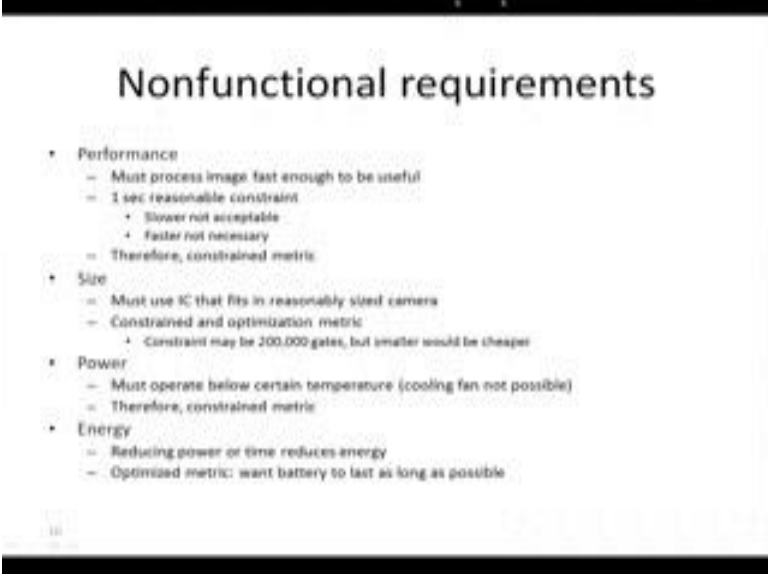


Embedded Systems Design
Prof. Anupam Basu
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture- 44
Digital Camera – Iterative Design

(Refer Slide Time: 00:32)

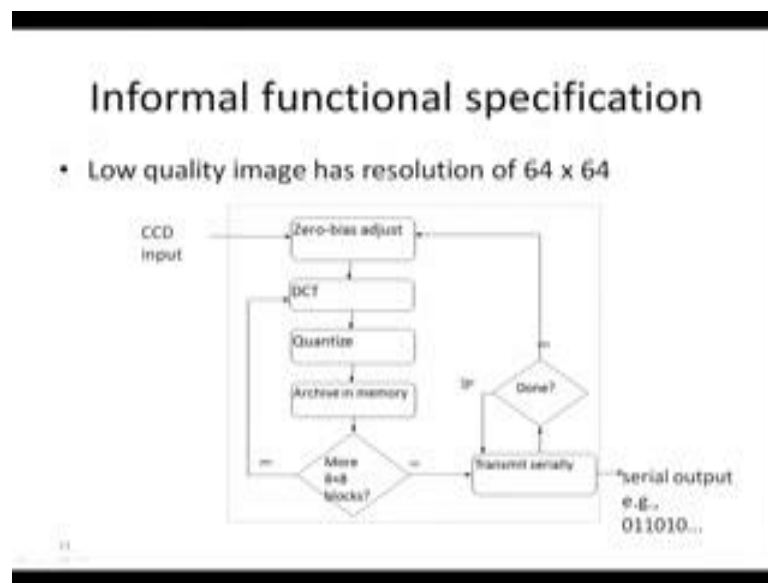


Nonfunctional requirements

- Performance
 - Must process image fast enough to be useful
 - 1 sec. reasonable constraint
 - Slower not acceptable
 - Faster not necessary
 - 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

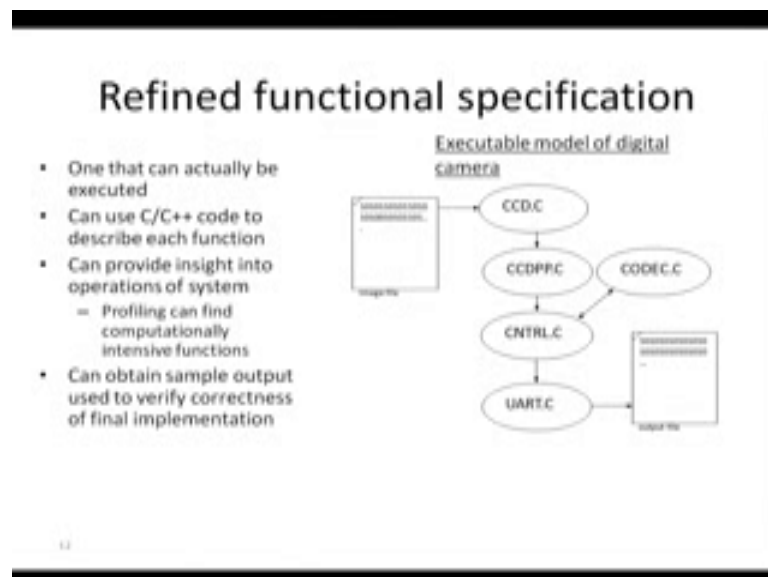
So, in the last lecture, we had seen the basic tasks that are required to be performed by a digital candidate. And we also mentioned about the nonfunctional requirements, I once again repeat those for the sake of completion. One is that the images must be captured and processed within 1 second that is reasonable constraint, and that is a constrained metric. We must optimize on the number of gates, so around 200,000 gates will be there. A smaller solution is preferable, the power should be less as this as possible and the same is for energy.

(Refer Slide Time: 01:17)



And here we saw some informal way of functional specification where we mention these steps that zero-bias adjustment, then discrete cosine transforms, then quantization then archival in the memory, and that 64 by 64 image we take into 8 by 8 blocks, so we repeat this 64 times.

(Refer Slide Time: 01:57)



Now, that was the informal specification. Now, we have to further refine this specification that can be actually executed. What we are showing here all these tasks taking the CCD input, preprocessing it, doing the compression part and encoding and

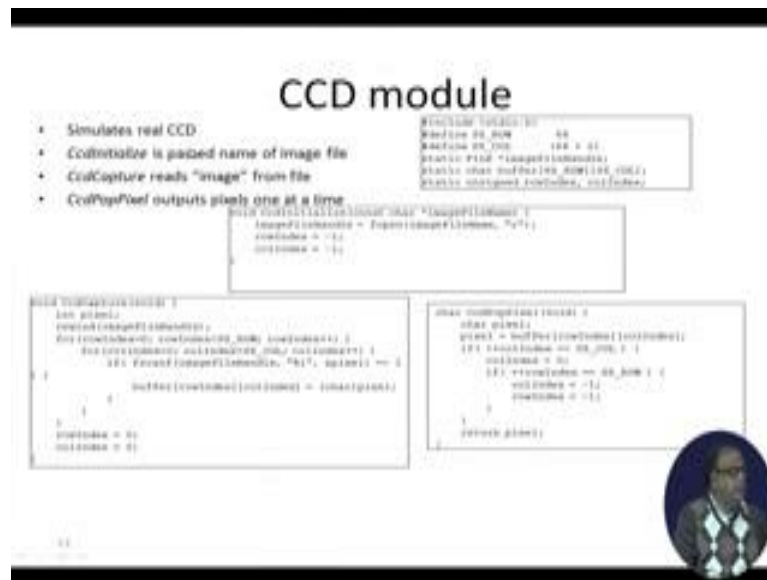
then storing it in memory. All these things we need to test and here we can specify it in a much more refined way, where we can put an executable model of the digital camera. So, as if I am actually running the digital camera on software.

So, here I will have a file that will capture the image, image that the CCD is coming from the lens and is coming on the CCD. And whatever CCD does I can write a CCD dot c program, for that here everything is we are using C or C++ code to describe every function and that will provide us the insight into the system. Once you do that I can do the profiling and see which part is taking more time as I execute this bunch of c codes, I will see what is where the hotspots are which one is taking more time which one is taking less time.

Next, we come to another code module which is CCD preprocessing, preprocessing of CCD. We have seen what we preprocessing consists of, preprocessing consists of zero bias adjustment and then we will have to do the here is another model called codec where we are doing the compression. And for the compression, we are doing DEC dot C. Now, there is a controller module which is actually controlling the invocation of all these c programs. And once this program is done once this steps are done then we can come to this another module software module called UART dot c.

What is UART? Universal asynchronous receiver transmission that is for sort of some serial communication, through that we can send it through some output file to (Refer Time: 04:28) and that is the overall functional specification. Now, once we get this functional specification, now let us look at the individual modules, I did not go into the details of the c code because you know the function it is already.

(Refer Slide Time: 04:48)



So, what happens in this CCD dot C, it is simulating the real CCD. It has got three parts three other sub functions; one is CCD initialize, which one is accepting the image file id file handler and the row and column indexes are initialized. Then comes the CCD capture, CCD capture is reading the image from the file. So, it is rewind image file handle and then for rows up to the size of the rows. So, for us it will be the image will be 64 by 64.

An internal loop up to column 64, we are scanning the image file reading the image file pixel by pixel. So, because the light was incident on the different pixels and depending on the image, different intense it is went our different pixels. So, we take that and we create a buffer. We are writing that pixel on the buffer that is the capturing the image in the buffer. Then there is another function pop pixel that means, I am popping each pixel popping the pixels here pixel is getting the buffer row index that that one that is being captured by CCD capture. So, each pixel is being pop and if and accordingly the row index and column index serving modified, so that is the CCD module what is being done.

(Refer Slide Time: 06:39)


CCDPP (CCD PreProcessing) 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) {
    char *bias;
    CcdppPixel();
    for (int row = 0; row < ROW; row++) {
        for (int col = 0; col < COL; col++) {
            buffer[row][col] = CcdPopPixel();
        }
        bias = (CcdppPixel() + CcdppPixel()) / 2;
        for (int col = 0; col < COL; col++) {
            buffer[row][col] -= bias;
        }
        row++;
        col = 0;
    }
}
```

```
void CcdppPixel(void) {
    row = 0;
    col = 0;
}

char CcdPopPixel(void) {
    char pixel;
    pixel = buffer[row][col];
    if (row == ROW - 1)
        col++;
    if (col == COL)
        row++;
    return pixel;
}
```



Next in the flow is the preprocessing module. In the preprocessing module, we will perform the zero bias adjustment. CCD capture the one that I am showing again here, uses here is CCDPP capture that means, preprocessing you are doing here what we are doing here for every row and column we are taking the pop pixel output. So, I will get that pixels now the top pixel output and then I am adjusting the buffer. How I am adjusting the buffer, I am taking the values of the last two columns of those and dividing that by two that is my bias and that bias I am subtracting from the row and column index that the one that I explained in the last lecture. So thus I am doing the preprocessing stage. So, again I am doing the pop pixel when I get the pixel I am doing the remaining tasks here. So, we have done the CCD capture then we have done the CCD pop pixel sorry CCD preprocessing. Once the preprocessing is done what is the next step that we should follow, the next step would be the compression part compression part.

(Refer Slide Time: 08:18)

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*

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

So, I am just keeping the UART module here. And the compression part is being done by the codec module. It is modeling the forward discrete cosine transform encoding. The *ibuffer*, there are two buffers, *ibuffer* is holding an 8 by 8 block, *obuffer* is holding the 8 by 8 block after compression. *Codec push pixel* called 64 times will come to that here. So, here therefore, one is just the declaration *ibuffer* 8 by 8, *obuffer* is 8 by 8. Then *codec initialize* is making the index to be 0. Now, *codec push pixel*, what it is doing is taking in the *ibuffer* 8 by 8, first 8 and it sending it over here. Now, what is it doing, this *codecdoFDCT* is forward discrete cosine transform. I am not; so here its calling it is taking x, y if you recall in that discrete cosine transform expression, we had the intensity as d x, y and that was being taken and then that was being cosine transform.

So, here we are taking the x and y values for each pixel from the *ibuffer*, and doing FDCT on that right for where discrete cosine transform and putting that in the output buffer. And then the *codec pop pixel* is again putting that on the output buffer from the output buffer to the pixel. So, *codec push pixel* is called this one 64 times to fill buffer with original block 8 by 8, and that will go on. *Codec do FDCT* will be called once to transform the 8 by 8 block because that is being done here in the loop to transform. And then *coded pop pixel* is called 64 times to retrieve pixel by pixel.

(Refer Slide Time: 10:48)

CODEC (cont.)

- Implementing FDCT formula
$$C(u) = \begin{cases} 1 & \text{if } u = 0 \\ \frac{1}{\sqrt{2}} & \text{else} \end{cases}$$
$$F(u,v) = \frac{1}{4} \times C(u) \times C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\left(\frac{(2x+1)u\pi}{2N}\right) \cos\left(\frac{(2y+1)v\pi}{2N}\right)$$
- 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
- FDCT unrolls inner loop of summation, implements outer summation as two consecutive for loops

```
static const short COS_TABLE[N][N] = {
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678
};

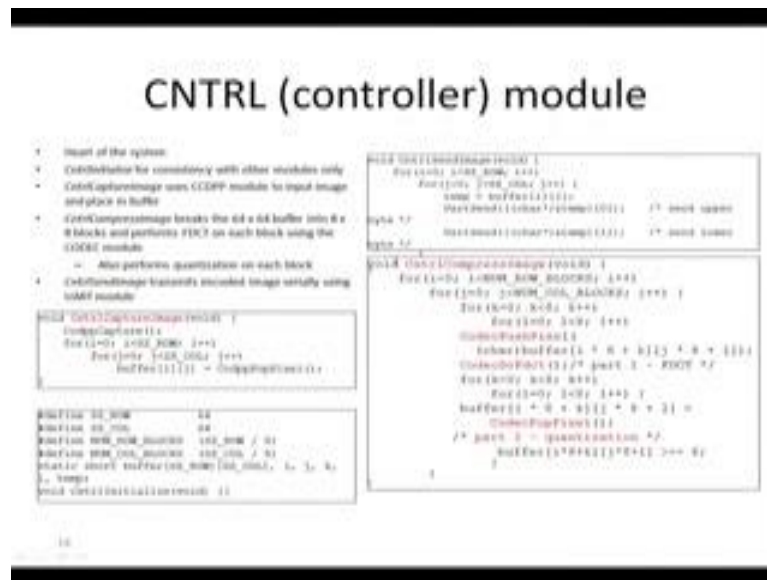
static const short COS_TABLE_90[N][N] = {
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678,
0 32678, 32678, 32678, 32678, 32678, 32678, 32678, 32678
};

static void fdct(short *f, int N) {
double *a[8], *v = f; int i, j;
for(i=0; i<8; i++) {
a[i] = f;
for(j=0; j<8; j++) {
a[i][j] = a[i][j] * COS_TABLE[j][j] / 32678.0;
}
}
for(i=0; i<8; i++) {
for(j=0; j<8; j++) {
a[i][j] = a[i][j] * COS_TABLE[i][i] / 32678.0;
}
}
return a;
}
```

Now, FDCT module here we need not go into the detail, you can try that in any mat lab and other tools. But just to put it in the context this is the expression that we had shown for the cosine transform. Now, that for this cos part, we are storing a cos table where the different values are stored and we will take the d x, y and corresponding to the given x and y I will select the element from this cos table and we will multiply with it all. So this cos table is pre stored and for this basis function one by square root 2 l so on one that is kept here. So, what happens here in this forward discrete cosine transform? We are taking eight rows one by one and for each element of the row x 0 x, x 1, x 2, x 3, we are multiplying that with the cos value of the table, which is pre stored.

We are putting over here we are normalizing it with 32 k, the floating point values are multiplied by this 32 k and you see here whatever we are getting from the cos table, we are dividing that with this and that is chosen. So, that we get in 2 bytes of memory 32 k pay for each of the multiplicands are coming in 2 bytes of memory. So, this in detail you can see in the textbooks or in other material, but here for our purpose, this is the code for the compression.

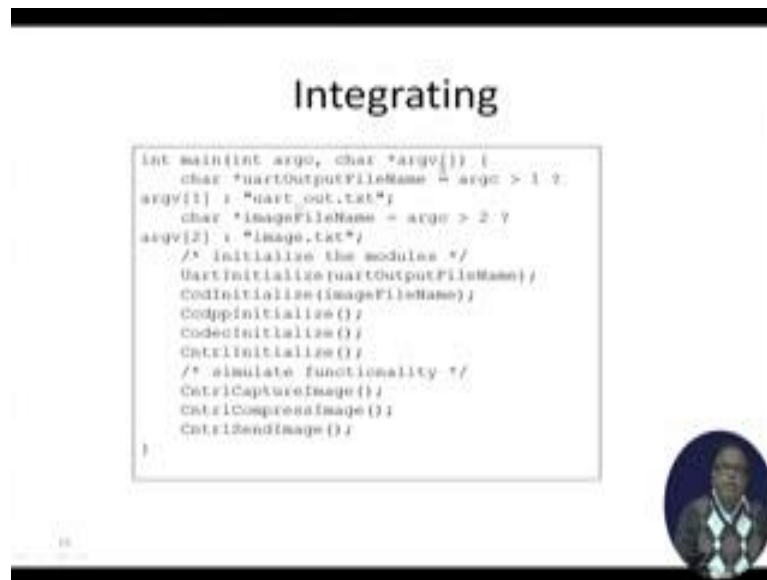
(Refer Slide Time: 13:01)



Once the compression is done, then we come to the controller module because the controller module is what is more important. And look here the controller module is the heart of the system. It has got control initialized here where the number of rows and number of columns are specified. Next, we control capture image, first of all we have to capture image. So, we will do the CCDPP capture which we have shown earlier after preprocessing and then will invoke CCD pop pixel get this, so that is being called first. Then here you will see control compress image the controller is invoking the compression.

So, basically what I want to say here is this that this flow is being maintained by, so as the controller is taking one row with that bias and then taking it row by row, it should not row by row, 8 by 8 and doing the compression, so that is being done by the controller. And in the process it is calling code codec push pixels, it is calling DFCT and then it is also doing the quantization. The quantization is it is taking the particular table, and then dividing it by a particular number 2 to the power something.

(Refer Slide Time: 14:52)



So, integrating everything what we get is that quickly let me also mention about this UART module these very simple, void UART send these actually a half UART because it is only transmitting not receiving. So, void UART send a character, f print f the character on the file that simple UART. So, we have to do, what will be our task that diagram that we had shown is initializing the UART, initializing the CCD except all the initialization. And then I am simulating the functionality, for that I am capturing called the control capturing image that is controlled capture image is this one which has being called which is invoking cddpp capture, which is invoking the CCD capture. And then control compress image; that means, it is calling this one which in turn is calling it is a follower discrete cosine transform etcetera. Then control send image. The send image is somewhere here where is sending UART the upper byte and the lower byte, so that is the overall flow.

Now if I now simulate this then I can very well understand that whether my functional units are all right whether the data values are coming all right, and how much time it is taking or not exactly the actual time, but it will tell me the relative time where it is spending more time where, which are the hot spots in terms of more number of computation and all those things. So, now, will come to designing the system we have seen the task and we have done the scheming; now we try to do the design.

(Refer Slide Time: 16:55)

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

The design will have to first determine the systems architecture it can be processor any comb I mean any combination of single purpose processors or general purpose processors will map the functionality to the architecture will be the implementation. So at the starting point - last bullet, at the starting point, let us start with a low and general purpose processor. First, we try to since, you have to minimize cost, we first try to do everything on a general-purpose person then we the first test turnaround time because I will have of the shells microprocessor and a flash memory connected to that.

(Refer Slide Time: 17:37)

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
 - CcdappCapture 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

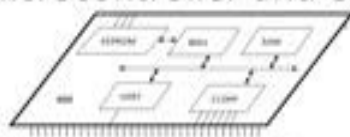
So, if I take a low-end Intel 8051 microcontroller, the total IC cost will be around 5 dollars nothing practically. It is well below 200 milliwatt power. Time-to-market is small. However, where am I getting stuff, 12 megahertz, and if I need 12 cycles per instruction, then I will execute one million instructions per second, 12 mega hertz and 12 cycles per instruction then I will need one million instructions per second.

So, CCDPP capture where I am doing the zero bias that loop nested loop results in 4000 iterations if I look here, here CCDPP capture here, it will be 64 time 64 iterations. Therefore, I will be needing 4 k iterations. And if I assume that (Refer Time: 18:57) iteration there are 100 assembly instructions; that means, I will have 400,000 instructions per image then half of 400,000 instructions per image, and I will need one million instructions per second. So, half a second is gone I am assuming that 500,000. So, half a second is gone only for capturing and preprocessing the image.


And after that doing the computation intensive DCT and all those will certainly I will not be able to accommodate to my time constant of one second. I had the time constant of one second. Therefore, after doing this analysis, fine said well signal micro process solution will not do. So, what can I do the, second one is botheration here was the CCDPP the pre processing part that was bothering that it up 50 percent of the budget.

(Refer Slide Time: 19:57)

**Implementation 2:
Microcontroller and CCDPP**



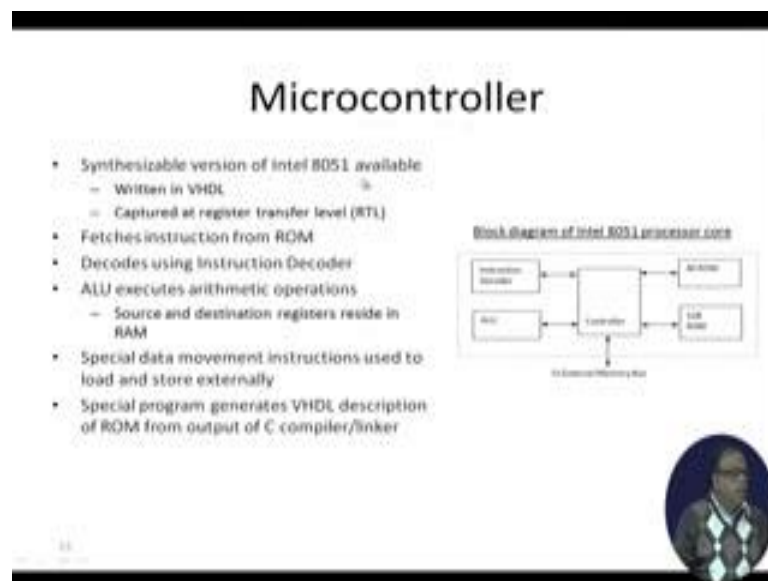
- CCDPP function implemented on custom single-purpose processor
 - = Improves performance – less microcontroller cycles
 - = Increases NRE cost and time-to-market
 - = Easy to implement
 - Simple datapath
 - Few states in controller
- Simple UART easy to implement as single-purpose processor also
- EEPROM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



So, now let me try to do this on a single-purpose processor an ASAP, ASAP or ASSIC whatever I do, I do it on a separate (Refer Time: 20:10). So, my architecture will look

like I have got an UART I have got some 8051, and there will be another CCDPP, CCDPP. Now, it will improve performance, it will require less microcontroller cycles. However, it will increase the energy costs, but it is simple to implement the data path as few states it is not because what are we doing there, what are you doing in the preprocessing, we are just doing subtract some averaging plus I mean we are doing the average and then subtract. So, we can reuse some other subtractors and have a simple data path. And simple UART is also easily implementable.

(Refer Slide Time: 20:57)




So, now microcontroller here, we can have a microcontroller code, code microcontroller thing. And from there all of you remember what is the code, we have got some course already prepared and we have to synthesize on that, so that is already synthesized I am sorry that is already synthesized we have to put it on the mask and all those. So, a block diagram of an 8051 code will be some instruction decoder ALU some 4 k RAM, 128 RAM, 4 k ROM, and there will be controller. Now, this is not very difficult to do and we will have special program that will generate the VHDL description of the whole thing this entire thing is written in VHDL when it when a particular design is offered to you as a core then the corresponding VHDL is given. So, you can synthesize from that VHDL not a big deal.

(Refer Slide Time: 22:01)

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




Similarly, the UART can be easily computed. It will be invoked, this is the automator, start then it will transmit the data till 8 bits sent that will go to stop. So, this is the cycle. So, lower 8 bits will come to the RAM lower address, and upper bit bits will come to the memory mapped I O devices. Now, so there is also simple designing an UART would be a problem.

(Refer Slide Time: 22:34)

CCDPP

- Hardware implementation of zero-bias operations
- Interacts with external CCD chip
 - CCD chip needs external for our SOC mostly because containing 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 B[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

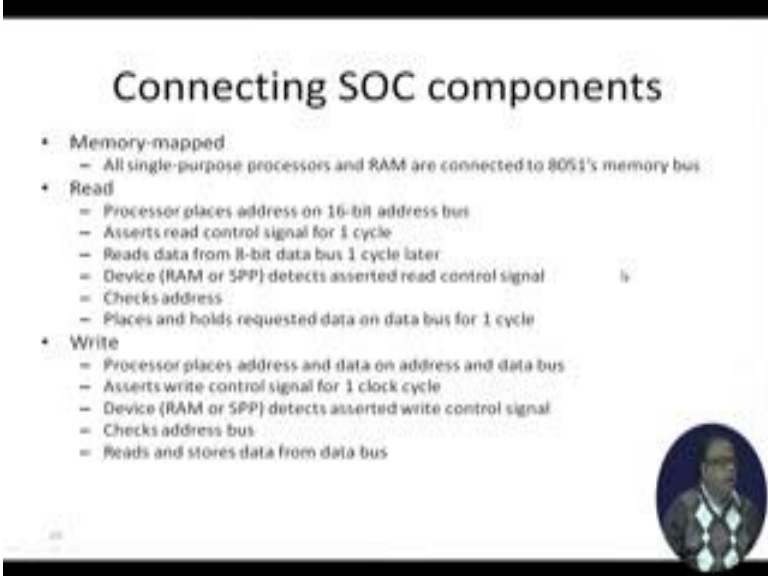


Now, the CCDPP, CCDPP the preprocessing will have the FSMD, let us see here we are giving. So, what we did earlier, whenever we did some synthesis, we started with an

FSMD and did the synthesis that we did for the GCD algorithm that we also did for the square root approximation algorithm. So, we now know how to do that, we know how to schedule it and all those. So, we can handle this from this FSMD.

Similarly, the CCDPP, what is it doing it is a idle row 0 column 0; its getting low first of all, it is not very clear. The buffer row and column is getting the pixel and it is going why is its less than 66, I was having the image 64 by 64, but there were two extra column therefore, I have to scan up to 66 and take the 65th and 66th columns and take their average right. So, that is I am getting the row computing the bias here. And then fixing the bias subtracting for all the elements in a loop, you see here there is a loop for all the 64 all the 64 columns, all the 64 columns I correct the error zero bias errors. And then I go to the next row all right in this cycle, I go on. I hope it is clear to all of you is the zero bias correction, so that will be my FSMD of the CCDPP.

(Refer Slide Time: 24:25)

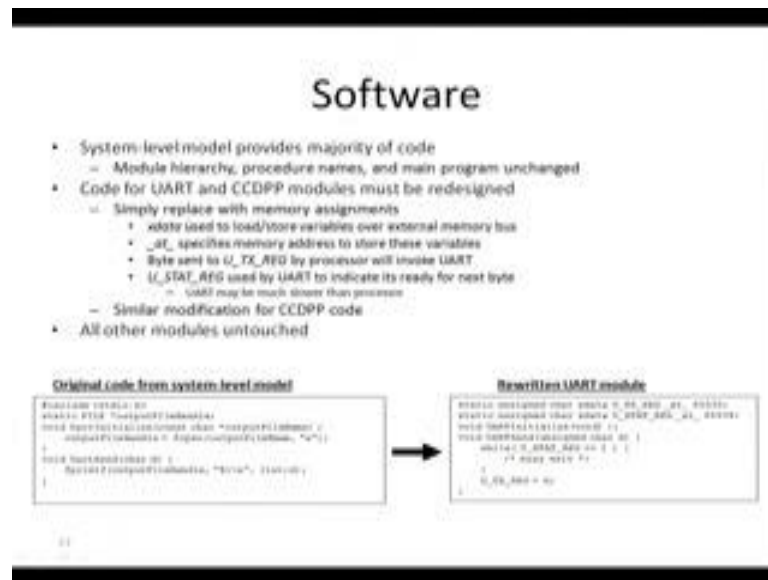


Connecting SOC components

- 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

And then we can connect all of these to make a system on chip. So, we had this I want to have this as a system on chip with all the things on one chip, so that I can put that in the camera. So, the I O is memory mapped, all single purpose processor and ram are connected to the 8051s memory bus, processor places address on the 16 bit address bus as our state control and all those things are done that is my second implementation. So, in my second implementation, what do I have I have the CCDPP as a separate processors, dedicated processors or an ASIC, and there is a 8051 and there is another UART.

(Refer Slide Time: 25:19)

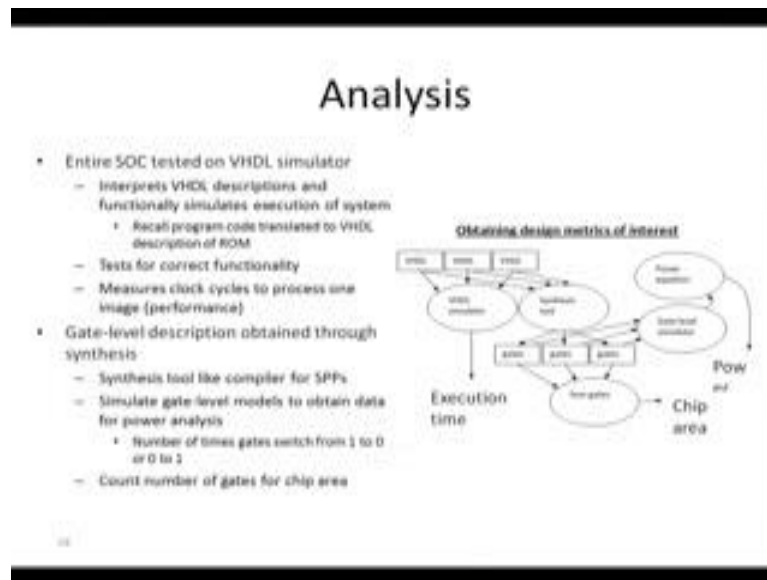


And as I do this, as the communication is changing if I look at this here again, now the UART original is the UART was the connected to the 8051. Now, the UART is (Refer Time: 25:39) has to communicate with the bus. Therefore, my change in my hardware decision will also affect the software. So, I will change this here little bit of change, where earlier I was just writing as if in a file here I am not doing that here I am going through a status register and as long as there is busy I am because I am looking at the bus I am waiting whenever that is free, I am writing the data. So, this change has to do because the communication. Each decision why do you call it hardware software code design, the reason is every time I am making a decision that decision is affecting may affect the other part also, so that is always happening as is being shown here.

Student: (Refer Time: 26:36).

That will increase your cost. The question is that why do we do this we have got we are fixing some design decisions that this transfer will be done serially that we have decided. If we had done it, if we had no constraint we could have said that we will do it parallelly, but that will increase the cost. So, we are not because that takes very little time even serially and that is not when you are taking the picture you are transferring into the pc and you are not so much hard paste with that time that is why.

(Refer Slide Time: 27:13)





Now, I have done the second implementation. How good is my second implementation, the first one I did so some analysis. So, a typical way of analysis of standard way of analysis will be that I have got a number of VHDL codes. So, all these VHDL codes can be simulated, this could be very low codes as well. I simulate and I find the execution time how much time it takes, but that is not the only thing, time is 1, then I am taking them and synthesizing them, so a synthesis tool. And looking at how many gates I am getting and from there I am estimating the chip area. Remember our nonfunctional constraint said we should restrict ourselves within 200,000 gates, that being this respected also we have to look at the power. So, this is how we will be doing the analysis of the different designs.

(Refer Slide Time: 28:25)

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





Now, as I do the analysis of our second implementation we find the total execution time for processing one image is 9.1 seconds not done, our for one image we needed 1 seconds here it is coming to 9.1 second even after dedicating a separate processing module for preprocessing. The power consumption is 0.33 watt not that bad. Energy consumption is 0.3 joule. Total chip area is 98,000 gates; I am still in the range of 200,000 gates. So, can I do something can I add some more hardware to make it make the process fast I can see that.

(Refer Slide Time: 29:23)

Implementation 3: Microcontroller and CCDPP/Fixed-Point DCT

- 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



So, now I look at implementation three. The implementation three is looking at the microcontroller and DCT. So, it should be initially, I will do floating point then I will come back to fixed point. So, 9.1 second is not meeting my performance. DCT operation is the prime candidate for improvement. Now, here at this level I can also see which one is consuming how much time, I can find out the hot spot that is called profiling and I can find out which are the culprits and here I can find that the DCT is a prime candidate. So, execution of implementation two shows that microprocessor spends most cycle in DCT computations. Could design custom hardware for the DCT, but obviously, that is a more complex and more design effort instead, so, one thing is that I can do this where is it the DCT path which was the (Refer Time: 30:50).


(Refer Slide Time: 30:50)

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

```
graph LR; ID[Instruction Decoder] --> C[Controller]; ALU[ALU] --> C; C --> AR[Accumulator]; C --> SR[Stack Pointer]; C --> RAM[RAM]; C --> ROM[ROM]; C --> MB[External Memory Bus];
```



(Refer Slide Time: 30:58)

Refined functional specification

- One that can actually be executed
- Can use C/C++ code to describe each function
- Can provide insight into operations of system
 - = Profiling can find computationally intensive functions
- Can obtain sample output used to verify correctness of final implementation

Executable model of digital camera


```
graph LR; ImageFile[Image file] --> CCD_C([CCD.C]); CCD_C --> CCDFPC([CCDFPC]); CCDFPC --> CNTRL_C([CNTRL.C]); CNTRL_C --> UART_C([UART.C]); UART_C --> OutputFile[Output file]; CCDFPC <--> CODEC_C([CODEC.C]);
```

Now, let us go. So, our flow the next flow was the DCT. Now, the DCT 1, I was trying to find this one yes. Here, this path would be designed as hardware just as we have done this one as hardware, but that will be more complicated. Since that is becoming more complicated, what we can try to do I will keep it in software, but I will work on the software specification and modify the specification to see if something can be done.

(Refer Slide Time: 31:32)

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



So, let us try to do that. What we do the floating point DCT that was there, we found that DCT was using around 260 floating point operations per pixel. And you know floating

(Refer Slide Time: 32:32)

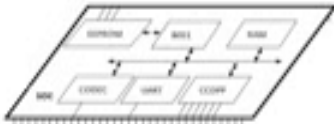
So, next we come to the fixed point implementation of the coding. So, as we go for the fixed point representation of the cosine values, this table changes, this table simply changes because now all these values we will be doing in fixed points, fixed point arithmetic. 6 bits used for the fractional portion. So, what is done you know each fractional portion is converted to an integer when we do it a fixed point. So, result of multiplication will be shifted right. So, the other things remain the same, other things remain the same I simply change the cos table. So the multiplication will not now be done in fixed point.

Now, as I do that and analyze it, so I did not touch the hardware I just change the computation and we find that we can do oh still 1.5 seconds, power consumption is same energy consumption what was the earlier one energy point; so energy let me see the energy consumption was 0.3 joule and here it became less. So, battery life is six times longer by doing the you see. So, whenever in a mobile phone or in a laptop, it runs out

the battery many things happens. How many fixed point operations you are doing how many floating point operations you are doing that also affects the battery life. So, here just by making it fixed point we could save on the battery life our chip area is even saved, but our main constraint has not been met still it is 1.5 seconds.

(Refer Slide Time: 34:37)

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

So, what can we do, the performance is closed. Now, we have got no option, but to implement the codec in the hardware. In the fourth option, we have to design the codec in the hardware. So, we want to do a single purpose, but I am not doing by 64 by 64, I am just doing a codec for 8 by 8, and I will invoke it 64 times.

(Refer Slide Time: 35:07)

CODEC design

- 4 memory mapped registers
 - C_DATA_REG/C_DATA_REG used to push/pop 8 x 8 block into and out of CODEC
 - C_CMD_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 in implementation 2

Rewritten CODEC software

```
void codec_init(void) {
    C_DATA_REG = 0;
    C_CMD_REG = 0;
    C_STAT_REG = 0;
}

void codec_write_block(void) {
    C_DATA_REG = 0;
    C_CMD_REG = 1;
    while (C_STAT_REG == 0) {}
    C_DATA_REG = 0;
    C_CMD_REG = 0;
}

void codec_read_block(void) {
    C_DATA_REG = 0;
    C_CMD_REG = 1;
    while (C_STAT_REG == 0) {}
    C_DATA_REG = 0;
    C_CMD_REG = 0;
}
```

So, as I do that that codec design will again the codec software will be rewritten in and then from the C code of the codec, I will take it to the VHDL in the fixed point version and then I will synthesize it.

(Refer Slide Time: 35:24)

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

As I do it then when I carry out the analysis I find that the total execution time is now 0.09, well under one second, well under one second, it has become too fast right. So, might be if I a time, I would have done smaller first I do 4 by 4, and call it a number of times all those things could have been done, but however. So, when I do this, so what do

I do for this codec design? So, I took this registers and converted into VHDL and synthesize it. So, we find that the time has been met power consumption is slightly increased because of the hardware. Energy consumption is less.

Can you tell me, why the power consumption is slightly increase, energy consumption is less? The reason is energy is integral over time now the time is becoming faster, and less amount is being done in the software, therefore my total energy consumption is less, battery life has become 12 times. Total chip area increase of course, because this codec has been done hardware 128000 gates that is a significant increase over previous implementation, but my constraints are satisfied.

(Refer Slide Time: 37:05)

Summary of implementations

	Implementation 1	Implementation 2	Implementation 3
Performance (cycles)	1.5	1.5	0.99
Power (mW)	0.33	0.33	0.40
Gate count	50,000	50,000	128,000
Energy (mJ)	0.50	0.50	0.39

- 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?

So, summary of the implementation; implementation 1 was not viable. Implementation two are you see 9.1 was the time from the 1.5 then 0.99 my constant was one I have made that. Now, this one is called the satisfying constraint or the constraint that I have to satisfy I need not optimized on this. But these ones have to minimize as much as possible, but here again no minimization was given. It was told that it should be low enough. So, 0.33, 033 and 040 not a big deal, the gate size is the constraint was given 200,000 it is well within that limit. Energy as reduced, so that is a great product. So, we got this great product by iterative design iterative design and this example has shown you that how we analyze our specification and look at the different components and see how we can optimize we identify the hot spot.

So, summarizing what do you say, we started to spec, we first of all we have to do the proper task analysis, understand the task, otherwise you will not be able to fix where I need to tweak the design, you will not be able to understand where I will have to tweak the design. So, you have to understand this what is happening because suppose you do you say that I will reduce the number of pixels that will affect the quality of the image. So, what you can do and what you can do must be understood and for that you have to understand the application process. Any embedded system designer must spend some time to understand the application. Then we looked at the performance and we have to do the analysis and it is an iterative cycle by which you see that where the shoe pinches and there we have to see, first of all whether we can do it in software, if possible less effort, otherwise you let it to hardware.

So, that was a nice example which summarized a number of things that we have done till now. So, in the next class, we will look at some formal approach to hardware, software partitioning, and we will move to optimization.

Thank you.