

- 6.14 Select one of the following parallel bus protocols, then, perform an Internet search for information on transfer rate, addressing, DMA and interrupt control (if applicable), and plug-and-play capability (if applicable). Then give timing diagrams for a typical transfer of data (e.g., a write operation). The protocols are STD 32, VME, SCSI, ATAPI, Micro Channel, or any other parallel bus in use by the industry and not described in this book.

## CHAPTER 7: *Digital Camera Example*



- 7.1 Introduction
- 7.2 Introduction to a Simple Digital Camera
- 7.3 Requirements Specification
- 7.4 Design
- 7.5 Summary
- 7.6 References and Further Reading
- 7.7 Exercises

### 7.1 Introduction

In the previous chapters, we introduced general-purpose processors, custom single-purpose processors, standard single-purpose processors, memory, and techniques for interfacing processors and memory. In this chapter, we apply this knowledge to design a simple digital camera. In particular, we will examine the trade-offs of using general-purpose versus single-purpose processors to implement the necessary camera functionality. We will see that choosing a good partitioning of functionality among the different processor types is essential to building a good design. This in turn requires a unified view of different processor types, as this book has thus far stressed.

We begin with a general introduction to digital cameras and their inner workings. We then develop the camera's specifications, which describe the desired behavior as well as constraints on design metrics like performance, size, and power. We explore several alternative implementations of the digital camera and compare their design metrics.

### 7.2 Introduction to a Simple Digital Camera

A digital camera is a popular consumer electronic device that can capture images, or "take pictures," and store them in a digital format. A digital camera does not contain film, but rather one or more ICs possessing processors and memories. Digital cameras were not possible over a decade ago, because small-enough ICs could not process fast enough or store enough bits to

be feasible. The advent of systems-on-a-chip and high-capacity flash memory has made such cameras possible.

### User's Perspective

From a user's point of view, a simple digital camera works as follows. The user turns on the digital camera, points the camera lens to the scene to be photographed, and clicks the "shutter" button. The user can repeat these steps until up to  $N$  images are stored internally in the camera. Here,  $N$  is a constant that depends on the model of the camera, which in turn depends on the amount of memory in the camera and the number of bits used per image. The user may also attach the digital camera to a PC, say, by using a serial cable, to download the photos to a hard disk for permanent storage.

### Designer's Perspective

From a designer's point of view, a simple digital camera performs two key tasks. The first task is that of processing images and storing them in internal memory. The second task is that of uploading the images serially to an attached PC.

The task of processing and storing images is initiated when the user presses the shutter button. At this point, the image is captured and converted to digital form by a *charge-coupled device* (CCD). Then, the image is processed and stored in internal memory. The task of uploading the image is initiated when the user attaches the digital camera to a PC and uses special software to command the digital camera to transmit the archived images serially. Let us look at these actions in more detail.

A CCD is a special sensor that captures an image. A CCD is a light-sensitive silicon solid-state device composed of many small cells. The light falling on a cell is converted into a small amount of electric charge, which is then measured by the CCD electronics and stored as a number. The number usually ranges from 0, meaning no light, to 256 or 65,535, meaning very intense light per pixel. Figure 7.1 illustrates the internals of a CCD. On the periphery, a CCD is composed of a mechanical shutter. This is a screen that normally blocks the light from falling on the light sensitive surface. When activated, the screen opens momentarily and allows light to hit the light sensitive surface, charging the cells with electrical energy that is proportional to the amount of light passed in. The screen typically sits behind an optical lens that focuses the scene observed through the viewfinder onto the light sensitive surface of the CCD. A CCD also has internal circuitry that measures the electric charge of each cell, converts it to a digital value, and provides an interface for outputting the data.

Due to manufacturing errors, the light-sensitive cells of a CCD may always measure the light intensity to be slightly above or below the actual value. This error, called the zero-bias error, is typically the same across columns but different across rows. For this reason, some of the left most columns of a CCD's light-sensitive cells are blocked by a strip of black paint. The actual intensity registered by these blocked cells should be zero. Therefore, a reading of other than zero would indicate the zero-bias error for that row. Figure 7.1 shows the covered cells. This becomes clearer as we give an example in the next paragraphs.

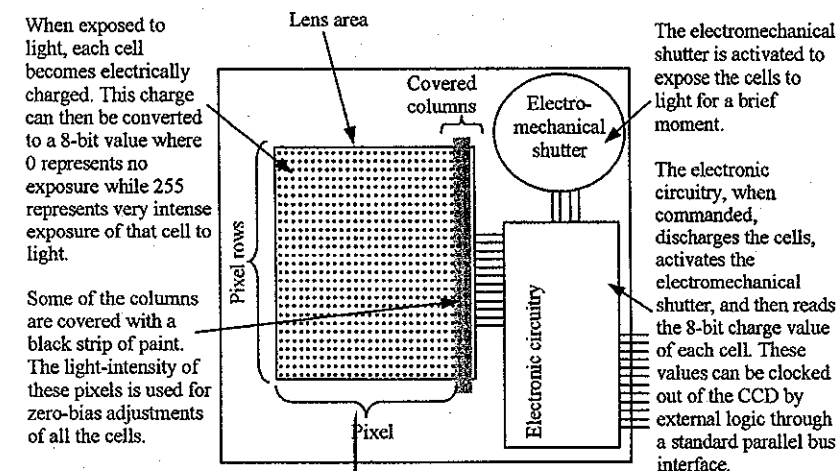


Figure 7.1: Internals of a charge-coupled device (CCD).

A digital camera uses a CCD to capture an image. Once the image is captured, it must be corrected to eliminate the zero bias error. Then, the image must be encoded using the JPEG encoding scheme. The task of bias adjustment is described next.

Figure 7.2 shows a raw image block of size  $8 \times 8$  pixels that is captured using a CCD of that size. Normally, the CCD would be of much greater resolution, say  $640 \times 480$  pixels, but we use a small one to be able to illustrate the various operations of a digital camera in this chapter. Notice in Figure 7.2(a) that there are 10 columns. As mentioned earlier, the last two columns are extra and are used to detect zero-bias. Recall that these two columns are covered and should normally read a value of zero. Looking at the last two columns of the first row, we see that the measured light intensity is on the average 13 units larger than the actual light intensity. We obtain 13 by averaging the last two columns  $((12 + 14) / 2) = 13$ . We can thus correct the error for this row by subtracting 13 from each element of the first row. We can repeat this process for each row to obtain a block of  $8 \times 8$  pixels that has been corrected for zero bias errors. The corrected block is given in Figure 7.2(b).

The next step is to compress the image, which reduces the number of bits needed to store the image in memory. Compression allows us to store more images in limited amount of memory. Compressed images can also be transmitted to a PC in less time. We'll perform JPEG encoding of the image. JPEG is a popular standard format for representing digital images in a compressed form. JPEG, pronounced "jay-peg," is short for Joint Photographic Experts Group. The word *joint* refers to the group's status as a committee working on both ISO and ITU-T standards. Their best-known standard is for still-image compression.

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

(a)

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		

(b)

Figure 7.2: A block of  $8 \times 8$  pixels as captured using a CCD: (a) before zero-bias adjustment; the last 2 columns help represent zero bias for a given row, and (b) after zero-bias adjustment.

JPEG encoding provides for a number of different modes of operation. For a full coverage of the JPEG encoding, the reader is referred to the reference section at the end of this chapter. The mode that we discuss in this chapter is an encoding that provides for high compression ratios using the discrete cosine transform (DCT). To compress an image, the image data is divided into blocks of  $8 \times 8$  pixels each. Each block is then processed in three steps. The first step performs the DCT, the second step performs quantization, and the last step performs Huffman encoding.

The DCT step transforms our original  $8 \times 8$  pixel block into a cosine-frequency domain. Once in this form, the upper-left corner values of the transformed data represent more of the essence of the image while the lower-right corner values represent finer details. We can, therefore, reduce the precision of these lower-right corner values to facilitate compression while retaining reasonable overall image quality. The actual DCT operation is given in this formula:

$$C(h) = \text{if } (h = 0) \text{ then } 1/\sqrt{2} \text{ else } 1.0$$

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

Here,  $C(h)$  is simply an auxiliary function used in the main equation, namely,  $F(u,v)$ . The function  $F(u,v)$  gives the encoded pixel at row  $u$ , column  $v$ .  $D_{xy}$  is the original pixel value at row  $x$ , column  $y$ . Of course, it would be useless to have a DCT transform if we are unable to reverse the process and obtain the original. Below is the inverse DCT (IDCT), although it is not necessary in the implementation of our simple digital camera:

$$C(h) = \text{if } (h = 0) \text{ then } 1/\sqrt{2} \text{ else } 1.0$$

$$f(x,y) = 1/4 \sum_{u=0..7} \sum_{v=0..7} C(u) \times C(v) \times E_{uv} \times \cos(\pi(2u+1)x/16) \times \cos(\pi(2v+1)y/16)$$

Again,  $C(h)$  is simply an auxiliary function used in the main equation, namely,  $f(x,y)$ . The function  $f(x,y)$  gives the original pixel at row  $x$ , column  $y$ .  $E_{uv}$  is the DCT-encoded pixel value,

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

(a)

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

(b)

Figure 7.3: A block of  $8 \times 8$  pixel as captured using a CCD, zero bias corrected: (a) after being encoded using DCT, (b) then after quantization.

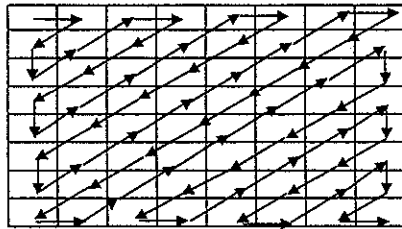
using the previous equation, for row  $u$  and column  $v$ . Figure 7.3(a) shows the DCT-encoded values for our sample block of  $8 \times 8$  pixels. The inverse process will obtain the block in Figure 7.2(b) from that in Figure 7.3(a).

The DCT is sometimes distinguished from the IDCT by referring to the DCT as the forward DCT, or FDCT.

The next processing step is to reduce the quality, of the encoded DCT image, which helps us compress the image. We do this by reducing the bit precision of the encoded data. Note that if we represent the pixels with less precision, we will need fewer bits to encode them, thus achieving compression. For example, we can divide all the values by some factor of 2 (since division by a factor of 2 is achieved simply by right shifts), such as 8. This is the step where we actually loose image quality in order to achieve high compression ratios. This process is referred to as *quantization*. To decompress, we would perform a dequantization. In other words, we would multiply each pixel by the same factor of 2 (i.e., 8 in our example). Figure 7.3(b) illustrates the quantization applied to the block of  $8 \times 8$  shown in Figure 7.3(a).

The last step of the JPEG compression is the encoding of data. Here, the block of  $8 \times 8$  pixels is first serialized. Specifically, the values are converted into a single list according to a zigzag pattern, as shown in Figure 7.4. Then, the values are Huffman encoded. *Huffman encoding* is a minimal variable-length encoding based on the frequency of each pixel. In other words, the frequently occurring pixels will be assigned a short binary code while those that don't occur as frequently will be assigned a longer code. Let us explain that with an example. In Figure 7.5(a), we have given the frequency of pixel occurrence of the encoded and quantized  $8 \times 8$  block shown in Figure 7.3(b). Here, as shown, the encoded pixel value -1 occurs fifteen times while the encoded pixel value 14 occurs only one time.

From this information, we construct a Huffman tree as illustrated in Figure 7.5(b). With each node in such a tree, we associate a value that is computed as follows. For an internal node, the value is the sum of the values of the children of that node. For a leaf node, the value is the frequency of occurrence of the pixel being represented by that leaf node. The tree is constructed from the bottom up (i.e., starting from leafs and working up toward the root).

Figure 7.4: Data encoding sequence of a block of  $8 \times 8$  pixel.

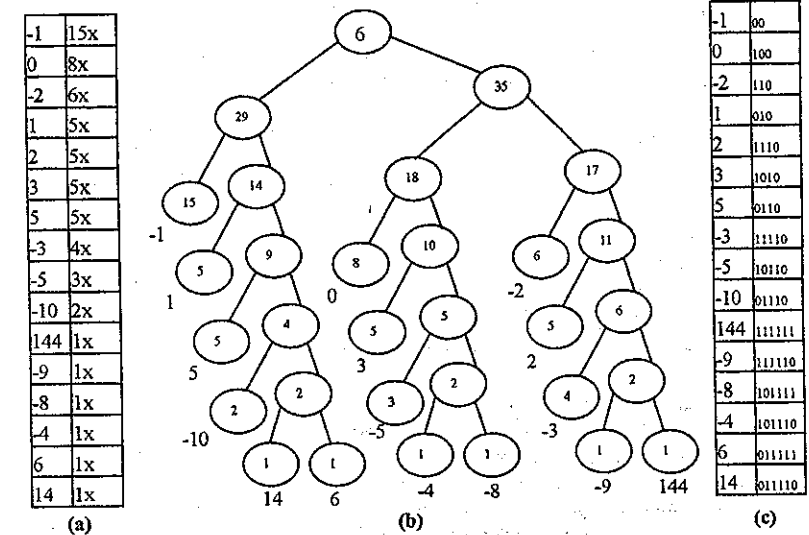
Initially, we create a leaf node for each of the pixels and initialize the values of these nodes according to the pixel's frequency. Then we create an internal node by joining any two nodes that will result in the minimum value. We repeat this process until we have a complete binary tree.

Once the Huffman tree is constructed, we can obtain a binary code for each of the pixel values by traversing the tree starting at the root down to the leaf labeled with that pixel. While traversing the tree, we construct a binary string. Each time we traverse down past a right child we append a "1" to our binary string, whereas each time we traverse down a left child we append a "0" to our binary string. For example, in order to obtain the binary code for the pixel value -3 in Figure 7.5(b), we would make four right traversals and a left traversal, thus obtaining the binary string "11110". Figure 7.5(c) gives the Huffman codes for the remaining pixel values.

Given these Huffman codes, we encode our block of  $8 \times 8$  pixels by creating a long string of 0s and 1s. Here we take the sequence of pixels generated by the zigzag ordering shown in Figure 7.4, and for each pixel we output the Huffman binary code. In our example of Figure 7.4, we would obtain the binary string "111111011001110...."

As stated earlier, Huffman encoding achieves compression by assigning a short binary code to the most frequently appearing pixel values, while leaving longer binary codes for the least frequently appearing pixels. Of course, this process is reversible since Huffman encoding also ensures that no two codes are a prefix of each other.

Our next processing step is to archive our image. This step is rather easy. We simply record the starting address and size of each image. We can use a linked list data structure to record this information. If we know beforehand that the camera will hold at most  $N$  images, we can set aside a portion of memory for our  $N$  addresses and  $N$  image-size variables. In addition, we would need to keep a counter that tells us the location of the next available address in memory. For example, initially, all  $N$  addresses and image-size variables might be set to 0. Our global memory address will be set to  $N \times 4$ , assuming that the address and image-size variables occupy the initial  $N \times 4$  bytes in memory. Then, the first image will be archived in memory starting at location  $N \times 4$ . Assuming the image was of size 1024, then we will update our global memory address to  $N \times 4 + 1024$ , and so on. Of course, there are other

Figure 7.5: Huffman encoding of the block of  $8 \times 8$  pixels shown in Figure 7.3(b): (a) the pixel values and associated frequencies, (b) the resulting Huffman tree, (c) and the Huffman codes.

ways to perform such archiving. In any event, our memory requirement will be based on  $N$ , the image size and the average compression ratio that we can obtain using JPEG encoding.

Finally, the only processing task that remains is to upload the images and free the space in memory when a PC is connected to the camera and an upload command is received. To accomplish this, we use a UART. As you'll recall, a UART transmits data serially over a single data wire. Our processing task will be to read the images from memory and transmit them using the UART. As we transmit images, we reset the pointers, image-size variables and the global memory pointer accordingly.

It must be noted again that our description of a digital camera is very simple. A real digital camera will enable you to take pictures of varied sizes, display images on an LCD, allow image deletion, perform advanced image processing such as digitally stretching, zooming in and out, and many other things.

## 7.3 Requirements Specification

Our digital camera product's life begins with a requirements specification. A specification describes what a particular system should do, namely the system's requirements. Specifications include both functional and nonfunctional requirements. Functional

requirements describe the system's behavior, meaning the system's outputs as a function of inputs (e.g., "output X should equal input Y times 2"). Nonfunctional requirements describe constraints on design metrics (e.g., "the system should use 0.001 watt or less"). The initial specification of a system may be very general and may come from our company's marketing department. The initial specification for our camera might be a short document detailing the market need for "a very basic low-end digital camera capable of capturing and storing at least 50 low-resolution images and uploading such images to a PC, costing around \$100, with a single medium-sized IC costing less than \$25, including amortized NRE costs. Battery life should be as long as possible. Expected sales volume is 200,000 if market entry is earlier than 6 months, and 100,000 if market entry is between 6 to 12 months. Beyond 12 months, this product will not sell in significant quantities."

Let us begin by discussing the nonfunctional requirements in more detail, followed by an informal high-level functional specification, and then a more detailed description of behavior.

### Nonfunctional Requirements

Given our initial requirements specification, we might want to pay attention to several design metrics in particular: performance, size, power, and energy. Performance is the time required to process an image. Size is the number of elementary logic gates (such as a two input NAND gate) in our IC. Power is a measure of the average electrical energy consumed by the IC while processing an image. Energy is power times time, which directly relates to battery lifetime. Some of these metrics will be constrained metrics — those metrics must have values below (or in some cases above) a certain threshold. Some metrics may be optimization metrics — those metrics should be improved as much as possible, since this optimization improves the product. A metric can be both a constrained and optimization metric.

Regarding performance, our design must process images fast enough to be useful. We might determine that a reasonable timing constraint is 1 second per image. Note that the terms timing and performance are often used interchangeably. More time than 1 second would probably be quite annoying from a camera user's perspective. Imagine having to wait 10 seconds after pressing the shutter button before you could press the button again. A typical soccer parent would probably not buy such a camera, for fear of missing a great goal! On the other hand, since we are aiming for the low-end of the digital camera market, our performance doesn't need to be much better than 1 second. Thus, performance is a constrained metric but not an optimization metric — anything less than 1 second is equally good.

Regarding size, our design must use an IC that fits in a reasonably sized camera. Suppose that, based on current technology, we determine that our IC has a size constraint of 200,000 gates. In addition to being a constrained metric, size is also an optimization metric, since smaller ICs are generally cheaper. They are cheaper because we can either get higher yield from a current technology or use an older and hence cheaper technology.

Finally, power is a constrained metric because the IC must operate below a certain temperature. Note that our digital camera cannot use a fan to cool the IC, so low power operation is crucial. Let's assume we determine the power constraint to be 200 milliwatt. Energy will be an optimization metric because we want the battery to last as long as possible. Notice that reducing power or time each reduces energy.

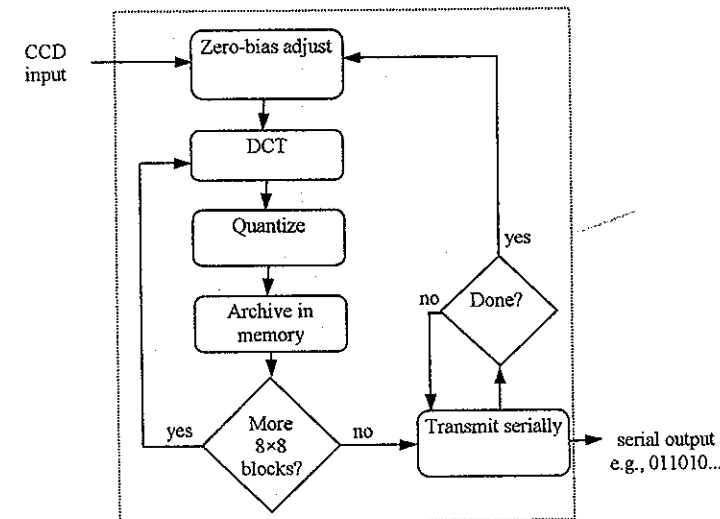


Figure 7.6: Functional block-diagram specification of a digital camera.

### Informal Functional Specification

We can describe the high-level functionality of the digital camera by using the flowchart in Figure 7.6. We see the major functions involved in image capture, namely zero-bias adjust, DCT, quantize and archive in memory. We also see the function transmit serially. We could then describe each function's details in English; we omit such descriptions here since they were included earlier in the chapter. We'll assume a very low-quality image with a  $64 \times 64$  resolution, meaning the CCD has 64 rows and 64 columns.

Note that Figure 7.6 does not dictate that each of the blocks be mapped onto a distinct processor. Instead, the description only aids in capturing the functionality of the digital camera by breaking that functionality down into simpler functions. The functions could be implemented on any combination of single-purpose and general-purpose processors.

### Refined Functional Specification

We can now concentrate on refining the informal functional specification into one that can actually be executed. This typically consists of a C or C++ program describing the functionality. In our case, we could write C or C++ code to describe each function in Figure 7.6. Such a software prototype of the system is often referred to as a system-level model, a

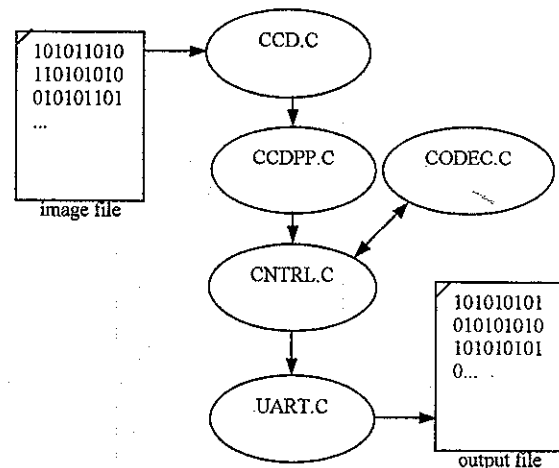


Figure 7.7: Block-diagram of the executable model of the digital camera.

prototype, or simply a model, though the prototype is also a first implementation. Keep in mind that one person's specification may be another person's implementation.

The software prototype can be executed on our development computer to verify its correctness. It can also provide insight into the operations of our system. For example, in our digital camera, we can profile our executable specification as it is running, in order to find the computationally intensive functions. Recall that a profiling tool is a tool that watches a program under execution and records the number of times a particular procedure or function call was made, or a variable was written or read. We can also use the prototype to obtain sample output that is later used to verify the correctness of our final implementation. For example, we can run an image through our executable specification and obtain the serially encoded output and store that in a file. Later, when we are testing our final IC chip, we can feed it the same image and check that the output matches the expected output.

Figure 7.7 gives the block-diagram of our high-level model of the digital camera. Our executable model is composed of five modules. We start with the CCD module and its corresponding C file called *CCD.C*, as shown in Figure 7.8. This module is responsible for simulating a real CCD (i.e., it is designed to mimic the operations of an actual CCD). It does that by simply reading the pixels of an image directly from a file that we specify. This module exports three procedures, *CcdInitialize*, *CcdCapture*, and *CcdPopPixel*.

```

#include <stdio.h>
#define SZ_ROW    64
#define SZ_COL    (64 + 2)
static FILE *imageFileHandle;
static char buffer[SZ_ROW][SZ_COL];
static unsigned rowIndex, colIndex;
void CcdInitialize(const char *imageName) {
    imageFileHandle = fopen(imageName, "r");
    rowIndex = -1;
    colIndex = -1;
}
void CcdCapture(void) {
    int pixel;
    rewind(imageFileHandle);
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            if( fscanf(imageFileHandle, "%i", &pixel) == 1 ) {
                buffer[rowIndex][colIndex] = (char)pixel;
            }
        }
        rowIndex = 0;
        colIndex = 0;
    }
}
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;
}

```

Figure 7.8: High-level implementation of the CCD module.

The *CcdInitialize* procedure is called to initialize our model just prior to execution. It takes as a parameter the name of the image file that is used to obtain the pixel data. The *CcdCapture* procedure is called to actually capture an image, in this case, read it from a file. The *CcdPopPixel* procedure is called to get the pixels out of the CCD, one at a time. At this point, you should have noted that in our executable specification, our modules communicate using procedure calls and parameter passing.

Our next module is called, rather cryptically, *CCDPP*, and its corresponding C file is called *CCDPP.C*, as shown in Figure 7.9. The PP stands for preprocessing. This module

```

#define SZ_ROW      64
#define SZ_COL      64
static char buffer[SZ_ROW][SZ_COL];
static unsigned rowIndex, colIndex;
void CcdppInitialize() {
    rowIndex = -1;
    colIndex = -1;
}
void CcdppCapture(void) {
    char bias;
    CcdCapture();
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] = CcdPopPixel();
        }
        bias = (CcdPopPixel() + CcdPopPixel()) / 2;
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] -= bias;
        }
    }
    rowIndex = 0;
    colIndex = 0;
}

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

```

Figure 7.9: High-level implementation of the CCDPP module.

performs the zero-bias adjustment processing, shown in Figure 7.9 and described at the beginning of this chapter.

This module also exports three procedures called *CcdppInitialize*, *CcdppCapture*, and *CcdppPopPixel*. The *CcdppInitialize* procedure performs any necessary initializations. The *CcdppCapture* procedure is called to actually capture an image. Note that this procedure calls on the *CcdCapture* and *CcdPopPixel* procedures of the CCD module to obtain an image. As it is obtaining the image pixels, it also performs the zero-bias adjustments. The *CcdppPopPixel* procedure is called to get the pixels out of the CCDPP. Note that the interface to the CCDPP

```

#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);
}

```

Figure 7.10: High-level implementation of the UART module.

module is identical to that of the CCD module. We can think of the CCDPP as a CCD that performs the zero-bias adjustments internally.

Let us now look at the UART module and its corresponding C file called *UART.C*, as shown in Figure 7.10. This is really a model of a half UART (i.e., one that only transmits, but does not receive). As with the other modules, the UART module exports an initialization procedure, called *UartInitialize*. This procedure takes a file name, where the transmitted data is written to. The other procedure, *UartSend*, is called when the digital camera is transmitting a byte. The procedure simply writes the transmitted byte to the output file.

Our next module is called CODEC and its corresponding C file is called *CODEC.C*, as shown in Figure 7.11. This file models the forward DCT encoding that was described earlier in this chapter. The CODEC module exports the procedures *CodecInitialize*, *CodecPushPixel*, *CodecPopPixel*, and *CodecDoFdct*. The *CodecInitialize* procedure resets an index that is used by the push and pop procedures for traversing two buffers, described next. The *CodecPushPixel* is called 64 times to fill an input buffer, called *ibuffer*, which holds the original block of  $8 \times 8$  pixels that is to be encoded. The *CodecPopPixel* is called 64 times to retrieve pixels from the output buffer, called *obuffer*, which holds the encoded block of  $8 \times 8$  pixels. Once a block is placed in the input buffer, *CodecDoFdct* is called to actually perform the transform. Therefore, to encode a block of  $8 \times 8$  pixels, we call *CodecPushPixel* 64 times, and *CodecDoFdct* once followed by 64 calls to *CodecPopPixel*. Let us now discuss the actual implementation of this module. The module simply implements the FDCT equation given earlier and presented here again:

$$C(h) = \text{if } (h == 0) \text{ then } 1/\sqrt{2} \text{ else } 1.0$$

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

The first thing that you may note after studying the code is the large table called *COS\_TABLE*. If you look at the above equation, you'll notice that the argument to the cosine function is always one of 64 possible values, because the only variables in the cosine argument expression are the integers  $x$  and  $u$  (or  $y$  and  $v$ ) and each of these variables can take one of 8 values, from 0 to 7. Thus, for performance purposes, we have decided to precompute the cosine value for all these 64 possibilities and store them in a table. Actually, we have done more than that. Instead of storing the floating-point values, we have converted these to an integer representation.

```

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 short ONE_OVER_SQRT_TWO = 23170, ibuffer[8][8], obuffer[8][8], idx;
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 int FDCT(int u, int v, short img[8][8]) {
  double s[8], r = 0; int x;
  for(x=0; x<8; x++) {
    s[x] = img[x][0] * COS(0, v) + img[x][1] * COS(1, v) + img[x][2] * COS(2, v) +
    img[x][3] * COS(3, v) + img[x][4] * COS(4, v) + img[x][5] * COS(5, v) +
    img[x][6] * COS(6, v) + img[x][7] * COS(7, v);
  }
  for(x=0; x<8; x++) r += s[x] * COS(x, u);
  return (short) (r * .25 * C(u) * C(v));
}

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

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

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;
}

```

Figure 7.11: High-level implementation of the CODEC module.

More specifically, we have multiplied the 64 cosine values by 32,678 and rounded the result to the nearest integer. The value 32,678 is chosen to allow us to store each value in 2 bytes of memory. To convert these integers back to floating point, we need to divide the stored values by 32,678.0. This is accomplished in the procedure called COS. This is a form

```

#define SZ_ROW      64
#define SZ_COL      64
#define NUM_ROW_BLOCKS (SZ_ROW / 8)
#define NUM_COL_BLOCKS (SZ_COL / 8)
static short buffer[SZ_ROW][SZ_COL], i, j, k, l, temp;
void CntrlInitialize(void) {}
void CntrlCaptureImage(void) {
  CodppCapture();
  for(i=0; i<SZ_ROW; i++)
    for(j=0; j<SZ_COL; j++)
      buffer[i][j] = CodppPopPixel();
}

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

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

```

Figure 7.12: High-level implementation of the CNTRL module.

of fixed-point representation, which is described later in this chapter. Thus the COS procedure handles the portions of the above equation involving the cosine and its arguments.

We have also implemented a procedure called *C* that simply corresponds to the function *C(h)* given above. All that remains now is the implementation of the nested summations. These summations are performed in the FDCT procedure. The inner summation is simply unrolled (i.e., we have expanded it into eight terms that are added together). The outer summation is implemented as two consecutive for loops. This choice of implementation, of course, is not unique. There are many ways to perform FDCT and the reader is encouraged, as an exercise, to implement these DCT functions with performance in mind.



```

int main(int argc, char *argv[]) {
    char *uartOutputFileName = argc > 1 ? argv[1] : "uart_out.txt";
    char *imageFileName = argc > 2 ? argv[2] : "image.txt";
    /* initialize the modules */
    UartInitialize(uartOutputFileName);
    CcdInitialize(imageFileName);
    CcdppInitialize();
    CodecInitialize();
    CntrlInitialize();
    /* simulate functionality */
    CntrlCaptureImage();
    CntrlCompressImage();
    CntrlSendImage();
}

```

Figure 7.13: Putting it all together is the main module.

The last module that we need in order to complete the implementation of our digital camera is the heart of the system, or what we have called the CNTRL, short for controller. The corresponding C file of the CNTRL module is called CNTRL.C and is shown in Figure 7.12. This module exports three procedures named *CntrlInitialize*, *CntrlCompressImage*, and *CntrlSendImage*. The *CntrlInitialize* procedure does nothing and is provided for consistency purposes only. The *CntrlCompressImage* procedure uses the other modules that we have described so far, namely the CCDPP and the CODEC to capture and perform FDCT and quantization on an image. Part of what this procedure has to do is to break the image into windows, or what we have referred to as blocks of  $8 \times 8$  pixels. Once a block is FDCT encoded, it is quantized and stored in memory. The *CntrlSendImage* procedure simply transmits the encoded image, serially, using the UART module.

Putting all this together is our main program, shown in Figure 7.13, that simply initializes all the modules and calls on the controller to capture, compress and transmit one image.

We now have a system-level model (executable specification) of our digital camera. We can experiment with this extensively. Note that any bugs we find here will be orders of magnitude easier to correct than if found at a later design stage.

## 7.4 Design

Design consists primarily of determining the system's architecture, and mapping the functionality to that architecture. The architecture consists of a set of processors, memories and buses. Processors may be any combination of single-purpose (custom or standard) or general-purpose processors. Multiple functions may be mapped to a single processor, and a function may be mapped to multiple processors. We'll say that an implementation is a particular architecture and mapping. The set of possible implementations defines the solution space. Note that the solution space is usually enormous. So where do we begin?

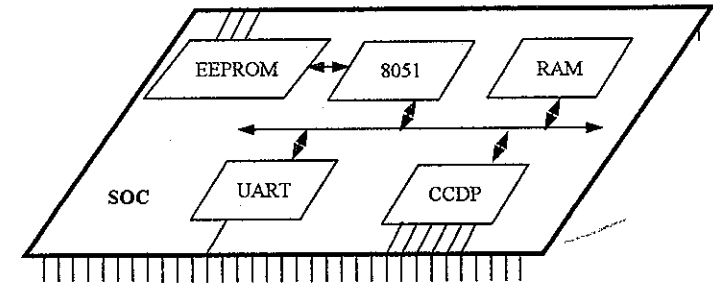


Figure 7.14: Block-diagram of our first implementation.

We might begin by examining a low-end general-purpose processor connected to flash memory, and trying to map all functionality to software running on that microprocessor. Such an implementation is often a good starting point for embedded system design, since the implementation will usually satisfy our power, size, and time-to-market constraints. If it also satisfies performance, then design is nearly complete. If this design doesn't satisfy constraints, then we could try a faster processor, we could use single-purpose processors for time-critical functions, or we could even rewrite the functional specification. We'll now start with such an implementation and then speed it up using different approaches.

### Implementation 1: Microcontroller Alone

Suppose we choose an Intel 8051 microcontroller (or similar such device) as our low-end processor. We determine that total IC cost (including NRE) would be about \$5, power well below 200 mW, and time-to-market only about three months. However, a rough analysis shows that there is no way an 8051 alone will satisfy our performance requirement of one image per second. Suppose the particular microcontroller we choose runs at 12 MHz and requires 12 cycles per instruction, meaning it executes one million instructions per second. Suppose we noticed during the execution of our earlier system-level model that CCD preprocessing consumed a lot of the computation time. Figure 7.9 shows the original code for the CCD preprocessor. The *CcdppCapture* function has a pair of nested loops that result in  $64 \times 64 = 4,096$  iterations per image. Looking at the code, we might estimate that each iteration will require about 100 assembly instructions during execution. Thus, this function alone will require  $4,096 \times 100 = 409,600$  instructions per image. This is nearly half of our budget of one million instructions per second, just to read the image alone, and not even considering the other more compute-intensive tasks of DCT and Huffman coding. Clearly, performance will be much worse than one image per second. We'll have to speed things up somehow.

### Implementation 2: Microcontroller and CCDPP

One method for improving performance is to implement a function using a custom single-purpose processor. Normally, we resist designing custom single-purpose processors

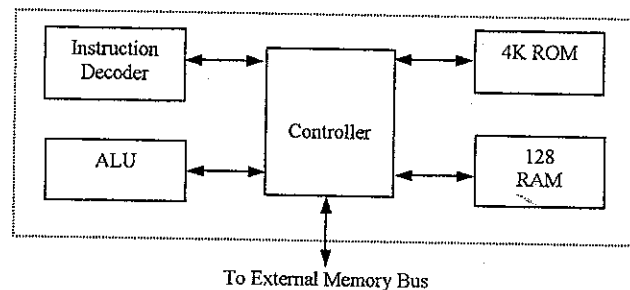


Figure 7.15: Block-diagram of the Intel 8051 processor core.

because they can increase NRE cost and time to market. However, the CCDPP function is a prime candidate for such implementation — not only is it taking up many microcontroller cycles, but it looks simple to implement as a single-purpose processor. There is no complicated arithmetic, so the datapath will be very simple, and the controller doesn't look like it will have many states either, since most of the cycles come from the  $64 \times 64$  loop iterations — these will likely translate to a couple of simple counters.

Thus, we decide to use an 8051 microcontroller coupled with a CCDPP single-purpose processor. Let's also implement a simple UART for the transmit-serially function. We'll also add an EEPROM for program memory and a RAM for data memory. Note that the CCDPP and UART processors could be implemented by finding standard components for each, but they are straightforward components, so let's implement them as custom instead. The CCDPP implements the zero-bias operations and interacts with the actual CCD chip, which resides external to our system-on-a-chip IC.<sup>2</sup> The rest of the functionality will be implemented in software on the microcontroller.

Let us briefly describe the three main processors depicted in Figure 7.14 in more detail. We begin with the microcontroller. A synthesizable implementation of this microcontroller, captured at the register transfer level (RTL) and written in VHDL, is available to us for integration into the rest of the system. A block-diagram of the main components of the 8051 is given in Figure 7.15. The controller fetches instructions from its read-only program memory and decodes them using the decoder component. The ALU component is used to actually execute arithmetic operations such as addition, multiplication, and division among many others. The source and destination of these operations are registers that reside in the internal RAM of the processor. Special data movement instructions are used to load and store data from external memory through the external memory bus. A C compiler/linker is used to

<sup>2</sup> We assume that the CCD chip resides external to our system-on-a-chip since given today's mainstream technology, and mostly due to fabrication process differences, combining a CCD with ordinary logic is not feasible.

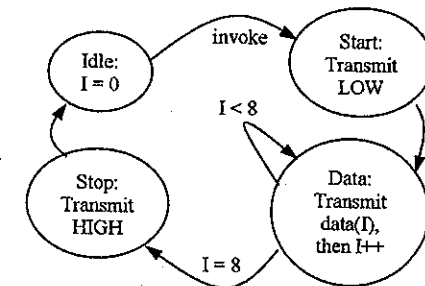


Figure 7.16: The UART single-purpose processor as an FSMD.

compile C programs for execution on our processor. The ROM is generated using a special program that reads the output of the C compiler/linker and outputs a VHDL description of the ROM.

The UART is a simple single-purpose processor. Its behavior is depicted in Figure 7.16 as a finite-state machine with data (FSMD). Normally, the UART is in its idle state. When invoked, it transitions into the start state, where it transmits a 0 indicating the start of a byte transmission. Then, it transitions into the data state, where it sends the 8 bits of the byte being sent. Then, it transitions into the stop state, where it transmits a 1 indicating the stop of the byte transmission. Finally, it transitions back into the idle state, ready to repeat the processes, when summoned again. Since the UART is memory mapped to the processor's memory address space, it is invoked when the processor executes a store instruction with the UART's enable register as its target memory location. Of course, the UART is constantly monitoring the address-bus, and when it detects the enable register's address, it captures the data on the data-bus and starts the transmission processes as just described. Note that we will use memory-mapped I/O for communication between the 8051 processor and any other single-purpose processor in our system. Since the 8051 processor's address space is 16 bits wide, we use lower memory address, those starting at 0 and going up, for RAM and upper memory address, and we use those starting at 65,535 and going down for memory-mapped I/O devices.

The CCDPP is one of the single-purpose processors that has been implemented in hardware. The FSMD of the CCDPP is depicted in Figure 7.17. Internally, the CCDPP single-purpose processor has a buffer, labeled *B*, and three variables called *R*, *C*, and *Bias*. The variables *R* and *C* are used as row and column indices. The variable *Bias* holds the zero-bias error for each of the rows as the rows are processed. The FSMD works as follows. Once invoked, it transitions into the *GetRow* state where it reads from the actual CCD a complete row including the last two blacked-out pixels. (For details, refer to the description of a CCD given at the beginning of this chapter.) Then, the FSMD transitions into the *ComputeBias* state where it computes the bias of the current row and stores it into the *Bias* variable. In the next state, called *FixBias*, the FSMD iterates over the same row subtracting away the bias from each element in that row. In the next state, called *NextRow*, the row index is incremented

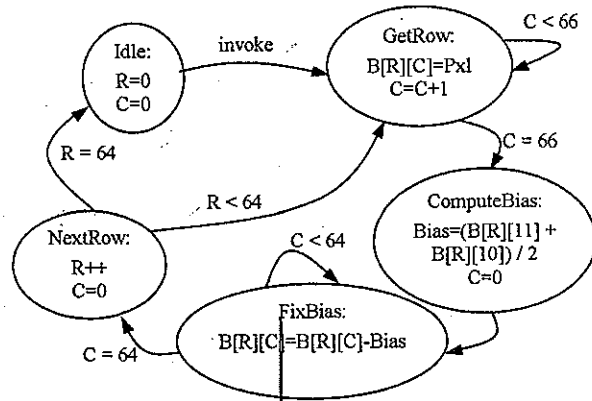


Figure 7.17: The CCDPP single-purpose processor as a FSM.

and the process either repeats, reading the next row, or stops when the entire image is processed. We assume that, as with the UART, this single-purpose processor is connected to the 8051 processor's memory bus with the content of the internal buffer mapped to upper memory addresses of the processor.

We now have all the components of our system-on-a-chip and are ready to connect things together making up our digital camera. This is accomplished through the 8051's memory bus, as stated before. The 8051 memory bus uses a simple read and write protocol and is composed of an 8-bit data-bus, a 16-bit address-bus, a read control signal and a write-control-signal. A memory read works as follows. The processor places the memory address on the address-bus, then asserts the read control signal for exactly one clock-cycle and reads the data from the data-bus one clock-cycle later. The device that is being read, either the RAM or one of our memory mapped single-purpose processors, when detecting that the read control signal is asserted, and after checking the content of the address-bus, places and holds the requested data on the data-bus for exactly one clock cycle. A write operation works in a similar fashion. The processor places the memory and the data on the address and data-bus, respectively. Then, it asserts the write control signal for exactly one clock cycle. The device that is being written, when detecting that the write control signal is asserted, and after checking the content of the address-bus, reads and stores the data from the data-bus.

Now that we have the hardware portion of our design implemented, we need to write the software to complete the project. Fortunately, our executable specification will provide the majority of the code that we need. In fact, we will maintain the same structure of the code (i.e., we will keep the same module hierarchy, procedure names, and main program). The only thing that needs to be done is to design the UART and CCDPP custom single-purpose processors. This is rather easy to do. All that we need to do is replace the code in these

```

static unsigned char xdata U_TX_REG_at_65535;
static unsigned char xdata U_STAT_REG_at_65534;
void UARTInitialize(void) {}
void UARTSend(unsigned char d) {
    while( U_STAT_REG == 1 ) {
        /* busy wait */
    }
    U_TX_REG = d;
}
  
```

Figure 7.18: Rewriting the UART module to utilize the hardware UART.

procedures with memory assignments to the respective hardware devices. Let us show this with the UART example. The code for this module is given in Figure 7.18. Here we have defined two variables, called `U_TX_REG`, and `U_STAT_REG`. There are two keywords used in defining these two variables that you may not recognize. The first one, called `xdata`, instructs our compiler to place these variables in the external memory; in other words, the compiler will generate code that will load and store these variables over the external memory bus of the processor. The second keyword, called `_at_`, instructs our compiler to place these variables at the specified memory address. These two keywords allow us to declare a variable such that when read or written will cause appropriate read or write operations to be performed on the bus. Now, all we have to do to send a byte using our UART single-purpose processor is write the byte to be sent to the `U_TX_REG` causing it to be invoked. But since our processor may be much faster than the UART, we need to first make sure that the UART is in its idle state. This is accomplished by the while loop. Having designed our UART such that we can check whether its busy or not, we can busy-wait until it becomes idle before sending the next data byte. The implementation of the CCDPP module is similarly modified to utilize the CCDPP single-purpose processor. The rest of the modules are untouched.

Now we can compile and link all our software modules and obtain the final program executable. This program executable is then translated into the VHDL representation of the ROM using a ROM generator. All that remains is to test our entire system-on-a-chip. This is done using a VHDL simulator program. A VHDL simulator takes as input the VHDL files, making up our system, and functionally simulates the execution of the final IC by interpreting the descriptions. By simulating, we are able to learn whether our design is functionally correct. Moreover, we can also measure the amount of time, or clock-cycles, that it takes to process a single image. This is our first metric of interest, namely, performance. Figure 7.19(a) shows how after simulating the VHDL models, we obtain the execution time. Figure 7.19(b) shows how we synthesize the high-level VHDL models and obtain the gate-level description of the corresponding circuits. Then, we simulate the gate-level models to obtain the intermediate data necessary to compute the power consumption of the circuit. Figure 7.19(c) shows how by adding the number of gates, we obtain the total area of the chip.

Once we are satisfied that our design functions correctly, we can use our synthesis tool to translate the VHDL files down to an interconnection of logic gates. A synthesis tool is like a

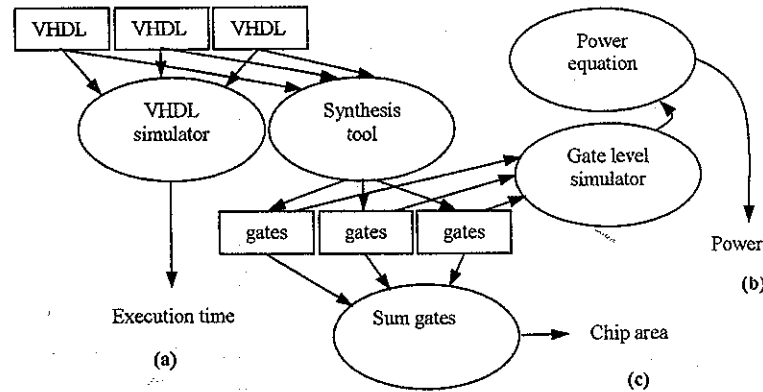


Figure 7.19: Obtaining design metrics of interest; (a) performance, (b) power, (c) area.

compiler for single-purpose processors. It reads a VHDL file and translates it to a corresponding gate-level description. You'll learn more about this process in a later chapter of this book. At this stage, these gates can be sent to an IC fabrication company to make our IC chip. But what we are interested in is counting the total number of gates to get an idea of how big our design is. This will tell us how big of an area we need to implement the digital camera, or the third metric of interest. To obtain the power consumption, our second metric of interest, we simulate the gate-level description of the digital camera and keep track of the number of times these gates switch from zero to one and from one to zero. Recall that we can estimate power consumption if we know the amount of switching that takes place in a circuit.

We can now analyze our first implementation using the approach outlined in Figure 7.19. Using simulation, we have measured the total execution time for processing a single image to be 9.1 seconds. The power consumption is measured to be 0.033 watt. The energy consumption is  $9.1 \text{ s} \times 0.033 \text{ watt} = 0.30 \text{ joule}$ . The area is measured to be 98,000 gates.

### Implementation 3: Microcontroller and CCDPP/Fixed-Point DCT

The previous implementation does not achieve 1-image-per-second processing. Looking at the execution of the previous implementation, we see that most of the microcontroller computer cycles are spent performing the DCT operation. Thus, we could consider pulling this compute-intensive function out from software to custom hardware, as we did for the CCD preprocessor. However, unlike the CCD preprocessor, the DCT functionality is fairly complex and thus will likely require more design effort. We can instead speed up the DCT functionality by modifying its behavior.

Recall that each DCT operation involves numerous floating-point operations. Actually, for each pixel that is transformed, about 260 floating-point operations are performed. There are  $64 \times 64 = 4,096$  pixels that are encoded, for a total of about one million floating-point

operations. To make matters worse, our processor is an 8-bit processor with no floating-point support; thus, the compiler needs to emulate each of these floating-point operations. Floating-point emulation is performed as follows. The compiler generates procedures for each of the floating-point operations, such as multiplication and addition. These procedures may execute tens of integer instructions in order to perform a single floating-point operation. Then, when the compiler encounters floating-point operations in the source file, it places a call to these compiler-generated procedures. Consequently, our one million floating-point operations will require ten million or more integer operations. In addition, our program will be larger, since it has to accommodate the compiler-generated procedures.

We can thus consider speeding up the CODEC module to use fixed-point arithmetic. We hope to reduce the total number of integer instructions required to encode each pixel. Our implementation is shown in Figure 7.20. Let us first describe how fixed-point arithmetic works. In fixed-point arithmetic, we use an integer to represent real numbers. The bits within this integer are interpreted as follows. We use a constant and known number of these bits to represent the portion of a real number after the decimal and the rest of the bits to represent the portion of the real number before the decimal point.

In our implementation of the CODEC, we have chosen to use 6 bits to represent the fractional part of all arithmetic operations. The choice here has to do with the accuracy that we desire. The more bits we use for the portion after the decimal, the more accurately we can represent a real number. However, this will leave us fewer bits to represent the portion of the real number before the decimal point (i.e., the magnitude of the real number).

Once we have chosen the number of bits to represent the portion after the decimal point, a.k.a. the fractional part, we can translate any constant to the fixed-point representation. For example, imagine that we are using 8-bit integers. Let us use 4 bits to represent the fractional part. The fixed-point representation of the real value 3.14 would be 50, or 00110010. We obtain 50 by multiplying the real value, 3.14, by  $2^4$  raised to the number of bits we are using for the fractional part,  $2^4 = 16$ , and rounding it to the nearest integer,  $3.14 \times 16 = 50.24 \approx 50$ . Note that the 4 least significant bits equal 2. Since there are a total of 16 possibilities, each would represent .0625. Given that we have 2, we get  $2 \times 0.0625 = 0.125$ . The four most significant bits encode the value 3, which when added to our fractional part, gives 3.125. Of course, our representation is not exact but close. We can improve this by using more bits for the fractional part. In fact, the cosine table in Figure 7.20 gives the fixed-point representation of the cosine values, using 8-bit integers.

Now that we know how to represent a real number using integers, we have to define the two operations that are used in our calculations, namely addition and multiplication. Addition is straightforward. All that we have to do is add the integers. For example, assume that we have 3.14 encoded as 50, or 00110010 and 2.71 as 43, or 00101011. To add these two together, we add the integers 50 and 43 to obtain 93, or 01011101. Converting this back to a real, we get  $5 + 13 \times 0.625 = 5.8125$ . This number is close to the actual value, which is 5.85, but not exact, as expected.

Similarly, with multiplication, we can multiply the two fixed-point values to obtain our result. But, at this point we need to perform an additional operation. Let us multiply the value 3.14 encoded as 50, or 00110010 and 2.71 as 43, or 00101011. From this we obtain 2,150, or

```

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 const char ONE_OVER_SQRT_TWO = 5;
static short xdata inBuffer[8][8], outBuffer[8][8], idx;
static unsigned char C(int h) { return h ? 64 : ONE_OVER_SQRT_TWO; }
static int F(int u, int v, short img[8][8]) {
    long s[8], r = 0;
    unsigned char x, j;
    for(x=0; x<8; x++) {
        s[x] = 0;
        for(j=0; j<8; j++) s[x] += (img[x][j] * COS_TABLE[j][v]) >> 6;
    }
    for(x=0; x<8; x++) r += (s[x] * COS_TABLE[x][u]) >> 6;
    return (short) (((r * ((16 * C(u)) >> 6) * C(v)) >> 6) >> 6) >> 6;
}

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

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

```

Figure 7.20: Fixed-point implementation of the CODEC module.

100001100110. Note that, when multiplying two 8-bit integers, we can expect the result to be 16 bits wide. What we have to do to obtain our final result is to discard the lower 6 bits of our 16-bit result, obtaining 10000110. Converting this back to a real, we get,  $8 + 6 \times 0.0625 = 8.375$ . The number is close to the correct value, which is 8.5094, but not exact, as expected.

The biggest difficulty with fixed-point arithmetic is to ensure that the resulting values, after performing addition and multiplication operations, do not exceed the bit-width of the integers that are being used. Therefore, it is important to consider the intervals, or range, of the real values that are being operated on. We have applied the fixed-point arithmetic scheme presented here in recoding the CODEC. This time our CODEC uses integer operations only and we expect it to execute faster than our first implementation.

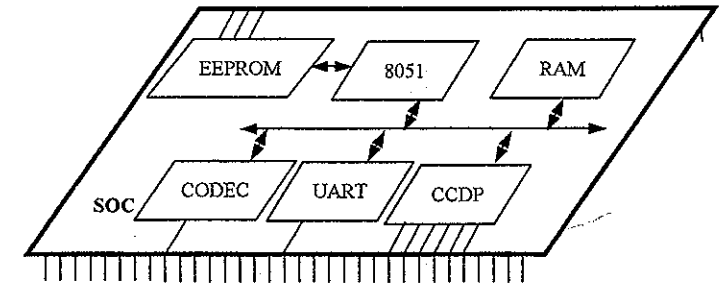


Figure 7.21: Block diagram of our fourth implementation.

We can now analyze our second implementation using the approach outlined in Figure 7.19. Using simulation, we have measured the total execution time for processing a single image to be 1.5 seconds. The power consumption is measured to be 0.033 watt, the same as before. The energy consumption of this design is  $1.5 \text{ s} \times 0.033 \text{ watt} = 0.050 \text{ joule}$ . This is means that our batteries will last six times longer when compared to the previous design! The area is measured to be 90,000 gates. We have improved performance by a factor of six and reduced the chip area by about 8,000 gates over the previous design. The gate reduction is because our program no longer needs to emulate the complex floating-point operations, thus requiring less memory for storing the corresponding code.

#### Implementation 4: Microcontroller and CCDPP/DCT

Our third implementation's performance is close to that required by our specification, achieving 1.5 seconds per image. Let us try to improve performance further to obtain 1 second per image. In our next implementation, we will resort to implementing the CODEC in hardware. That means that we will design a single-purpose processor that performs the DCT operation on a block of  $8 \times 8$  pixels. The block diagram of our new system-on-a-chip is given in Figure 7.21. Designing the processor for the CODEC may take some time to get correct.

To use this CODEC, we will need to make some changes to our software. Specifically, we will need to change the CODEC module, as we did the UART and the CCDPP modules. The code is presented in Figure 7.22. We have designed our hardware CODEC to have four memory-mapped registers. Two of these registers, called *C\_DATAI\_REG* and *C\_DATAO\_REG*, are used to push and pop a block of  $8 \times 8$  pixels into and out of the CODEC. Another register, called *C\_CMND\_REG*, is used to command the CODEC. Specifically, writing a one to this register will invoke the CODEC. The last register, called *C\_STAT\_REG*, can be polled in software to tell when our CODEC is done encoding a block of pixels. The actual implementation of the CODEC is a direct translation of our fixed-point version of the CODEC written in C, and used in our second implementation, into VHDL. Using a single-purpose processor for encoding data, we expect to improve our execution time, and thus satisfy our timing constraints.

```

static unsigned char xdata C_STAT_REG_at_65527;
static unsigned char xdata C_CMD_REG_at_65528;
static unsigned char xdata C_DATAI_REG_at_65529;
static unsigned char xdata C_DATAO_REG_at_65530;
void CodecInitialize(void) {}
void CodecPushPixel(short p) { C_DATAO_REG = (char)p; }
short CodecPopPixel(void) {
    return ((C_DATAI_REG << 8) | C_DATAI_REG);
}
void CodecDoFdct(void) {
    C_CMD_REG = 1;
    while( C_STAT_REG == 1 ) { /* busy wait */ }
}

```

Figure 7.22: Rewriting the CODEC module to utilize the hardware CODEC.

We can now analyze our final implementation using the approach outlined in Figure 7.19. Using simulation, we have measured the total execution time for processing a single image to be 0.099 seconds. The power consumption is measured to be 0.040 watt. Notice that the power consumption increased, because our chip is now doing more (i.e., there are multiple processors working). The energy consumption of this design is  $0.099 \text{ s} \times 0.040 \text{ watt} \approx .00040$  joule. This means that our batteries will last 12 times longer than the previous design. The area is measured to be 128,000 gates. We are now well under 1 second, processing one image in about 1/10th of a second (approaching video camera speed now). However, we have increased the IC size significantly. This implementation certainly meets our timing requirements. More importantly, if we design the DCT ourselves, we will likely increase our NRE cost and time-to-market. If we purchase an existing DCT, we may increase our IC cost.

We have summarized our results in Figure 7.23. In designing an embedded system, many other metrics need to be considered. As with any other commercial product, in addition to engineering issues, a careful cost analysis of a system must be made.

Implementation 3 is close in terms of performance but a little slow, and consumes more energy, but is likely much cheaper and will be built in less time. Implementation 4 meets the performance (by a lot) and consumes much less energy (by a lot), but will be more expensive and may result in missing our time-to-market cutoff. Which is better? It's a choice that our company will have to make. As mentioned in Chapter 1, a key challenge facing the embedded system designer is to construct an implementation that simultaneously optimizes numerous design metrics. We can't always get what we want!

	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

Figure 7.23: Summary of design metrics.

## 7.5 Summary

We have introduced a digital camera and have described its various components. These components capture, digitize, process, and store images, among other things. As part of our presentation, we have described JPEG encoding, to a limited extent. We have specified our design project in an informal format using English as well as an executable specification. We have described three design metrics of interest, namely, performance, power consumption, and chip area. For each of these metrics, we have suggested optimization techniques. In the second part of the chapter, we have described several successively improved implementations. The first implementation we considered used a single microcontroller, but would have been far too slow. Our second implementation used a coprocessor to speed things up a bit, but we were still much too slow. Our third implementation gave up some accuracy during compression by using fixed instead of floating-point numbers. It came close to our performance constraint, but was still a bit slow. Our last implementation involved another coprocessor for compression, meeting performance easily but costing more and taking more design time. The better of these last two implementations is not clear.

The executable specification and the three latter implementations are available in source code format on this book's Web page.

## 7.6 References and Further Reading

- C. Wayne Brown and Barry J. Shepherd. *Graphics File Formats — Reference and Guide*. Connecticut: Manning Publications Company, 1995.
- P. van der Wolf, P. Lieveise, M. Goel, D.L. Hei, and K. Vissers. An MPEG 2 Decoder Case Study as a Driver for a System-level Design Methodology. International Workshop on Hardware/Software Co-Design, March 1999.

## 7.7 Exercises

- 7.1 Using any programming language of choice, (a) implement the FDCT and IDCT equations presented in section 7.2 using double precision and floating-point arithmetic. (b) Use the block of  $8 \times 8$  pixel given in Figure 7.2(b) as input to your FDCT and obtain the encoded block. (c) Use the output of part (b) as input to your IDCT to obtain the original block. (e) Compute the percent error between your decoder's output and the original block.
- 7.2 Assuming 8 bits per each pixel value, calculate the length, in bits, of the block given in Figure 7.3(b).
- 7.3 Using the Huffman codes given in Figure 7.5, encode the block given in Figure 7.3(b). (a) What is the length, in bits? (b) How much compression did we achieve by using Huffman encoding? Use the results of the last question to calculate this.

- 7.4 Convert 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, and 1.9 to fixed-point representation using (a) two bits for the fractional part, (b) three bits for the fractional part, and (c) three bits for the fractional part.
- 7.5 Write two C routines that, each, take as input two 32-bit fixed-point numbers and perform addition and multiplication using 4 bits for the fractional part and the remaining bits for the whole part.
- 7.6 Using any programming language of choice to (a) implement the FDCT and IDCT equations presented in Section 7.2 using fixed-point arithmetic with 4 bits used for the fractional part and the remaining bits used for the whole part, (b) use the block of  $8 \times 8$  pixels given in Figure 7.2(b) as input to your FDCT and obtain the encoded block, (c) use the output of part (b) as input to your IDCT to obtain the original block, and (d) compute the percent error between your decoder's output and the original block.
- 7.7 List the modifications made in implementations 2 and 3 and discuss why each was beneficial in terms of performance.

## CHAPTER 8: State Machine and Concurrent Process Models



- 8.1 Introduction
- 8.2 Models vs. Languages, Text vs. Graphics
- 8.3 An Introductory Example
- 8.4 A Basic State Machine Model: Finite-State Machines
- 8.5 Finite-State Machine with Datapath Model: FSM D
- 8.6 Using State Machines
- 8.7 HCFSM and the Statecharts Language
- 8.8 Program-State Machine Model (PSM)
- 8.9 The Role of an Appropriate Model and Language
- 8.10 Concurrent Process Model
- 8.11 Concurrent Processes
- 8.12 Communication among Processes
- 8.13 Synchronization among Processes
- 8.14 Implementation
- 8.15 Dataflow Model
- 8.16 Real-Time Systems
- 8.17 Summary
- 8.18 References and Further Reading
- 8.19 Exercises

### 8.1 Introduction

We implement a system's processing behavior with processors. But to accomplish this, we must have first described that processing behavior. One method we've discussed for describing processing behavior uses assembly language. Another more powerful method uses a high-level programming language like C. Both methods use what is known as a sequential