define SZ_COL 64
static char buffer[SZ_ROW][SZ_COL];
static unsigned rowIndex, colIndex;

void CcdppInitialize() {
    rowIndex = -1;
    colIndex = -1;
}

void CcdppCapture(void) {
    char bias;
    CcdCapture();
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] = CcdPopPixel();
        }
        bias = (CcdPopPixel() + CcdPopPixel()) / 2;
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] -= bias;
        }
    }
    rowIndex = 0;
    colIndex = 0;
}

char CcdppPopPixel(void) {
    char pixel;
    pixel = buffer[rowIndex][colIndex];
    if( ++colIndex == SZ_COL ) {
        colIndex = 0;
        if( ++rowIndex == SZ_ROW ) {
            rowIndex = -1;
            colIndex = -1;
        }
    }
    return pixel;
}
```
UART Module

- Actually a half UART
  - Only transmits, does not receive

- `UartInitialize` is passed name of file to output to

- `UartSend` transmits (writes to output file) bytes at a time

```c
#include <stdio.h>
static FILE *outputFileHandle;
void UartInitialize(const char *outputFileName) {
    outputFileHandle = fopen(outputFileName, "w");
}
void UartSend(char d) {
    fprintf(outputFileHandle, "%i\n", (int)d);
}
```
CODEC Module

- Models FDCT encoding
- `ibuffer` holds original 8 x 8 block
- `obuffer` holds encoded 8 x 8 block
- `CodecPushPixel` called 64 times to fill `ibuffer` with original block
- `CodecDoFdct` called once to transform 8 x 8 block
  - Explained in next slide
- `CodecPopPixel` called 64 times to retrieve encoded block from `obuffer`

```c
static short ibuffer[8][8], obuffer[8][8], idx;

void CodecInitialize(void) { idx = 0; }

void CodecDoFdct(void) {
    int x, y;
    for(x=0; x<8; x++) {
        for(y=0; y<8; y++)
            obuffer[x][y] = FDCT(x, y, ibuffer);
    }
    idx = 0;
}

void CodecPushPixel(short p) {
    if( idx == 64 ) idx = 0;
    ibuffer[idx / 8][idx % 8] = p; idx++;
}

short CodecPopPixel(void) {
    short p;
    if( idx == 64 ) idx = 0;
    p = obuffer[idx / 8][idx % 8]; idx++;
    return p;
}
```
CODEC Module (cont.)

- Implementing FDCT formula
  \[ C(h) = \begin{cases} 1 / \sqrt{2} & \text{if } h = 0 \\ 1.0 & \text{else} \end{cases} \]
  \[ F(u,v) = \frac{1}{4} \times C(u) \times C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} D_{xy} \times \cos\left(\pi(2x + 1)u/16\right) \times \cos\left(\pi(2y + 1)v/16\right) \]

- Only 64 possible inputs to \( \text{COS} \), so table can be used to save performance time
  - Floating-point values multiplied by 32,678 and rounded to nearest integer
  - 32,678 chosen in order to store each value in 2 bytes of memory
  - Fixed-point representation explained more later

- \( \text{FDCT} \) unrolls inner loop of summation, implements outer summation as two consecutive for loops

```c
static const short COS_TABLE[8][8] = {
    { 32768,  32138,  30273,  27245,  23170,  18204,  12539,   6392 },
    { 32768,  27245,  12539,  -6392, -23170, -30273, -18204 },
    { 32768,  18204, -12539, -32138,  30273,  27245 },
    { 32768,  -6392, -30273, -18204,  23170,  27245, -12539, -32138 },
    { 32768, -18204, -12539,  32138, -23170,  -6392,  30273, -27245 },
    { 32768, -27245,  12539,  6392, -30273, -18204,  23170,  27245 },
    { 32768, -32138,  30273, -27245,  23170, -18204,  12539, -6392 }
};

static int FDCT(int u, int v, short img[8][8]) {
    double s[8], r = 0; int x;
    for(x=0; x<8; x++) {
        s[x] = img[x][0] * COS(0, v) + img[x][1] * COS(1, v) +
               img[x][2] * COS(2, v) + img[x][3] * COS(3, v) +
               img[x][4] * COS(4, v) + img[x][5] * COS(5, v) +
               img[x][6] * COS(6, v) + img[x][7] * COS(7, v);
    }
    for(x=0; x<8; x++) r += s[x] * COS(x, u);
    return (short)(r * .25 * C(u) * C(v));
}
```

static short ONE_OVER_SQRT_TWO = 23170;
static double COS(int xy, int uv) {
    return COS_TABLE[xy][uv] / 32768.0;
}
static double C(int h) {
    return h ? 1.0 : ONE_OVER_SQRT_TWO / 32768.0;
}
CNTRL (Controller) Module

- Heart of the system
- CntrlInitialize for consistency with other modules only
- CntrlCaptureImage uses CCDPP module to input image and place in buffer
  - Also performs quantization on each block
- CntrlCompressImage breaks the 64 x 64 buffer into 8 x 8 blocks and performs FDCT on each block using the CODEC module
  - Also performs quantization on each block
- CntrlSendImage transmits encoded image serially using UART module

```c
#define SZ_ROW 64
#define SZ_COL 64
#define NUM_ROW_BLOCKS (SZ_ROW / 8)
#define NUM_COL_BLOCKS (SZ_COL / 8)
static short buffer[SZ_ROW][SZ_COL], i, j, k, l, temp;
void CntrlInitialize(void) {}

void CntrlCaptureImage(void) {
    CcdppCapture();
    for (i = 0; i < SZ_ROW; i++)
        for (j = 0; j < SZ_COL; j++)
            buffer[i][j] = CcdppPopPixel();
}

void CntrlSendImage(void) {
    for (i = 0; i < SZ_ROW; i++)
        for (j = 0; j < SZ_COL; j++) {
            temp = buffer[i][j];
            UartSend((char*)&temp[0]); /* send upper byte */
            UartSend((char*)&temp[1]); /* send lower byte */
        }
}

void CntrlCompressImage(void) {
    for (i = 0; i < NUM_ROW_BLOCKS; i++)
        for (j = 0; j < NUM_COL_BLOCKS; j++) {
            for (k = 0; k < 8; k++)
                for (l = 0; l < 8; l++)
                    CodecPushPixel((char)buffer[i * 8 + k][j * 8 + l]);
            CodecDoFdct(); /* part 1 - FDCT */
            for (k = 0; k < 8; k++)
                for (l = 0; l < 8; l++)
                    buffer[i*8+k][j*8+l] >>= 6;
        }
    CodecDoNumBlocks(); /* part 2 - quantization */
}
```

```c
for (i = 0; i < SZ_ROW; i++)
    for (j = 0; j < SZ_COL; j++)
        temp = buffer[i][j];
for (i = 0; i < NUM_ROW_BLOCKS; i++)
    for (j = 0; j < NUM_COL_BLOCKS; j++)
        for (k = 0; k < 8; k++)
            for (l = 0; l < 8; l++)
                CodecPushPixel((char)buffer[i*8+k][j*8+l]);
for (i = 0; i < NUM_ROW_BLOCKS; i++)
    for (j = 0; j < NUM_COL_BLOCKS; j++)
        CodecDoFdct(); /* part 1 - FDCT */
for (i = 0; i < NUM_ROW_BLOCKS; i++)
    for (j = 0; j < NUM_COL_BLOCKS; j++)
        CodecDoNumBlocks(); /* part 2 - quantization */
```
Putting it All Together

- **Main** initializes all modules, then uses CNTRL module to capture, compress, and transmit one image
- This system-level model can be used for extensive experimentation
  - Bugs much easier to correct here rather than in later models

```c
int main(int argc, char *argv[]) {
    char *uartOutputFileName = argc > 1 ? argv[1] : "uart_out.txt";
    /* initialize the modules */
    UartInitialize(uartOutputFileName);
    CcdInitialize(imageFileName);
    CcdppInitialize();
    CodecInitialize();
    CntrlInitialize();
    /* simulate functionality */
    CntrlCaptureImage();
    CntrlCompressImage();
    CntrlSendImage();
}
```
Design

- Determine system’s architecture
  - Processors
    - Any combination of single-purpose (custom or standard) or general-purpose processors
  - Memories, buses
- Map functionality to that architecture
  - Multiple functions on one processor
  - One function on one or more processors
- Implementation
  - A particular architecture and mapping
  - Solution space is set of all implementations
- Starting point
  - Low-end general-purpose processor connected to flash memory
    - All functionality mapped to software running on processor
    - Usually satisfies power, size, and time-to-market constraints
    - If timing constraint not satisfied then later implementations could:
      - use single-purpose processors for time-critical functions
      - rewrite functional specification
Implementation 1

- Low-end processor could be Intel 8051 microcontroller
- Total IC cost including NRE about $5
- Well below 200 mW power
- Time-to-market about 3 months
- However, one image per second not possible
  - 12 MHz, 12 cycles per instruction
    - Executes one million instructions per second
  - CcppCapture has nested loops resulting in 4096 (64 x 64) iterations
    - ~100 assembly instructions each iteration
    - 409,000 (4096 x 100) instructions per image
    - Half of budget for reading image alone
  - Would be over budget after adding compute-intensive DCT and Huffman encoding
Implementation 2

- CCDPP function implemented on custom single-purpose processor
  - Improves performance – less microcontroller cycles
  - Increases NRE cost and time-to-market
  - Easy to implement
    - Simple datapath
    - Few states in controller
- Simple UART easy to implement as single-purpose processor also
- EEPROM for program memory and RAM for data memory added as well
Microcontroller

- Synthesizable version of Intel 8051 available
  - Written in VHDL
  - Captured at register transfer level (RTL)
- Fetches instruction from ROM
- Decodes using Instruction Decoder
- ALU executes arithmetic operations
  - Source and destination registers reside in RAM
- Special data movement instructions used to load and store externally
- Special program generates VHDL description of ROM from output of C compiler/linker

Block diagram of Intel 8051 processor core
UART

- UART in idle mode until invoked
  - UART invoked when 8051 executes store instruction with UART’s enable register as target address
    - Memory-mapped communication between 8051 and all single-purpose processors
    - Lower 8-bits of memory address for RAM
    - Upper 8-bits of memory address for memory-mapped I/O devices
- Start state transmits 0 indicating start of byte transmission then transitions to Data state
- Data state sends 8 bits serially then transitions to Stop state
- Stop state transmits 1 indicating transmission done then transitions back to idle mode
**CCDPP**

- Hardware implementation of zero-bias operations
- Interacts with external CCD chip
  - CCD chip resides external to our SOC mainly because combining CCD with ordinary logic not feasible
- Internal buffer, $B$, memory-mapped to 8051
- Variables $R$, $C$ are buffer’s row, column indices
- GetRow state reads in one row from CCD to $B$
  - 66 bytes: 64 pixels + 2 blacked-out pixels
- ComputeBias state computes bias for that row and stores in variable $Bias$
- FixBias state iterates over same row subtracting $Bias$ from each element
- NextRow transitions to GetRow for repeat of process on next row or to Idle state when all 64 rows completed
Connecting SOC Components

- Memory-mapped
  - All single-purpose processors and RAM are connected to 8051’s memory bus
- Read
  - Processor places address on 16-bit address bus
  - Asserts read control signal for 1 cycle
  - Reads data from 8-bit data bus 1 cycle later
  - Device (RAM or SPP) detects asserted read control signal
  - Checks address
  - Places and holds requested data on data bus for 1 cycle
- Write
  - Processor places address and data on address and data bus
  - Asserts write control signal for 1 clock cycle
  - Device (RAM or SPP) detects asserted write control signal
  - Checks address bus
  - Reads and stores data from data bus
Software

- System-level model provides majority of code
  - Module hierarchy, procedure names, and main program unchanged
- Code for UART and CCDPP modules must be redesigned
  - Simply replace with memory assignments
    - \textit{xdata} used to load/store variables over external memory bus
    - \textit{at} specifies memory address to store these variables
    - Byte sent to \textit{U_TX_REG} by processor will invoke UART
    - \textit{U_STAT_REG} used by UART to indicate its ready for next byte
      - UART may be much slower than processor
  - Similar modification for CCDPP code
- All other modules untouched

\begin{tabular}{ll}
\textbf{Original code from system-level model} & \textbf{Rewritten UART module} \\
\end{tabular}

\begin{verbatim}
#include <stdio.h>
static FILE *outputFileHandle;
void UartInitialize(const char *outputFileName) {
    outputFileHandle = fopen(outputFileName, "w");
}
void UartSend(char d) {
    fprintf(outputFileHandle, "%i\n", (int)d);
}
\end{verbatim}

\begin{verbatim}
static unsigned char xdata U_TX_REG _at_ 65535;
static unsigned char xdata U_STAT_REG _at_ 65534;
void UARTInitialize(void) {}  
void UARTSend(unsigned char d) {
    while( U_STAT_REG == 1 ) {
        /* busy wait */
    }
    U_TX_REG = d;
}
\end{verbatim}
Analysis

- Entire SOC tested on VHDL simulator
  - Interprets VHDL descriptions and functionally simulates execution of system
    - Recall program code translated to VHDL description of ROM
  - Tests for correct functionality
  - Measures clock cycles to process one image (performance)
- Gate-level description obtained through synthesis
  - Synthesis tool like compiler for SPPs
  - Simulate gate-level models to obtain data for power analysis
    - Number of times gates switch from 1 to 0 or 0 to 1
  - Count number of gates for chip area

Obtaining design metrics of interest

- VHDL simulator
- Synthesis tool
- Gate level simulator
- Execution time
- Sum gates
- Power equation
- Chip area
- Power
Implementation 2 (cont.)

- Analysis of implementation 2
  - Total execution time for processing one image:
    - 9.1 seconds
  - Power consumption:
    - 0.033 watt
  - Energy consumption:
    - 0.30 joule (9.1 s x 0.033 watt)
  - Total chip area:
    - 98,000 gates
Implementation 3

- 9.1 seconds still doesn’t meet performance constraint of 1 second
- DCT operation prime candidate for improvement
  - Execution of implementation 2 shows microprocessor spends most cycles here
  - Could design custom hardware like we did for CCDPP
    - More complex so more design effort
  - Instead, will speed up DCT functionality by modifying behavior
DCT Floating-Point Cost

- Floating-point cost
  - DCT uses ~260 floating-point operations per pixel transformation
  - 4096 (64 x 64) pixels per image
  - 1 million floating-point operations per image
  - No floating-point support with Intel 8051
    - Compiler must emulate
      - Generates procedures for each floating-point operation
        - mult, add
      - Each procedure uses tens of integer operations
  - Thus, > 10 million integer operations per image
  - Procedures increase code size
- Fixed-point arithmetic can improve on this
Fixed-Point Arithmetic

- Integer used to represent a real number
  - Constant number of integer’s bits represents fractional portion of real number
    - More bits, more accurate the representation
  - Remaining bits represent portion of real number before decimal point
- Translating a real constant to a fixed-point representation
  - Multiply real value by $2^\text{(number of bits used for fractional part)}$
  - Round to nearest integer
  - E.g., represent $3.14$ as 8-bit integer with 4 bits for fraction
    - $2^4 = 16$
    - $3.14 \times 16 = 50.24 \approx 50 = 00110010$
    - 16 ($2^4$) possible values for fraction, each represents 0.0625 (1/16)
    - Last 4 bits (0010) = 2
    - $2 \times 0.0625 = 0.125$
    - $3(0011) + 0.125 = 3.125 \approx 3.14$ (more bits for fraction would increase accuracy)
Fixed-Point Arithmetic Operations

- Addition
  - Simply add integer representations
  - E.g., \(3.14 + 2.71 = 5.85\)
    - \(3.14 \rightarrow 50 = 00110010\)
    - \(2.71 \rightarrow 43 = 00101011\)
    - \(50 + 43 = 93 = 01011101\)
    - \(5(0101) + 13(1101) \times 0.0625 = 5.8125 \approx 5.85\)
- Multiply
  - Multiply integer representations
  - Shift result right by \# of bits in fractional part
  - E.g., \(3.14 \times 2.71 = 8.5094\)
    - \(50 \times 43 = 2150 = 100001100110\)
    - \(\gg 4 = 10000110\)
    - \(8(1000) + 6(0110) \times 0.0625 = 8.375 \approx 8.5094\)
- Range of real values used limited by bit widths of possible resulting values
Fixed-Point CODEC

- COS_TABLE gives 8-bit fixed-point representation of cosine values
- 6 bits used for fractional portion
- Result of multiplications shifted right by 6

```c
#define ONEOVERSQRTTWO 5

static const char code COS_TABLE[8][8] = {
    {  64,  62,  59,  53,  45,  35,  24,  12 },
    {  64,  53,  24, -12, -45, -62, -59, -35 },
    {  64, -24, -62, -45,  12,  59,  53,  35 },
    {  64,  12, -59, -35,  45,  53, -24, -62 },
    {  64, -12, -59,  35,  45, -24,  62,  62 },
    {  64, -35,  24,  12, -45,  62, -59, -53 },
    {  64, -53,  24,  12, -59,  35, -24,  62 },
    {  64, -62,  59, -53,  45, -35,  24, -12 }
};

static const char ONEOVERSQRTTWO = 5;

static short xdata inBuffer[8][8], outBuffer[8][8], idx;
void CodecInitialize(void) { idx = 0; }

void CodecPushPixel(short p) {
    if( idx == 64 ) idx = 0;
inBuffer[idx / 8][idx % 8] = p << 6; idx++;
}

void CodecDoFdct(void) {
    unsigned short x, y;
    for(x=0; x<8; x++)
        for(y=0; y<8; y++)
            outBuffer[x][y] = F(x, y, inBuffer);
    idx = 0;
}
```

```c
static unsigned char C(int h) { return h ? 64 : ONEOVERSQRTTWO; }
static int F(int u, int v, short img[8][8]) {
    long s[8], r = 0;
    unsigned char x, j;
    for(x=0; x<8; x++) {
        s[x] = 0;
        for(j=0; j<8; j++)
            s[x] += (img[x][j] * COS_TABLE[j][v] ) >> 6;
    }
    for(x=0; x<8; x++) r += (s[x] * COS_TABLE[x][u]) >> 6;
    return (short)((((r * (((16*C(u)) >> 6) *C(v)) >> 6)) >> 6) >> 6);
}
```
Implementation 3 (cont.)

- Analysis of implementation 3
  - Use same analysis techniques as implementation 2
  - Total execution time for processing one image:
    - 1.5 seconds
  - Power consumption:
    - 0.033 watt (same as 2)
  - Energy consumption:
    - 0.050 joule (1.5 s x 0.033 watt)
    - Battery life 6x longer!!
  - Total chip area:
    - 90,000 gates
    - 8,000 less gates (less memory needed for code)
Implementation 4

- Performance close but not good enough
- Must resort to implementing CODEC in hardware
  - Single-purpose processor to perform DCT on 8 x 8 block
CODEC Design

- 4 memory mapped registers
  - `C_DATAI_REG/C_DATAO_REG` used to push/pop 8 x 8 block into and out of CODEC
  - `C_CMND_REG` used to command CODEC
    - Writing 1 to this register invokes CODEC
  - `C_STAT_REG` indicates CODEC done and ready for next block
    - Polled in software
- Direct translation of C code to VHDL for actual hardware implementation
  - Fixed-point version used
- CODEC module in software changed similar to UART/CCDPP

Rewritten CODEC software

```c
static unsigned char xdata C_STAT_REG _at_ 65527;
static unsigned char xdata C_CMND_REG _at_ 65528;
static unsigned char xdata C_DATAI_REG _at_ 65529;
static unsigned char xdata C_DATAO_REG _at_ 65530;
void CodecInitialize(void) {}
void CodecPushPixel(short p) { C_DATAO_REG = (char)p; }
short CodecPopPixel(void) {
    return ((C_DATAI_REG << 8) | C_DATAI_REG);
}
void CodecDoFdct(void) {
    C_CMND_REG = 1;
    while( C_STAT_REG == 1 ) { /* busy wait */ }
}
```
Implementation 4 (cont.)

- Analysis of implementation 4
  - Total execution time for processing one image:
    - 0.099 seconds (well under 1 sec)
  - Power consumption:
    - 0.040 watt
    - Increase over 2 and 3 because SOC has another processor
  - Energy consumption:
    - 0.00040 joule (0.099 s x 0.040 watt)
    - Battery life 12x longer than previous implementation!!
  - Total chip area:
    - 128,000 gates
    - Significant increase over previous implementations
Summary of Implementations

- Implementation 3
  - Close in performance
  - Cheaper
  - Less time to build

- Implementation 4
  - Great performance and energy consumption
  - More expensive and may miss time-to-market window
    - If DCT designed ourselves then increased NRE cost and time-to-market
    - If existing DCT purchased then increased IC cost

- Which is better?

<table>
<thead>
<tr>
<th></th>
<th>Implementation 2</th>
<th>Implementation 3</th>
<th>Implementation 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (second)</td>
<td>9.1</td>
<td>1.5</td>
<td>0.099</td>
</tr>
<tr>
<td>Power (watt)</td>
<td>0.033</td>
<td>0.033</td>
<td>0.040</td>
</tr>
<tr>
<td>Size (gate)</td>
<td>98,000</td>
<td>90,000</td>
<td>128,000</td>
</tr>
<tr>
<td>Energy (joule)</td>
<td>0.30</td>
<td>0.050</td>
<td>0.0040</td>
</tr>
</tbody>
</table>
Summary

- Digital camera example
  - Specifications in English and executable language
  - Design metrics: performance, power and area
- Several implementations
  - Microcontroller: too slow
  - Microcontroller and coprocessor: better, but still too slow
  - Fixed-point arithmetic: almost fast enough
  - Additional coprocessor for compression: fast enough, but expensive and hard to design
- Tradeoffs between hw/sw – one of the main lessons of this course!