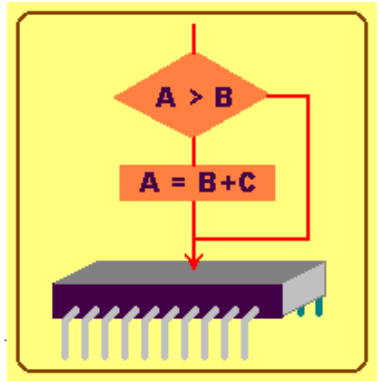
Platform Selection – Motivating Example and Case Study

Example from Embedded System Design: A Unified Hardware/Software Approach. Vahid & Givargis, 2000.

Overview

- All real systems contain both hardware and software! (no such thing has a software-only system)
- We will talk about some platform choices for systems.
  - So far we have assumed an abstract task model with timing parameters.
  - Where do those parameters come from? Can they be improved?
- What is the impact of hardware & software choices?



Overview

- In this example, we will step through a simple system and show how it can be partitioned among hardware and software
  - Four implementations, each with varying degrees of hardware
- Along the way, we will start to understand some of the tradeoffs between implementing something in hardware and implementing something in software.
- We'll also talk about fixed-point vs. floating-point, an important optimization that involves both hardware and software.


MODE


Vivitar

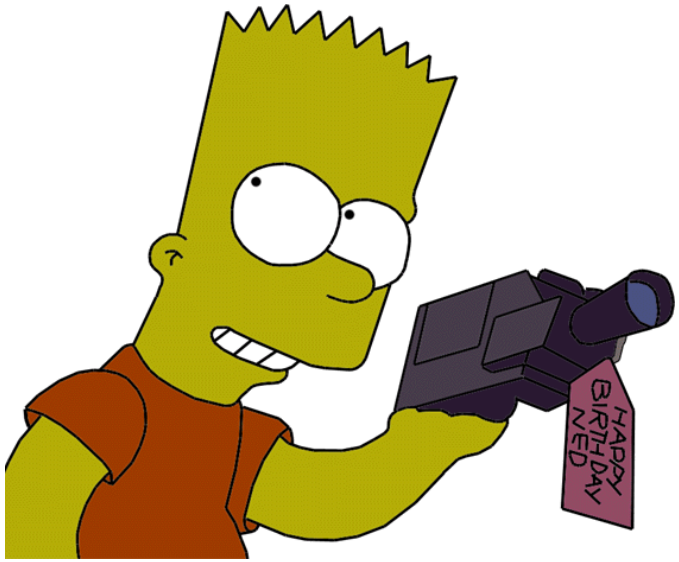

Nikon
ISO QUAL WB
POWER LINK BUSY


LUMIX
LEICA
12x

What does a digital camera do?

- Capturing images, processing them, and storing them in memory
- Uploading images to a PC

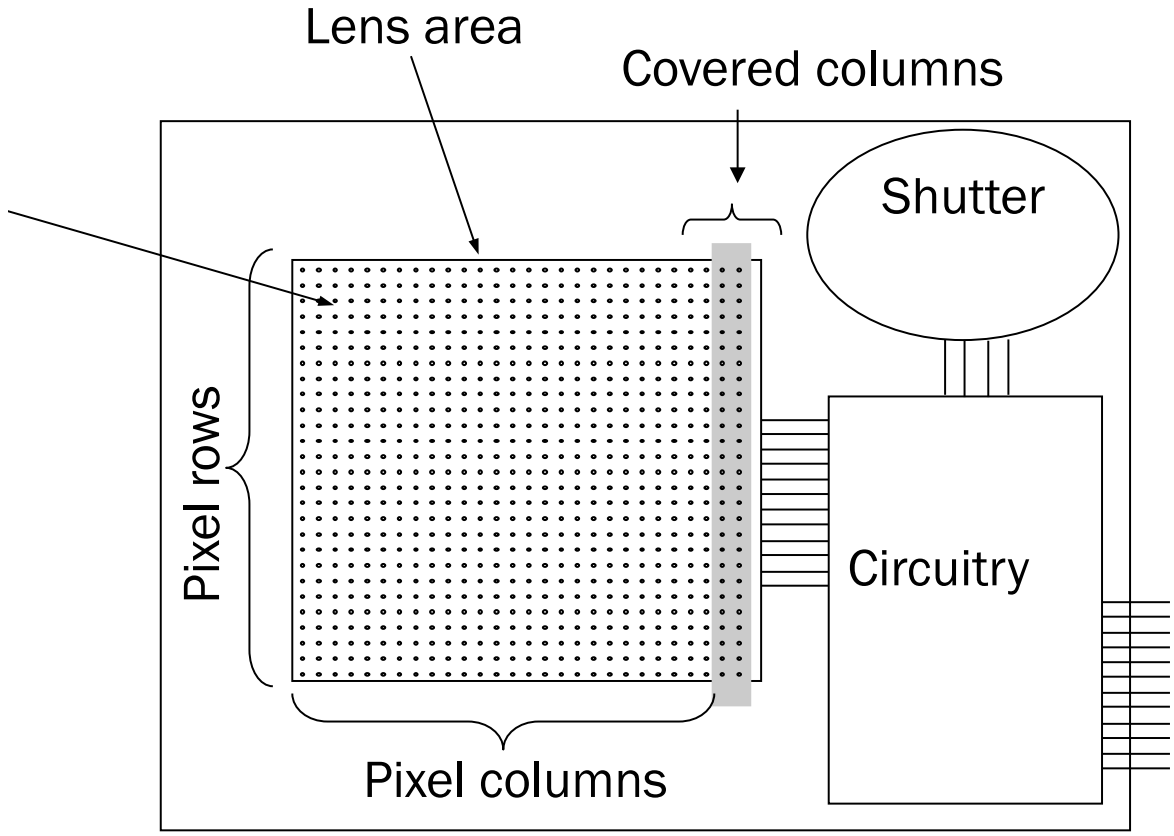
- We will focus on the first:
  - When the shutter is pressed:
    - Image captured
    - Converted to digital form by CCD
    - Compressed and archived



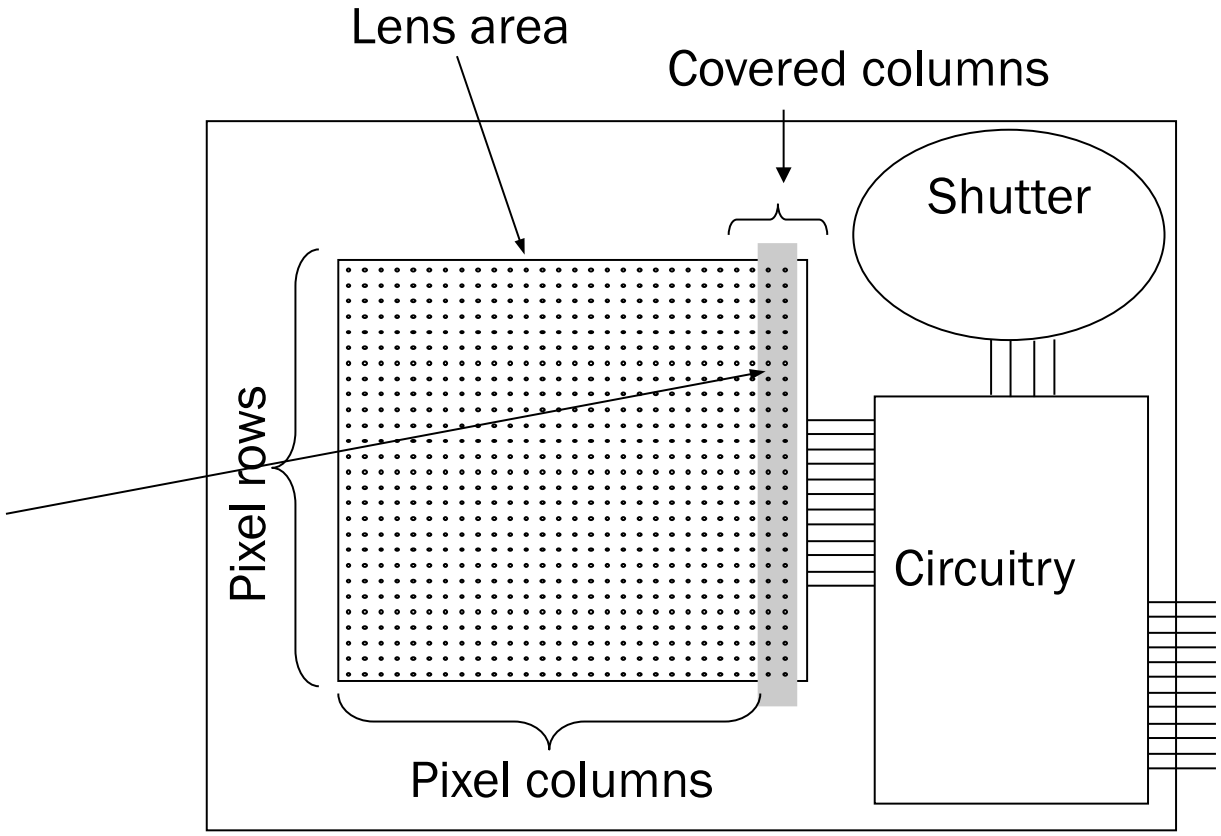
Requirements

- Performance
  - We want to process a picture in one second
    - Slower would be annoying
    - Faster not necessary for a low-end camera
- Size
  - Must fit on a low-cost chip
  - Let's say 200,000 gates, including the processor
- Power and Energy
  - We don't want a fan
  - We want the battery to last as long as possible

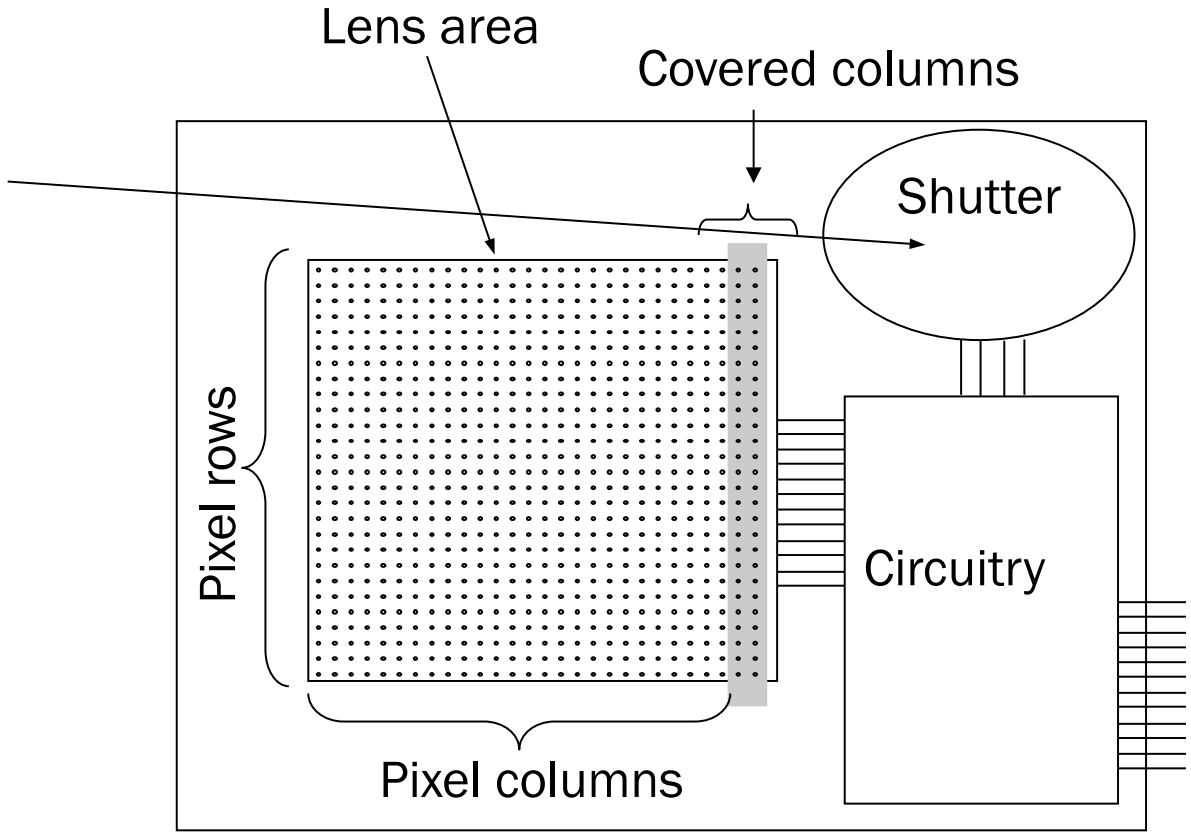
When exposed to light, each cell becomes electrically charged. This charge can then be converted to a 8-bit value where 0 represents no exposure while 255 represents very intense exposure of that cell to light.



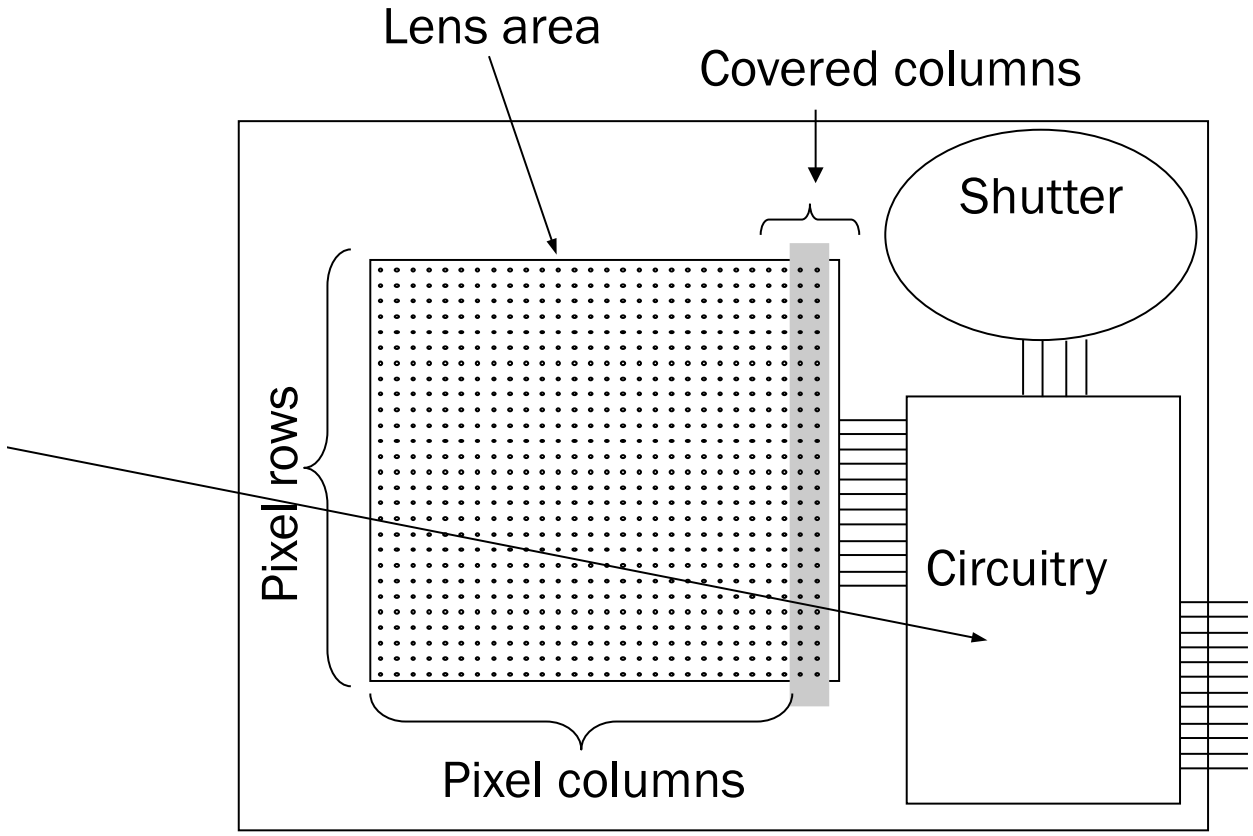
Some of the columns are covered with a black strip of paint. The light-intensity of these pixels is used for zero-bias adjustments of all the cells.

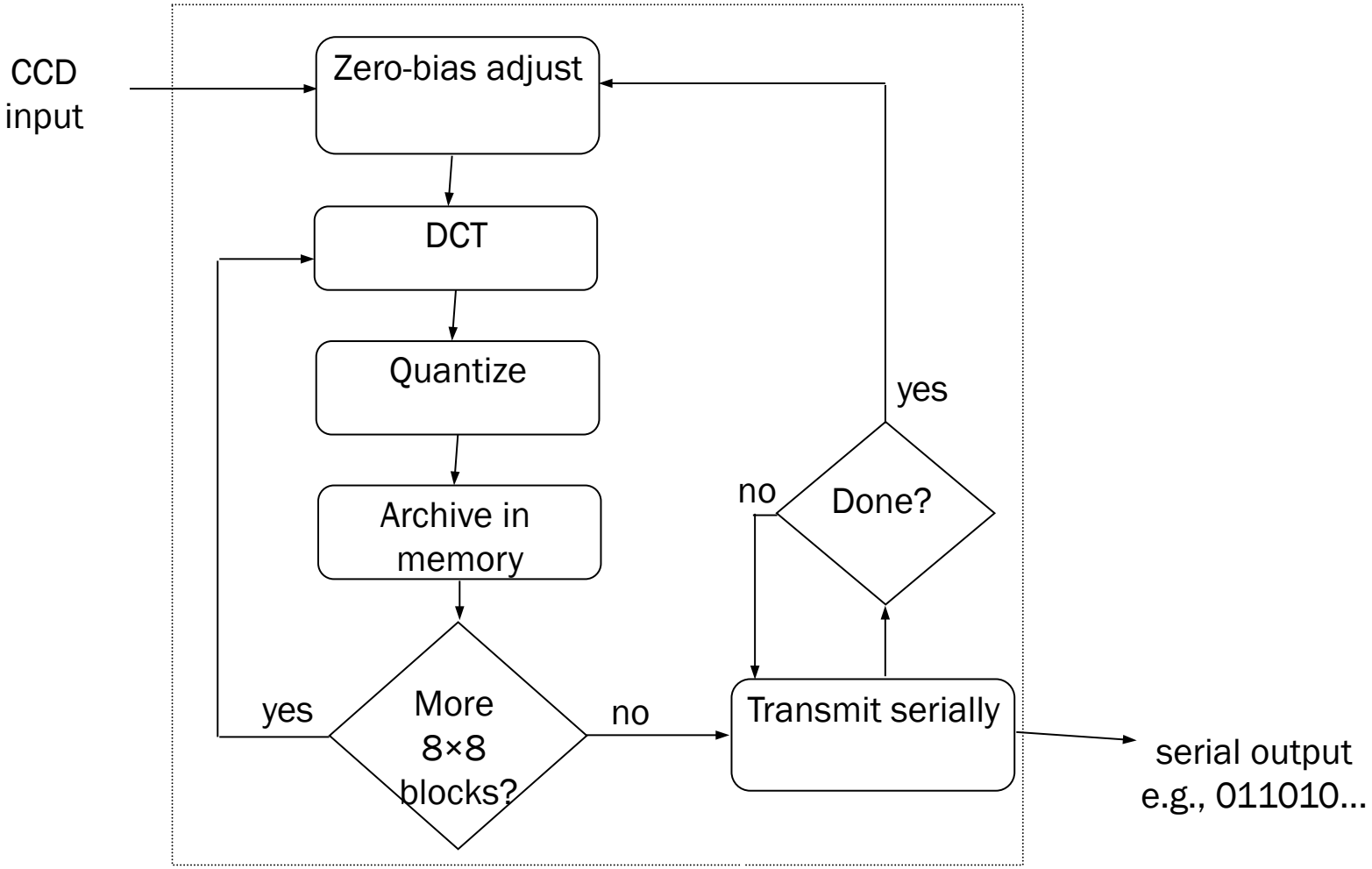


The electromechanical shutter is activated to expose the cells to light for a brief moment.



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.




CCD
input
Zero-bias adjust
DCT
Quantize
Archive in
memory
More
8×8
blocks?
yes
no
Transmit serially
Done?
no
yes
serial output
e.g., 011010…

Zero-Bias Error

- Manufacturing errors cause cells to measure slightly above or slightly below the actual light intensity
- Error typically the same across columns, but different across rows
- Some of the left-most columns are blocked by black paint
  - If you get anything but 0, you have a zero-bias error
  - Each row is corrected by subtracting the average error in all the blocked cells for that row

Zero-Bias Error

Covered cells

Zero-bias adjustment

136	170	155	140	144	115	112	248	**12**	**14**	**-13**
145	146	168	123	120	117	119	147	**12**	**10**	**-11**
144	153	168	117	121	127	118	135	**9**	**9**	**-9**
176	183	161	111	186	130	132	133	**0**	**0**	**0**
144	156	161	133	192	153	138	139	**7**	**7**	**-7**
122	131	128	147	206	151	131	127	**2**	**0**	**-1**
121	155	164	185	254	165	138	129	**4**	**4**	**-4**
173	175	176	183	188	184	117	129	**5**	**5**	**-5**

Before zero-bias adjustment

123	157	142	127	131	102	99	235
134	135	157	112	109	106	108	136
135	144	159	108	112	118	109	126
176	183	161	111	186	130	132	133
137	149	154	126	185	146	131	132
121	130	127	146	205	150	130	126
117	151	160	181	250	161	134	125
168	170	171	178	183	179	112	124

After zero-bias adjustment

CCD Pre-Processing Module

Performs zero-bias adjustment

CcdppCapture uses *CcdCapture* and *CcdPopPixel* to obtain image

Performs zero-bias adjustment after each row read in

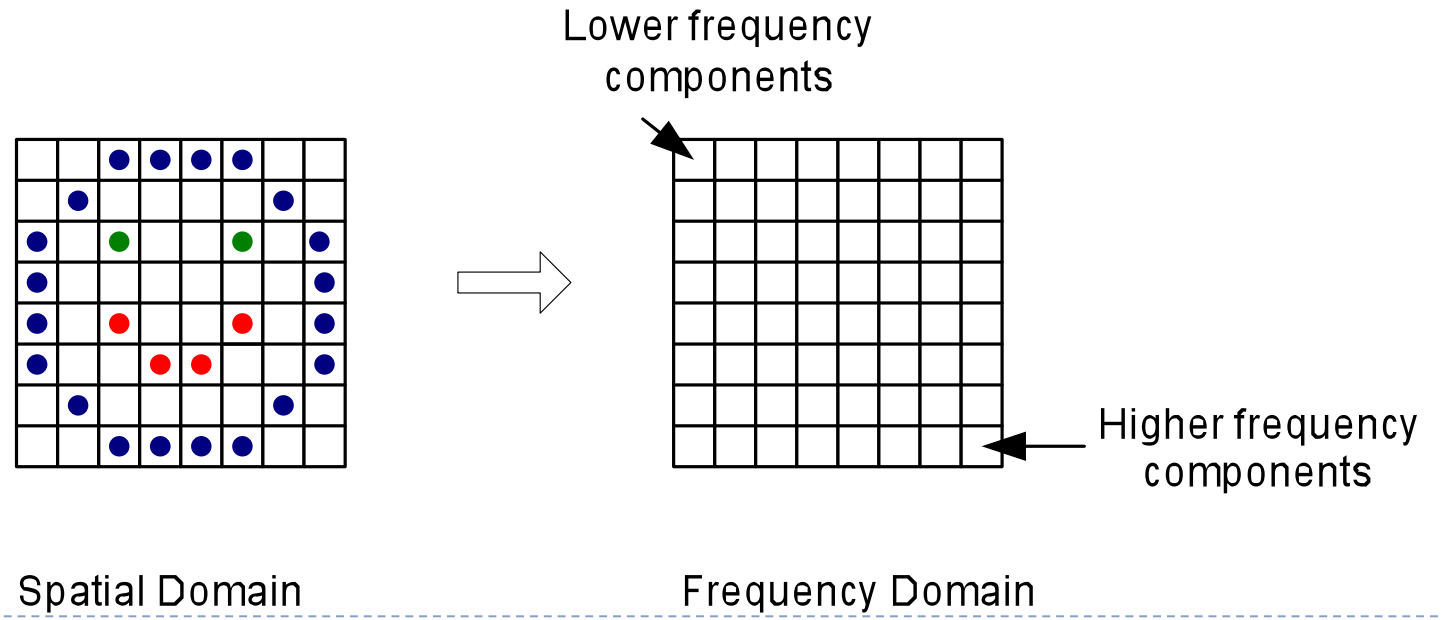
```
void CcdppCapture(void) {
    CcdCapture();
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] = CcdPopPixel();
        }
        bias = (CcdPopPixel() + CcdPopPixel()) / 2;
        for(colIndex=0; colIndex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

Compression

- JPEG Compression
  - Based on discrete cosine transform (DCT)
  - Image data is divided into 8x8 blocks of pixels
  - On each block, do the following:
    - DCT
    - Quantization
    - Huffman Encoding

Discrete Cosine Transform (DCT)

- Transforms an 8x8 block of pixels into the frequency domain
  - We produce a new 8x8 block of values such that
    - Upper-left corner represent the “low frequency” components
    - Lower-right corner represents the “high frequency” components
    - We can reduce the precision of the higher frequency components and retain reasonable image quality



Discrete Cosine Transform (DCT)

- Equation to perform DCT:

$$F(u,v) = \frac{1}{4} * C(u) * C(v) * \sum_{x=0..7} \sum_{y=0..7} D_{xy} * \cos(\frac{\pi(2x+1)u}{16}) * \cos(\frac{\pi(2y+1)v}{16})$$

- where $C(h) = \begin{cases} 1/\sqrt{2} & \text{if h = 0} \\ 1 & \text{otherwise} \end{cases}$

Quantization

- Reduce bit precision. In our case, let's reduce the precision
  - equally across all frequency values

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

After being decoded using DCT

Divide each cell's value by 8

144	5	-5	-1	3	-10	1	5
-10	0	14	-9	-1	0	3	-1
2	-1	0	-5	3	0	2	-5
0	-8	-2	-2	5	-3	-2	1
6	2	5	-1	1	-3	1	-1
5	-1	-1	-1	3	-4	-3	2
-2	-1	3	-1	0	0	2	-3
-1	-2	-1	-2	-1	0	1	-1

After quantization

CODEC

```
void CodecDoFdct(void) {
    int i, j;
    for(i=0; i<NUM_ROW_BLOCKS; i++)
        for(j=0; j<NUM_COL_BLOCKS; j++)
            CodecDoFdct_for_one_block(i, j);
}
```

```
void CodecDoFdct_for_one_block(int i, int j) {
    int x, y;
    for(x=0; x<8; x++)
        for(y=0; y<8; y++)
            obuffer[i*8+x][j*8+y] = FDCT(i, j, x, y, ibuffer);
}
```

Aside: Fixed-Point Number Representation

- Rather than computing the floating point cosine function

$$F(u,v) = \frac{1}{4} * C(u) * C(v) * \sum_{x=0..7} \sum_{y=0..7} D_{xy} * \cos(\frac{\pi(2x+1)u}{16}) * \cos(\frac{\pi(2y+1)v}{16})$$

  - Notice that there are only 64 distinct values need for the cosine
  - So let's pre-compute them

Aside: Fixed-Point Number Representation

- The result of the cosine is floating point
  - It would be better if we could store the table in less memory
- Example: Suppose we want to represent -1 to 1 using 16 bits

Floating Point	Fixed Point
0	0
0.25	8192
0.5	16384
0.999999…	32767
-0.5	-16384

- So if x is the floating point number, the fixed point number is **round (32768 * x)**

CODEC

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

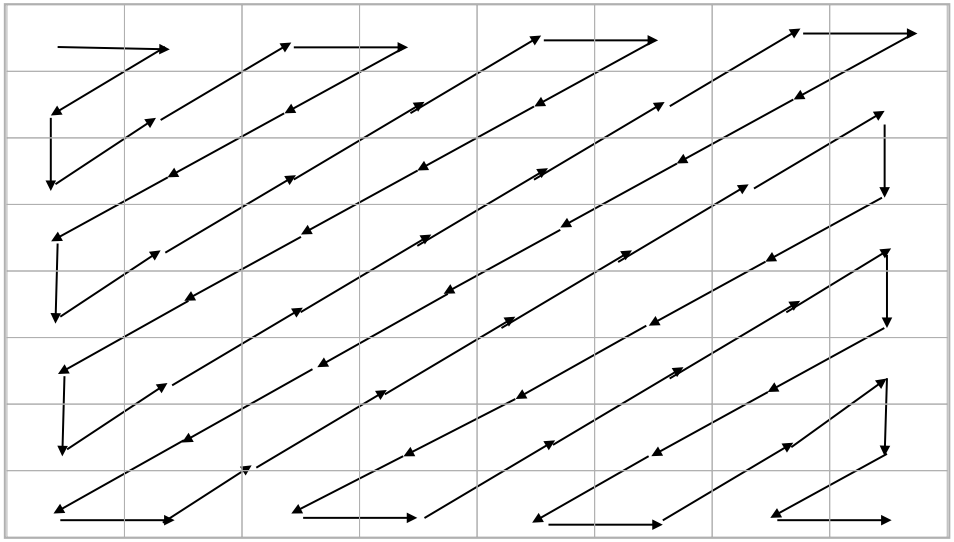
```
static double COS(int xy, int uv) {
      return( COS_TABLE[xy][uv] / 32768.0);
}
```

CODEC

```
static int FDCT(int base_x, base_y, offset_x, offset_y, short **img) {
    r = 0;
    u = base_x*8 + offset_x;
    v = base_y*8 + offset_y;
    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 (r * .25 * C(u) * C(v));
}
```

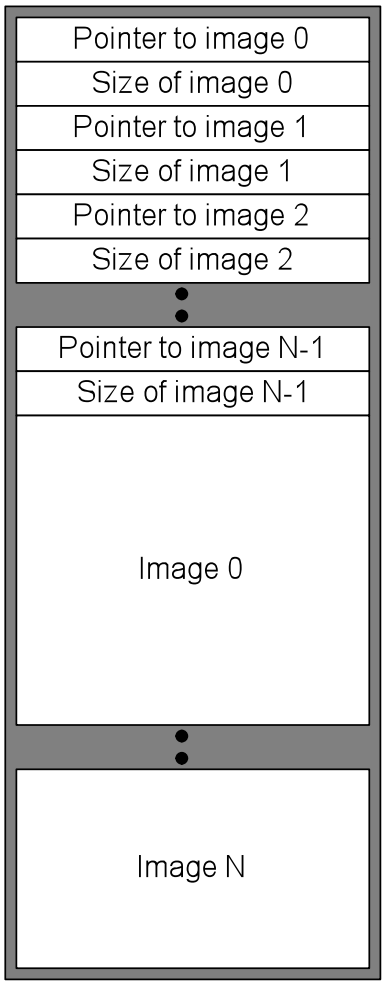
Huffman Encoding

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



- Perform Huffman encoding (self-study)
  - More frequently occurring pixels assigned short binary code
  - Longer binary codes left for less frequently occurring pixels

Archiving Images

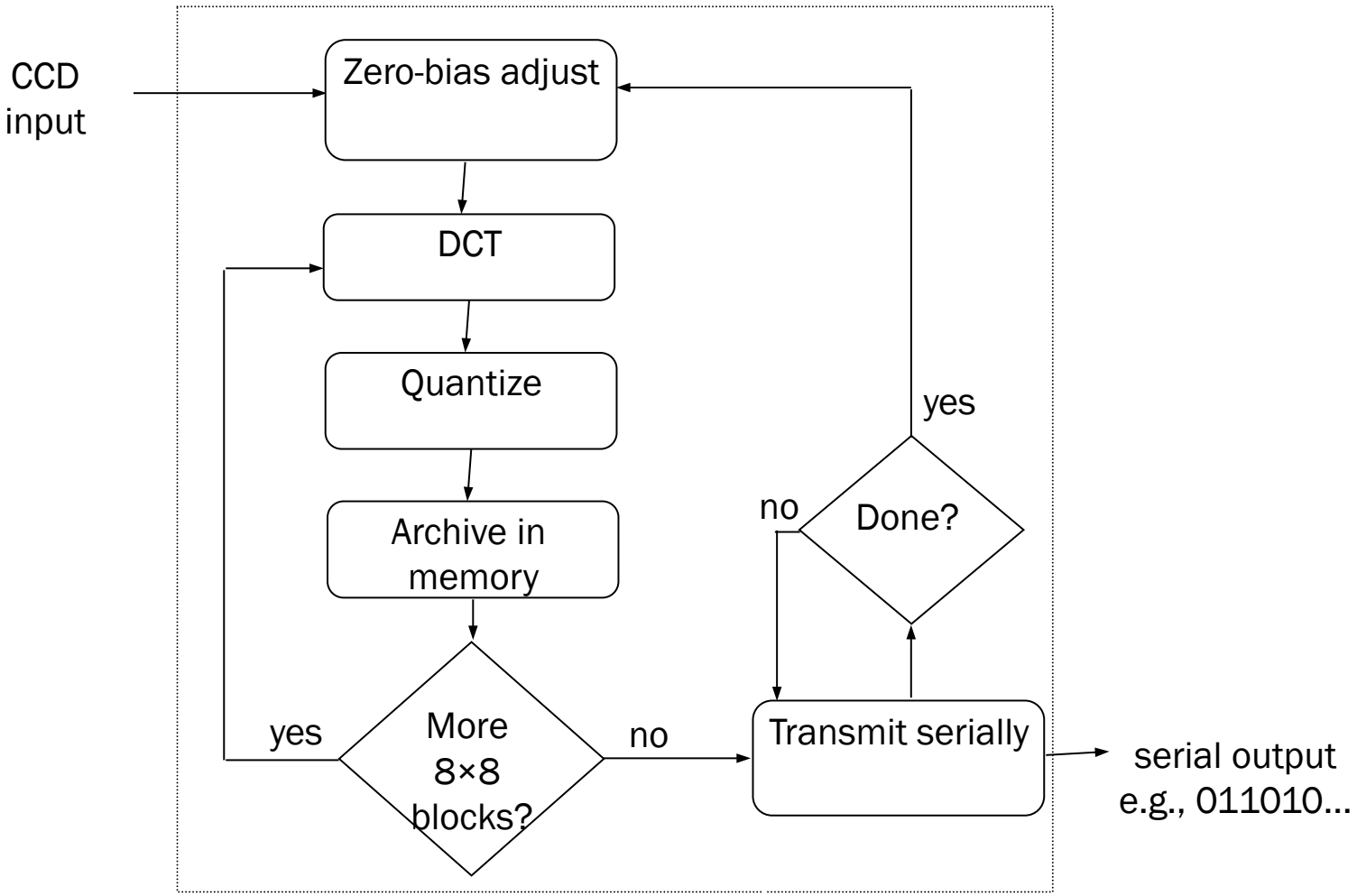


Here's a really simple memory map

The amount of memory required depends on N and the compression ratio

Uploading to the 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

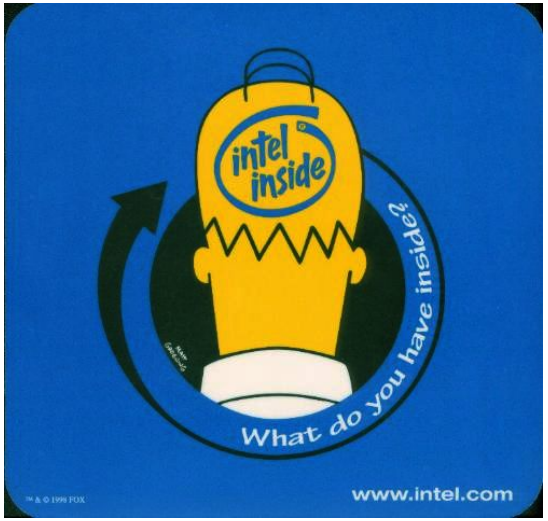

CCD
input
Zero-bias adjust
DCT
Quantize
Archive in
memory
More
8×8
blocks?
yes
no
Done?
no
yes
Transmit serially
serial output
e.g., 011010...

Implementing a simple digital camera

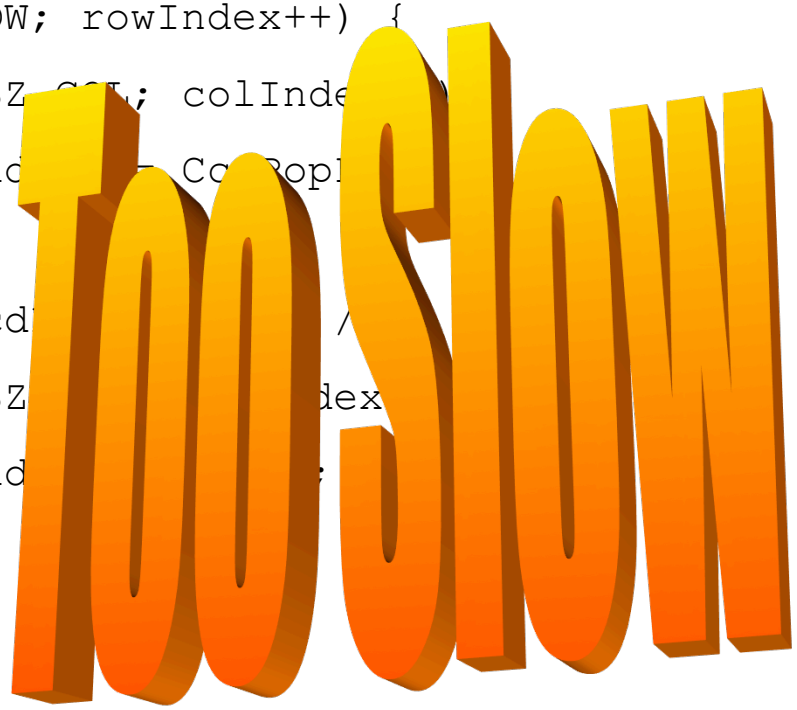
- We are going to talk about four potential implementations
  - Microcontroller Alone (everything in software)
  - Microcontroller and CCDPP
  - Microcontroller and CCDPP/Fixed-Point DCT
  - Microcontroller and CCDPP/DCT

Implementation 1: Microprocessor Alone

- Suppose we use an Intel 8051 Microcontroller
  - Total IC Cost about $5
  - Well below 200mW power
  - We figure it will take 3 months to get the product done
  - 12 Mhz, 12 cycles per instruction
    - one million instructions per sec
  - Can we get the required performance?
    - let's say our grid is 64x64



```
void CcdppCapture(void) {
    CcdCapture();
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colInde
            buffer[rowIndex][colInd
        }
        bias = (CcdPopPixel() + Ccd
        for(colIndex=0; colIndex<SZ
            buffer[rowIndex][colInd
        }
    }
}
```



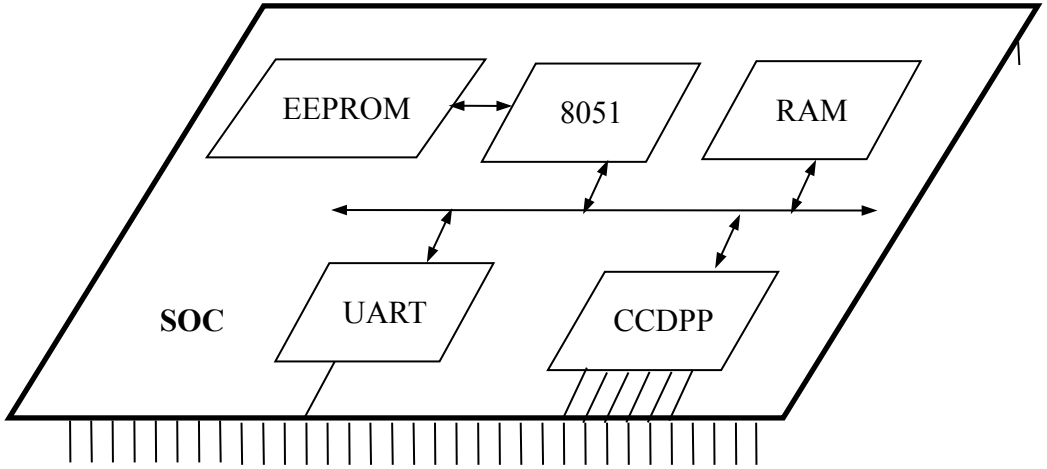
Nested loops, 64*(64+64) iterations.

If each iteration is 50 assembly language instructions,

8192 * 50 instructions = 409,600 instructions per image

This is half our budget and we haven't even done DCT or Huffman yet!

Implementation 2: Microcontroller and CCDPP



- CCDPP function implemented on custom hardware unit
  - Improves performance – less microcontroller cycles
  - Increases engineering cost and time-to-market
  - Easy to implement
    - Simple datapath
    - Few states in controller
- Simple UART easy to implement as custom hardware unit 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

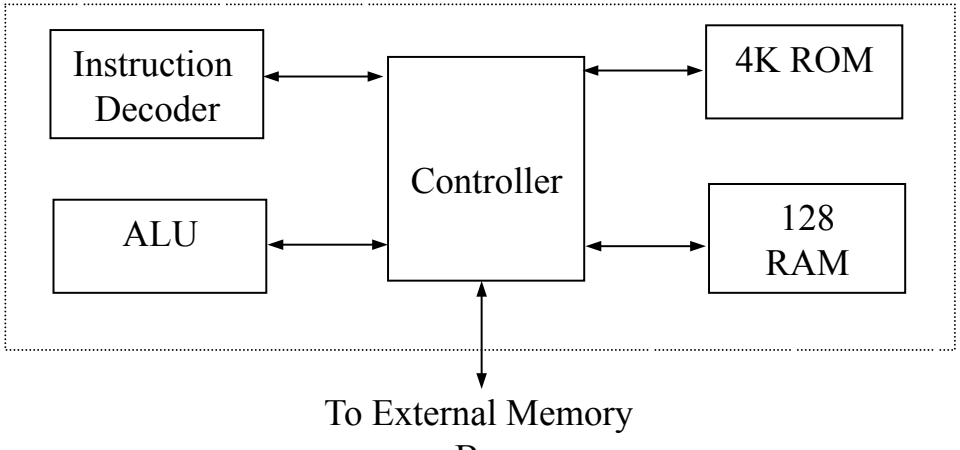
ALU executes arithmetic operations

- Source and destination registers reside in RAM

Special data movement instructions used to load and store externally

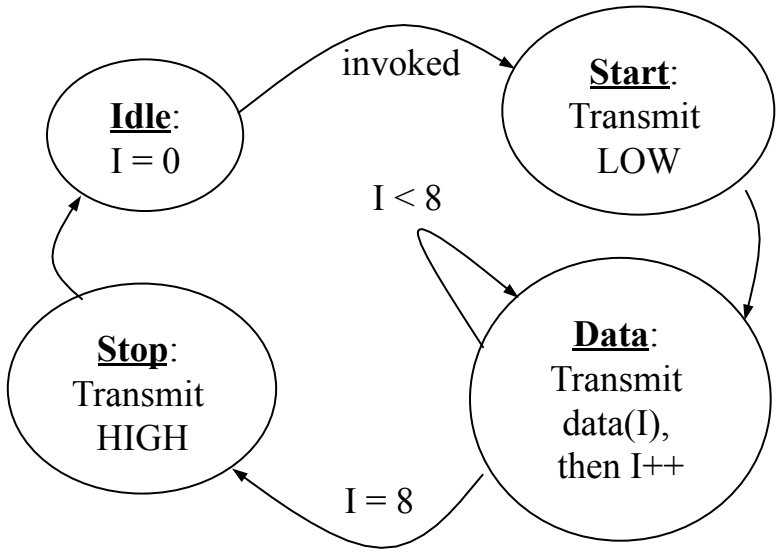
Special program generates VHDL description of ROM from output of C compiler

Block diagram of Intel 8051 processor core



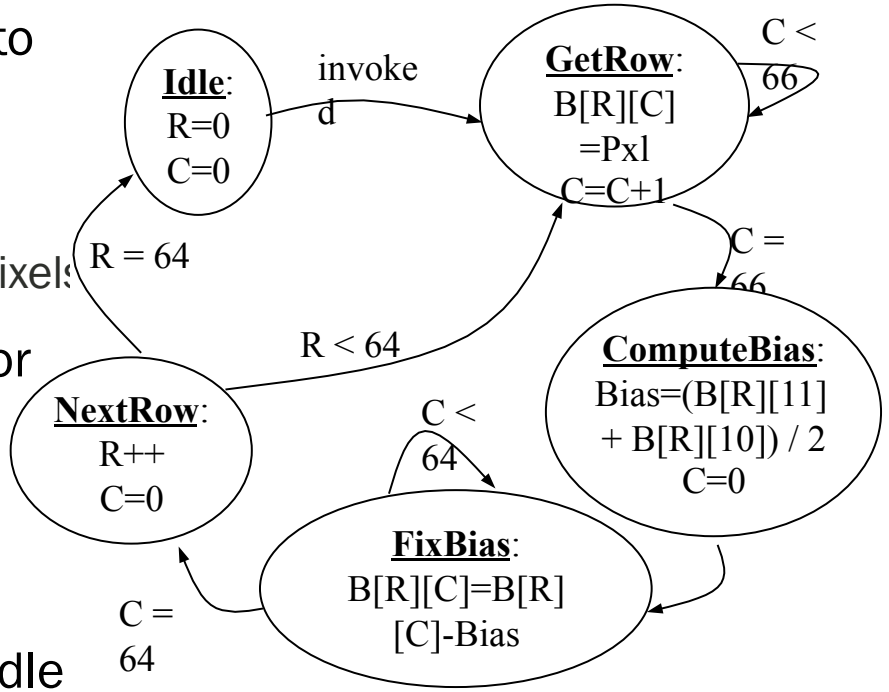
UART

- UART invoked when 8051 executes store instruction with UART's enable register as target address
  - Memory-mapped communication between 8051 and UART
- 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

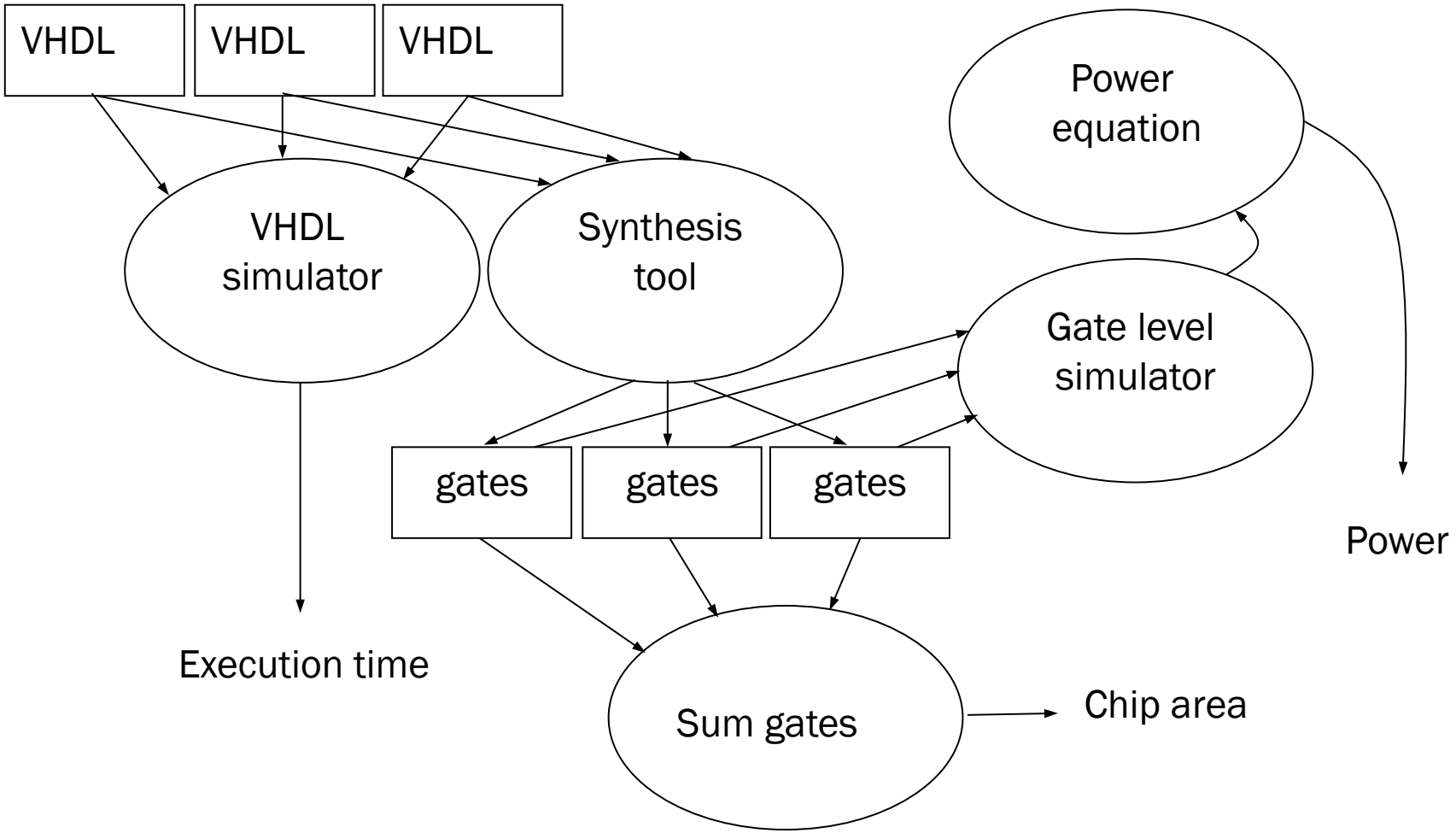
- Hardware implementation of zero-bias operations
- Internal buffer, *B*, memory-mapped to 8051
- *GetRow* state reads in one row from CCD to *B*
  - 66 bytes: 64 pixels + 2 blacked-out pixels
- *ComputeBias* state computes bias for that row and stores in variable *Bias*
- *FixBias* state iterates over 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 custom circuit) 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 custom circuit) detects asserted write control signal
  - Checks address bus
  - Reads and stores data from data bus

Analysis


VHDL
VHDL
VHDL
Power equation
VHDL simulator
Synthesis tool
Gate level simulator
gates
gates
gates
Power
Execution time
Sum gates
Chip area

Analysis

- Entire SOC tested on VHDL simulator
  - Interprets VHDL descriptions and functionally simulates execution of system
  - Tests for correct functionality
  - Measures clock cycles to process one image (performance)

- Gate-level description obtained through synthesis
  - Synthesis tool like compiler for hardware
  - 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

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: Fixed-Point DCT

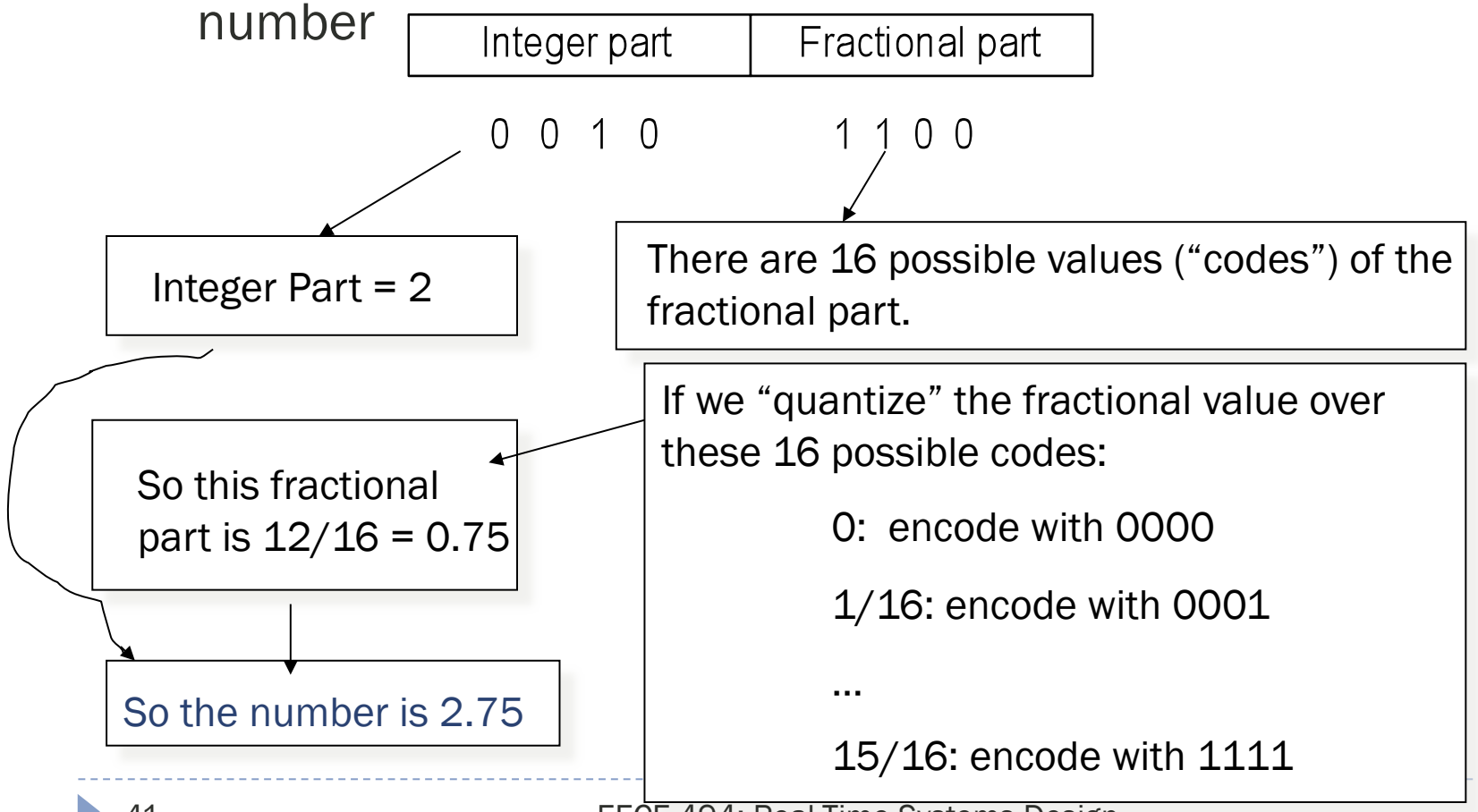
- Most of the execution time is spent in the DCT
- We could design custom hardware like we did for CCDPP
  - More complex, so more design effort
- Let's see if we can speed up the DCT by modifying the number representation (but still do it in software)

DCT 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
  - Some bits represent fraction, some bits represent whole number



Fixed-Point Arithmetic

- Addition
  - A good approximation is to simply add the fixed-point representations

- Example: Suppose we want to add 3.14 and 2.71
- 3.14 is represented as 00110010
- 2.71 is represented as 00101011

- Add these two representations to get: 01011101
- This corresponds to 5.8125, which is kind of close to 5.85

Fixed-Point Arithmetic

- Multiply
  - Multiply the representations and shift right by the number of bits in the fractional part
- Example: Suppose we want to multiply 3.14 and 2.71
- 3.14 is represented as 00110010
- 2.71 is represented as 00101011
- Multiply these two representations to get: 100001100110
- Shift right by 4 bits: 10000110
- This corresponds to 8.375, which is kind of close to 8.5094
- Moral: we can add and multiply easily. This is faster and smaller than floating point.

New CODEC

```
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,   35,  -24,  -62,  -45,   12,   59,   53 },
    {   64,   12,  -59,  -35,   45,   53,  -24,  -62 },
    {   64,  -12,  -59,   35,   45,  -53,  -24,   62 },
    {   64,  -35,  -24,   62,  -45,  -12,   59,  -53 },
    {   64,  -53,   24,   12,  -45,   62,  -59,   35 },
    {   64,  -62,   59,  -53,   45,  -35,   24,  -12 }
};
```

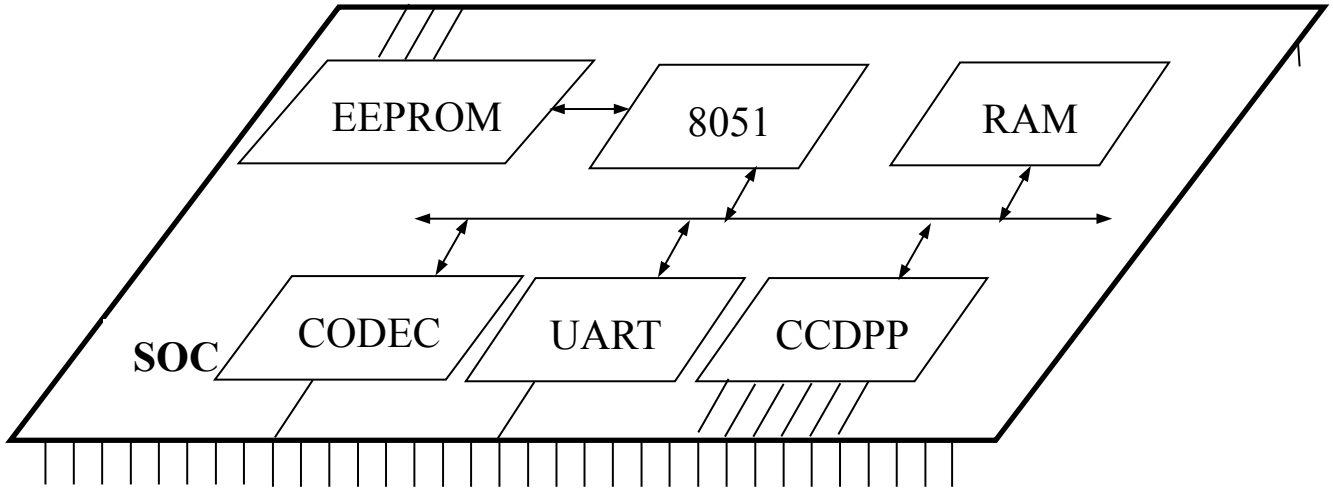
```
static int FDCT(int base_x, base_y, offset_x, offset_y, short **img) {
    r = 0;
    u = base_x*8 + offset_x;
    v = base_y*8 + offset_y;
    for (x=0; x<8; x++) {
        s[x] = 0;
        for(j=0; j<8; j++)
            s[x] += (img[x][j] * COS_TABLE[j][v] ) >> 6;
    }
    for(x=0; x<8; x++) r += (s[x] * COS_TABLE[x][u]) >> 6;
    return (short)((((r * (((16*C(u)) >> 6) *C(v)) >> 6)) >> 6) >> 6);
}
```

Analysis of Implementation 3

- Total execution time for processing one image:
  - 1.5 seconds
- Power consumption:
  - 0.033 watt (same as implementation 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 fewer gates – less memory needed for code)

Implementation 4: Implement the CODEC in H/W



- The CODEC block will be specially-designed hardware to perform the DCT on one 8x8 block
  - Still need software to control the whole thing

CODEC Design

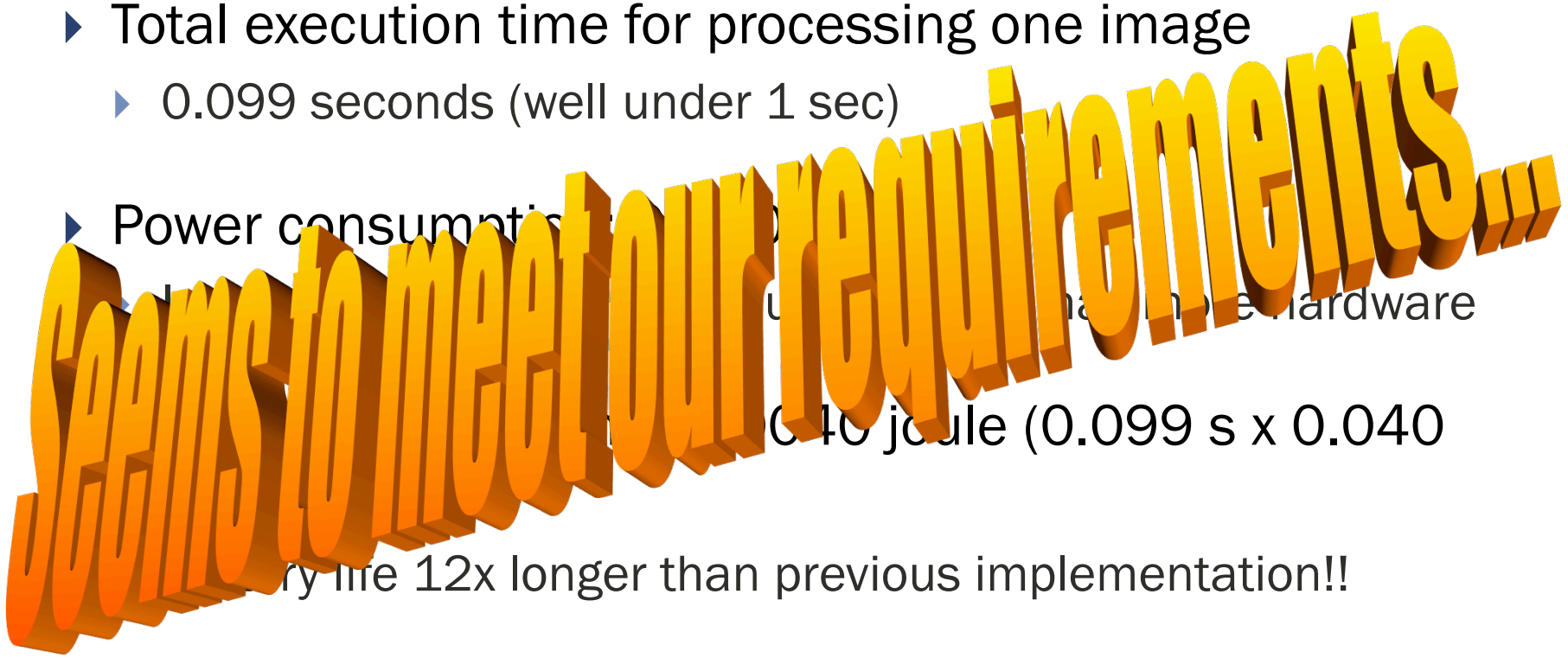
- Four memory mapped registers
  - C_DATAI_REG: used to push 8x8 block into CODEC
  - C_DATAO_REG: used to pop 8 x 8 block out of CODEC
  - C_CMND_REG: used to command CODEC
    - Writing 1 to this register invokes CODEC
  - C_STAT_REG: indicates CODEC done and ready for next block
    - Polled in software
- Direct translation of C code to VHDL for actual hardware implementation.
- Fixed-point version used

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 the chip has more hardware
- 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

Analysis of Implementation 4

- Total execution time for processing one image
  - 0.099 seconds (well under 1 sec)
- Power consumption
  - Using Xilinx hardware
  - 0.040 joule (0.099 s x 0.040
  - Battery life 12x longer than previous implementation!!
- Total chip area: 128,000 gates
  - Significant increase over previous implementations



So, what do you tell your boss?

	Implementation 2	Implementation 3	Implementation 4
Performance (second)	9.1	1.5	0.099
Power (watt)	0.033	0.033	0.040
Size (gate)	98,000	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 engineering cost and time-to-market
    - If existing DCT purchased then increased IC cost
- **Which is better?**

Highlights

- We saw an example/case study that illustrates some of the tradeoffs
  - Hardware takes longer to design, but will be faster
  - Sometimes you can optimize the software instead
  - Always a tradeoff between performance, cost, and time

- What you should know
  - Not all details equally important
  - Understand how to make trade-offs
  - Floating point vs. fixed point
  - Other details will be provided for you to analyze (IC cost, etc.)