

# Embedded Systems Design: A Unified Hardware/Software Introduction

## Chapter 7 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
    - Custom
    - Standard
  - Memory
  - Interfacing
- Knowledge applied to designing a simple digital camera
  - General-purpose vs. single-purpose processors
  - Partitioning of functionality among different processor types

# Introduction to a simple digital camera

- 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

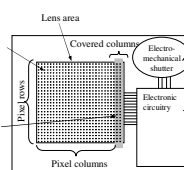
- Two key tasks
  - Capturing, processing images and storing in memory
    - When shutter pressed:
      - Image captured
      - Converted to digital form by charge-coupled device (CCD)
      - Compressed and archived in internal memory
  - Uploading images to PC
    - Digital camera attached to PC
    - 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 an 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.

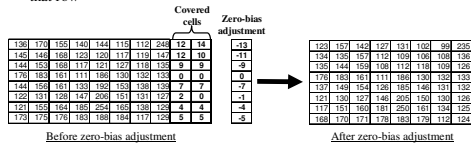


The electromechanical shutter is activated to expose the cells to light for a brief moment.

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



## Compression

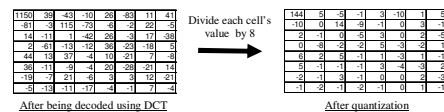
- Store more images
- Transmit image to PC in less time
- JPEG (Joint Photographic Experts Group)
  - Popular standard format for representing digital images in a compressed form
  - Provides for a number of different modes of operation
  - Image data divided into blocks of 8 x 8 pixels
  - 3 steps performed on each block
    - DCT (discrete cosine transform)
    - 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
  - Converts the array into embedded values which represent function of other values in same row and column
- IDCT (Inverse DCT)
  - Reverses process to obtain original block (not needed for this design)

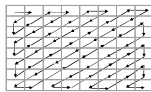
## Quantization step

- 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



## Huffman encoding step

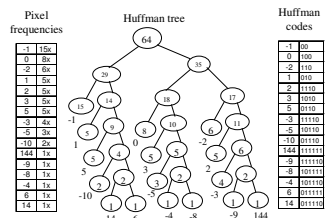
- Serialize 8 x 8 block of pixels
  - Values are converted into single list using zigzag pattern



- 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  $N \times 4$ 
      - Assuming addresses, image-size variables occupy  $N \times 4$  bytes
    - First image archived starting at address  $N \times 4$
    - Global memory address updated to  $N \times 4 + (\text{compressed image size})$
- 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 (universal asynchronous receiver/transmitter)
  - 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 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

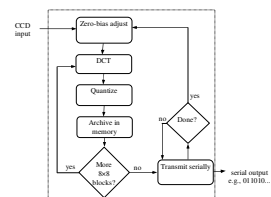
- Design metrics of importance based on initial specification
  - Performance:** time required to process image
  - Size:** number of elementary logic gates (2-input NAND gate) in IC
  - Power:** measure of avg. electrical energy consumed while processing
  - Energy:** battery lifetime (power x time)
- 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

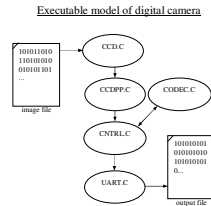
## Informal functional specification

- Flowchart breaks functionality down into simpler functions
- Each function's details could then be described in English
- Low quality image has resolution of  $64 \times 64$
- 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

```
#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;

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;
}
```

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

```
void CcdCapture(void) {
    int pixel;
    static FILE *imageFileHandle;
    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;
}
```

## CCDPP (CCD PreProcessing) module

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

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

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

```
void CdppInitialize() {
    rowIndex = -1;
    colIndex = -1;
}
```

```
char CdppPopPixel(void) {
    char pixel;
    pixel = buffer[rowIndex][colIndex];
    if( ++colIndex == SZ_COL ) {
        colIndex = 0;
        if( ++rowIndex == SZ_ROW ) {
            colIndex = -1;
            rowIndex = -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

```
#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
- CodecDoFdet* called once to transform 8 x 8 block
  - Explained in next slide
- CodecPopPixel* called 64 times to retrieve encoded block from *obuffer*

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

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

```
void CodecDoFdet(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;
}
```

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

## CODEC (cont.)

- Implementing FDCT formula
  - $C(u) = \text{if } (u == 0) \text{ then } 1/\sqrt{2} \text{ else } 1.0$
  - $F(x,y) = k_x \times C(u) \times C(v) \times \sum_{m=0}^{7-u} \sum_{n=0}^{7-v} D_{m,n} \times \cos(\pi/2 \times (2m+1)(u+1)/16) \times \cos(\pi/2 \times (2n+1)(v+1)/16)$
- Only 64 possible inputs to *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
- FDCT* unrolls inner loop of summation, implements outer summation as two consecutive for loops

```
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;
}
```

```
static const short COS_TABLE[8][8] = {
    { 32768, 32198, 30271, 27945, 25170, 22004, 18204, 12539, 6392 },
    { 32768, 27245, 22539, 16399, 12170, 72198, 30271, 18204 },
    { 32768, 18204, 12539, 72198, 25170, 30271, 32198, 27245 },
    { 32768, 6392, 30271, 18204, 25170, 27245, 12539, 32198 },
    { 32768, 18204, 12539, 32198, 25170, 6392, 30271, 27245 },
    { 32768, 27245, 12539, 6392, 25170, 32198, 30271, 18204 },
    { 32768, 32198, 30271, 27245, 25170, 18204, 12539, 6392 }
};
```

```
static int FDCT(int u, int v, short img[8][8]) {
    double a[8], r = 0; int x2;
    for(x=0; x<8; x++) {
        a[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 += a[x] * COS(x, u);
    return (short)(r * .25 * C(u) * C(v));
}
```

## 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
- 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

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

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

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

void CntrlCompressImage(void) {
    for(i=0; i<ROW_ROW_BLOCKS; i++)
        for(j=0; j<COL_COL_BLOCKS; j++) {
            for(k=0; k<8; k++)
                for(l=0; l<8; l++)
                    CodePushPixel(
                        (char)buffer[i * 8 + k][j * 8 + l]);
            CodeDoFdt(); /* part 1 - FDCT */
            for(k=0; k<8; k++)
                for(l=0; l<8; l++) {
                    buffer[i * 8 + k][j * 8 + l] = CodePopPixel();
                    /* part 2 - quantization */
                    buffer[i*8+k][j*8+l] >>= 6;
                }
        }
}
```

## 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

```
int main(int argc, char *argv[]) {
    char *uartOutputFilename = argc > 1 ? argv[1] : "uart_out.txt";
    char *imageFilename = argc > 2 ? argv[2] : "image.txt";
    uartInitialize(uartOutputFilename);
    CntrlInitialize(imageFilename);
    CcdppInitialize();
    CntrlInitialize();
    /* simulate functionality */
    CntrlCaptureImage();
    CntrlCompressImage();
    CntrlSendImage();
}
```

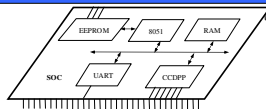
## Design

- 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: Microcontroller alone

- 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
  - CcdppCapture* 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

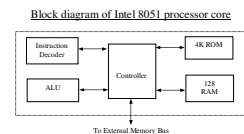
## Implementation 2: Microcontroller and CCDPP



- 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



## 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

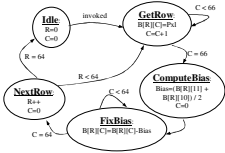
FSMD description of UART



## 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

FSMD description of CCDPP



## 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 of the hardware devices
    - $adata$  used to load/store variables over external memory bus
    - $at$  specifies memory address to store these variables
    - Byte sent to  $U\_TX\_REG$  by processor will invoke UART
    - $U\_STAT\_REG$  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

Original code from system-level model

```
#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);
}

```

Rewritten UART module

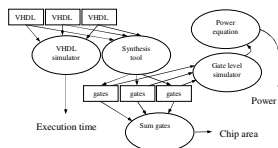
```
static unsigned char adata;
static FILE *outputFileHandle;
static unsigned char adata;
void uartInitialize(void) {
    void uartSend(unsigned char d) {
        while (U_TX_REG == 1) {}
        /* busy wait */
        U_TX_REG = d;
    }
}

```

## 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

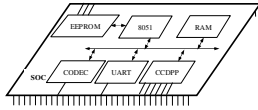


## Implementation 2: Microcontroller and CCDPP

- 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 4: Microcontroller and CCDPP/DCT



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

## Implementation 4: Microcontroller and CCDPP/DCT

- 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 2	Implementation 3	Implementation 4
Performance (seconds)	9.1	1.5	0.099
Power (watt)	0.053	0.033	0.040
Size (gates)	90,000	90,000	128,000
Energy (joule)	0.30	0.050	0.0040

- 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?

## 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 – the main lesson of this book!

## HW #2: Application Examples

- Create an embedded device and present it in class (Be creative)
  - 10 min presentation
  - Slides & Presentations are due in class next Thurs. Sept 25<sup>th</sup>
  - Provide: (similar to the summary you read on the digital camera, but with less technical detail)
    - Introduction to the product
    - Specify user requirements (size, timing, etc)
    - Specify system requirements (speed, battery, etc)
    - Discuss possible costs based on potential device volume
    - H/W ideas (which type of device? ASIC, FPGA? Why?)
    - What type of S/W components are needed
  - Simple Examples
    - MP3 Alarm clock
    - Video accelerator
    - Smart-Elevator controller
    - Digital Picture Frame