

CPRE 488X

LAB04

Lab Conventions

The following conventions are used throughout the labs:



Location of additional information or tips.



An important error or step that you should be aware of.



A reference to a chapter in the book.



An item to write in your lab notebook.

Lab Overview

During the previous labs, we learned how to use our Virtex-II Pro platform and its tools. Over the next three weeks you will adapt a simple JPEG encoding algorithm to our platform. This adaptation process will take you from a software-only implementation running on the PowerPC (like you did in labs 1 and 2) to a mixed software-hardware implementation (like the one you saw in lab 3).

In the previous lab, you learned about system performance, and you saw how to measure it. In Lab 3, you measured the performance of a simple multiplication and used these measurements to improve the design. In Lab 4, we will look at a much more realistic scenario, where you are given a software implementation of a JPEG encoding algorithm and you are to realize this algorithm on the Virtex-II Pro platform. To do this, you must evaluate your design and, like you did on Lab 3, you must find where and how to improve it in order meet the design specifications.

In Lab 3 you used the function `XTime_GetTime()` to measure the performance of your application. Here you will use GNU GCC and GNU gprof for performance evaluation. While you can use `XTime_GetTime()` for a simple function like you did on Lab 3, for a more accurate application-wide evaluation, you need tools like GCC and gprof.

In Lab 4 you will use GCC to compile the application in such a way that it will be possible to collect performance data at run-time. Then, you will use gprof to analyze this data and effectively profile your application. With this profile you can find where the

critical sections of the application are. By improving these sections, you will develop a better application.

In some cases moving a function to hardware would be difficult and expensive, and you may want to consider what options you have by strictly modifying the software. Software optimizations will be explored in Lab 5.

Once you have optimized the software and still don't meet your requirements, the next step would be to consider moving these critical sections into hardware. This last step is known as hardware acceleration, which is explored in Lab 6.

Prelab



You will be using GCC and gprof from GNU in this lab. Make sure you have read through the documentation for these tools¹. Read section 5.6 from the book and answer the following questions:

1. Why is it difficult to precisely determine the execution time of a program?
2. List 3 ways to measure a program's performance.
3. Talk about the role of cache in a program's execution. Focus on how enabling cache changes the program's execution time and performance.
4. List the steps for "Optimizing for Execution Speed".

Profiling

As you know from section 5.6 of your book, you can profile an application to find the hot spots. Profiling is the process of collecting data about the runtime of an application. By profiling your application, you can identify the areas of your code which consume the most time.

Profiling is a useful tool when developing an application. The profile data is collected during run-time, so an analysis of the application's actual execution can be made -- as opposed to a simulation which can have varying levels of accuracy.

Remember that a program's execution depends as much on the way the code is written as on the input to the application at any given time. Therefore, you must select the input to your application carefully when profiling. The input for the JPEG encoder will be a raw picture to be encoded, and for our encoder almost any picture would provide a good profile. An example of a bad choice might be a picture that is completely one color, since the JPEG encoding would find this picture very simple to encode. This might skew your profiling and make it seem like some functions are taking a lot of time, when it turns out that with a more complex picture other functions would take the most time. On the other hand, if your camera is supposed to be optimized for taking pictures from a telescope, perhaps a mostly-black picture would be an appropriate input for profiling. The underlying idea here is that you need to **optimize for your application**.

¹ <http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html>, official GNU gprof site.

In some cases it is not possible to profile your application on the actual hardware. This might be the case in a very small embedded system that does not have enough memory to store profiling information, as one example. In this case you can get approximate results by running the algorithm on a more powerful computer – although you must be careful to realize that the results will only be approximate. Remember from Lab 3 that something as small as memory access times, cache performance, and other factors can drastically change the time it takes to run an algorithm.

In this lab, you will use the GNU tools GCC and gprof to generate the profile data and analyze it. The workflow with these tools can be summarized in three steps:

1. Compiling the application for profiling.
2. Executing the application.
3. Analyzing the profile data that was generated during the execution.

First, we will work with a sample application to understand the GNU profiling workflow. Let's start by compiling an application on a Linux system. We will run the application normally once, and then we will go through the steps to generate a profile.

Download the file `profile_example.tar.gz` off the course website. Save it to your U: drive. Open a TeraTerm Pro and connect to one of the ECE linux computers (`linux-1.ece.iastate.edu`, `linux-2.ece.iastate.edu`, etc.).



Write down in your lab notebook which computer you connected to.

Go to the file you just downloaded and decompress it. It will generate a folder called `profile_example` containing two files:

1. `Makefile`
2. `profile_example.c`

Run the gcc compiler to generate an executable file.

```
gcc -o profile_example profile_example.c
```

Then run the application.

```
./profile_example
```

Be sure that the program runs. You will see text scrolling on your terminal during the program execution. If the program executes correctly, we can begin profiling. In order to use profiling, you must recompile the program passing special options to gcc. To enable profiling use the '-pg' option. Then run the application again.

```
gcc -g -pg -o profile_example profile_example.c
./profile_example
```

In your working directory, you will see a new file named `gmon.out`. This file contains the profile data collected during the program execution. Now, let's convert `gmon.out` to a human-readable output.

```
gprof profile_example gmon.out > output.txt
```

If all went well, you should have a file called `output.txt` on your current working directory which contains a table somewhat like the one below, plus a lot more information. Let's look at what all of this means². This table is known as the Flat Profile.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
46.96	221.00	221.00	16	13.81	13.81	foxtrot
21.13	320.43	99.43	12	8.29	8.29	golf
16.45	397.83	77.40	14	5.53	19.34	echo
7.64	433.77	35.94	2	17.97	17.97	beta
5.29	458.68	24.91	2	12.46	12.46	hotel
1.17	464.20	5.52	2	2.76	214.13	delta
0.77	467.84	3.64				main
0.58	470.59	2.75	2	1.38	19.34	alpha
0.00	470.59	0.00	2	0.00	0.00	charlie

The last column lists the functions, ordered by execution time. The function that spends the most time is at the top. More information on the table layout is appended at the end of the `output.txt` file. You can also find detailed information on the GNU `gprof` website³. One thing worth noticing is the first line: "Each sample counts as 0.01 seconds". This line indicates the sample frequency used to collect the profile data during the execution. You will read more about this topic on the section of Statistical Sampling Error on `gprof`'s website.



From what you read on the Statistical Sampling Error of `gprof`, explain how the run-time figures are collected during execution.

Give answers to the following questions.

What is the method mentioned to improve the collected run-time figures?

Can you think of another way to get better results on the presence of statistical error?

Now that you know that run-time figures are subject to statistical error, you must rerun your program in such a way to minimize this error. You can use the `make` file that is in your current working directory. This is how you use it:

- `make build`, to compile your application with profiling enabled.
- `make run`, to collect profile data using the method described on `gprof`'s website.
- `make analyze`, to generate a text file called `prof_output.txt` where your profile summary will be saved.



Write down answers to the following questions.

² Note that this is a general example and your table will look different from this one.

³ http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_node/gprof_20.html#SEC20

1. What is the total run time of profiling_example?
2. Which function consumes the most run time?
3. Which function call consumes the most time on its own?
4. Which function call consumes the most time including children?

The next table that we see is the call graph. This table shows how much time was spent in each function and its children, as well as statistics on the number of times each child was called.

Index	% time	self	children	called	name
[1]	100.0	3.64	466.95		<spontaneous>
		5.52	422.74	2/2	main [1]
		2.75	35.94	2/2	delta [2]

[2]	91.0	5.52	422.74	2/2	alpha [7]
		5.52	422.74	2	main [1]
		176.83	193.38	2/2	delta [2]
		27.62	0.00	2/16	echo <cycle 1> [4]
		24.91	0.00	2/2	foxtrot [5]
					hotel [9]



Using Visio (available in the lab) or some other drawing program, create a call tree. The call tree should show each function as a labeled node, with arrows going from the calling functions to the called functions. It should start from main and branch into levels of function calls. Include this with your lab notebook.

Now that you've seen how to profile your code with the GNU workflow, it is time to move back to our platform. In this lab, you will be working with the digital camera.

Digital Camera

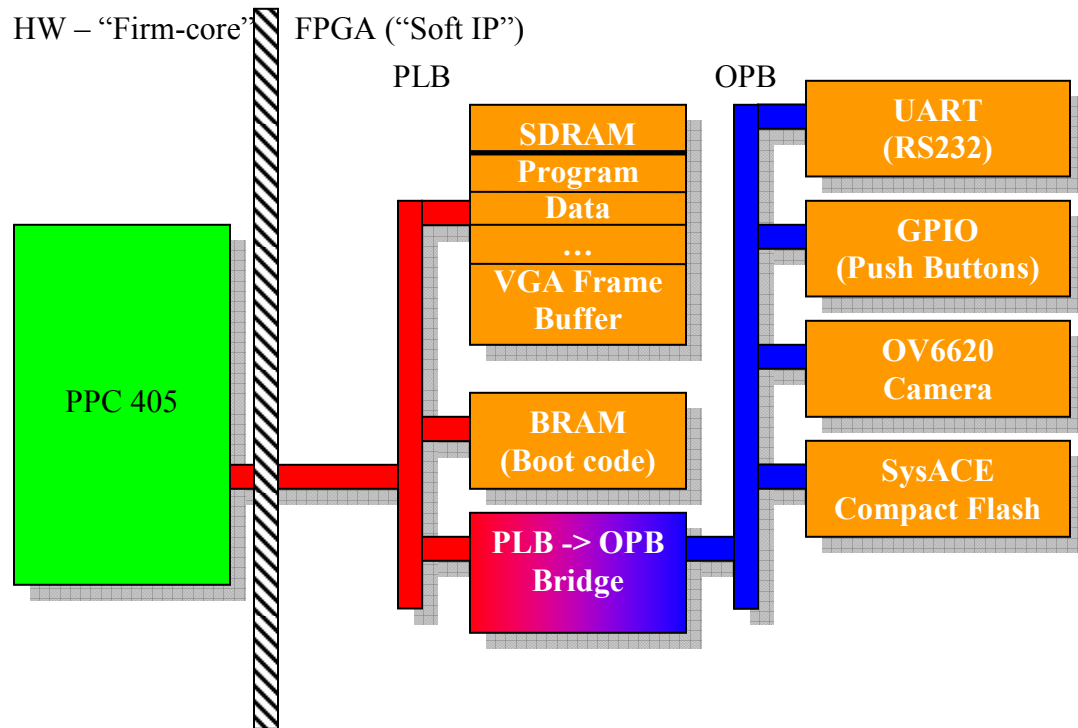


Figure 1. Block Diagram of the Digital Camera

Figure 1 shows a block diagram of the Digital Camera System on the Virtex-II platform. Take a moment to inspect this diagram. Notice a few new components:

- VGA Port: this is the VGA port that you can see on the board. You will use this to see the picture preview on your monitor. Ask the lab TA how to set it up if it's not connected yet.
- OV6620 Camera: this is the small camera that is attached to the board. **This camera must have the power connected correctly. If it's not properly set up, the camera can and will burn out. Please ask the lab TA if you have any questions on how to set it up (see Figure 2) and make sure the TA has double-checked the connections before you power up the board.**
- SystemACE Compact Flash: the camera saves the encoded pictures into a compact flash card. You will need the compact flash card for this lab and all subsequent labs. Please ask the lab TA for a card.



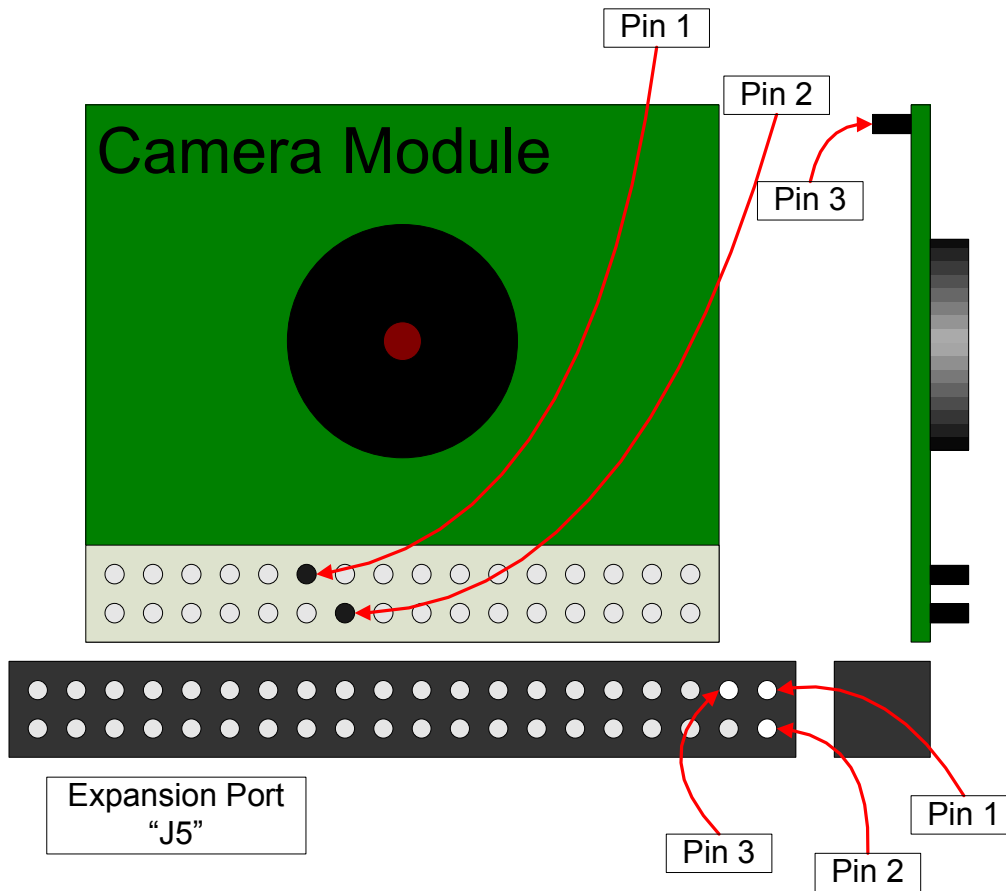


Figure 2. Camera Setup Diagram

Figure 2 is a diagram of the camera module that you will use on this lab. Please take a minute to identify the pins 1 through 3 on the system. You must connect the power pins as follows:

- Pin 1 on the board to Pin 1 on the camera,
- Pin 2 on the board to Pin 2 on the camera,
- Pin 3 on the board to Pin 3 on the camera.



Remember to verify with your TA before you apply power to the board. If you connect the camera in the wrong way **it will be damaged**.

Setting up the XUP Board

For this lab, you will not use a copy of the previous lab but instead will get a pre-built system. From the class website, download lab4.zip and unzip it into your working directory, “C:\temp\” This will create a new folder with the Lab 4 project in it; open this new project.



Remember from previous labs that, by default, a project will be set to load into BRAM. Our project, however, is too large to load directly into the BRAM, so make sure it is set to use DDR RAM.

Download the new project to the board and build the application.

This application uses TeraTerm to send status messages to the user. Open TeraTerm and set it up accordingly, remember that you can find the baud rate on the MHS file. Open XMD download and run the application you just build, remember that you can use **Tools->Get Program Size**, to get the location of executable.elf.

Now run the application. You will see “Enable single-shot mode...done.” on the TeraTerm window once the camera set up has finished.

Switch your display to see the camera preview. You do this by pressing the button label as “-” on the monitor. Pressing this button switches between the computer and the board. Once you have a picture lined up, press the **RIGHT** push button on the board. It will take a few seconds for the system to take a picture, watch for:

```
done
Total time: ## ms
```

on TeraTerm to know when the camera is done. At this point you can remove the card from the board, insert the compact flash card into the computer, and take a look at your wonderful picture.

Profiling In XPS

At this point, you should have a working system. Pictures are saved onto the Compact Flash Card, which we can view to confirm that the application is working properly. This next section will cover how to profile your application inside XPS.

Just like before, we will collect the profiling data at run time; XPS refers to this method as *software intrusive* profiling. Look at the Platform Studio User Guide, chapter 10, for more information about profiling with XPS. The first step, when profiling in XPS, is to enable software intrusive profiling. To do this, go to **Project -> Software Platform Settings**, and under the **Library/OS Parameters** tab, set **enable_sw_intrusive_profiling** to *true* and set **profile timer** to *none*. This last option tells gcc to use the PowerPC built in timer for profiling. Make sure your options look like the ones on Figure 3 and Click OK.

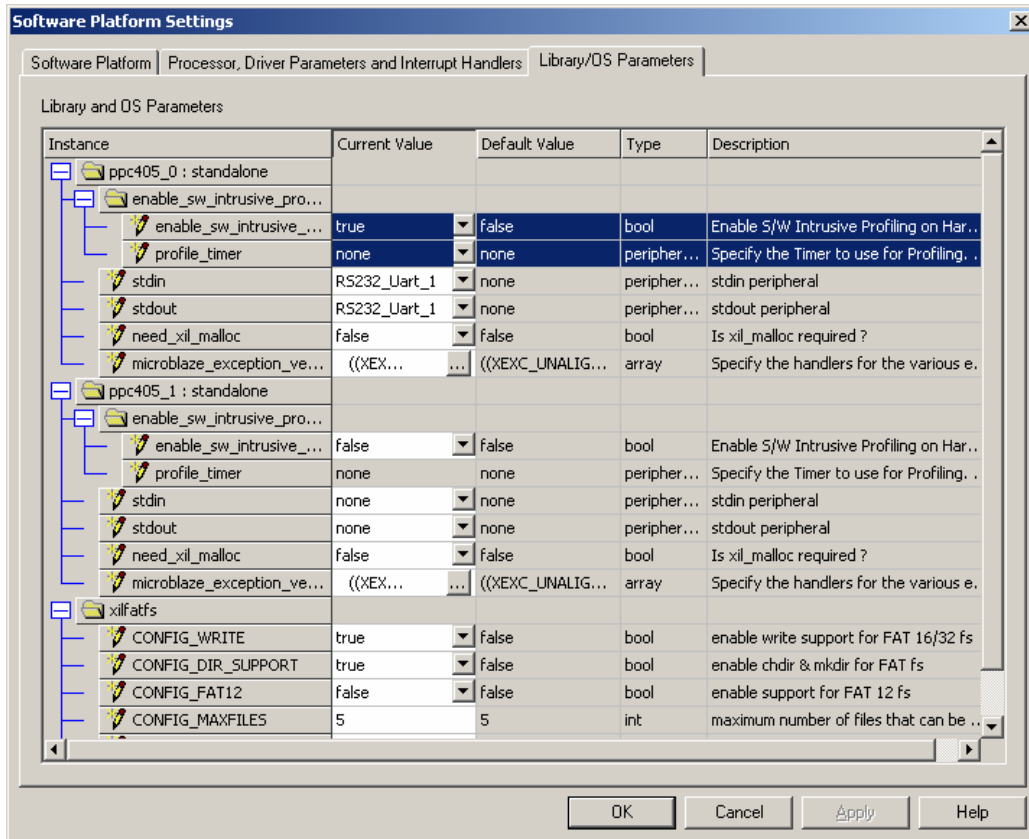
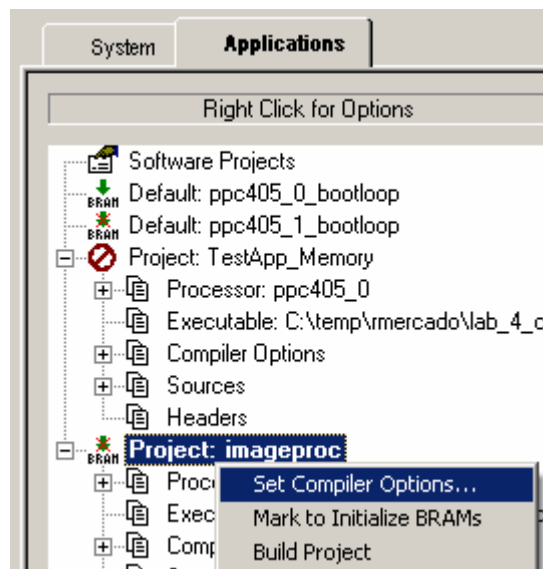


Figure 3. Library/OS Parameters

Now, we will add a flag for the compiler to add profiling tags into our binary.

Under the **Applications** tab, right-click on the project name and select **Set Compiler Options**.



Under the **Advanced** tab, add the flag **-pg** to the **Program Sources Compiler Options**, just like on Figure 4. Click OK.

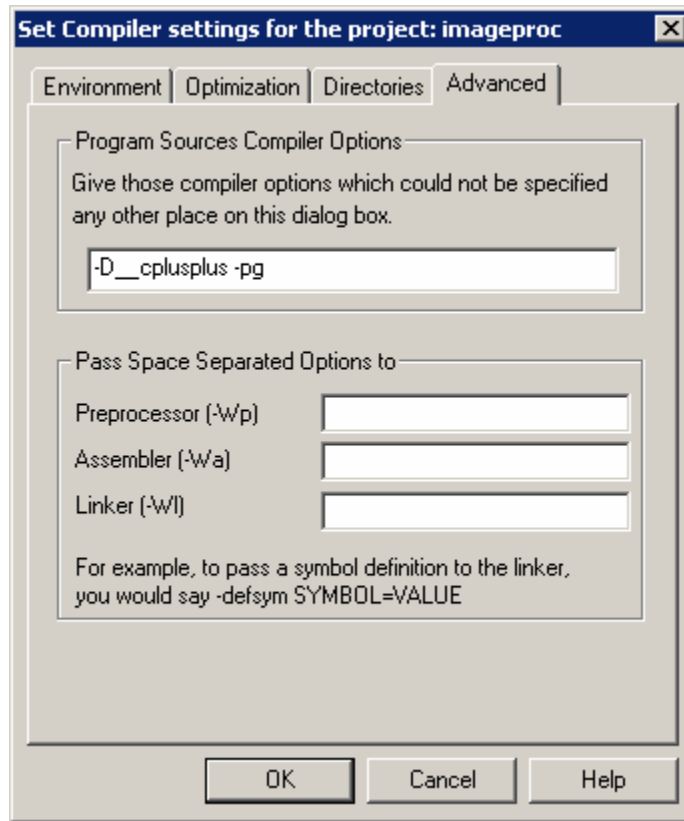


Figure 4

Now build your application.

Before you can download your new application into the PowerPC you must configure XMD for profiling. This includes indicating the sample frequency and defining the memory to use for profiling. Connect to the target in XMD and type⁴:

```
profile -config sampling_freq_hz 10000 binsize 4 profile_mem 0xf000000
```

Now download your application.

This time XMD should display the configuration that you have set. You should see the text shown in Figure 5. If not, you did not setup profiling correctly and should restart XMD and try again.

⁴ For more information about the profile command, look on the online documentation.

light-sensitive solid-state device composed of many small cells. Each one of these cells is what we know as “pixels”. These pixels are arranged on a square matrix and the resolution of a CCD is simply the number of row pixels times the number of column pixels.

Now that we have the raw data, the next step is to compress and encode it. Figure 6 shows a block diagram of the Joint Photographic Experts Group (JPEG) compression/encoding scheme. The compression is not performed on the entire picture at one time. Instead, the picture is decomposed into blocks of 8x8 pixels. Each block is then run through the encode function, as you can see in Figure 6. Figure 6 represents the encoding which is done in the “encoder.cxx” file in the Lab 4 XPS project.

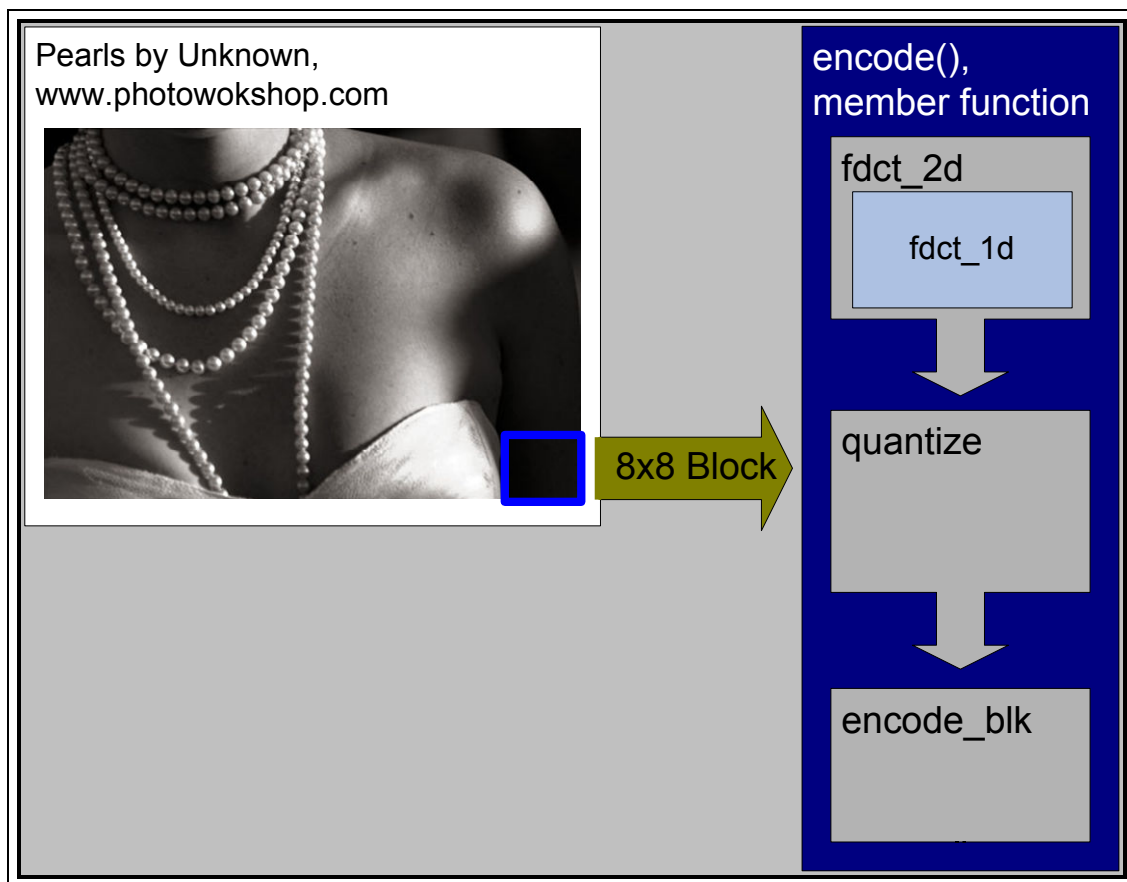


Figure 6. Block Diagram of Compression/Encoding Scheme.

The following is a discussion of how the JPEG encoding algorithm works.

The first step inside encode is to transform the 8x8 block into the frequency domain. This is done by running the block through the fdct_2d function, which performs a discrete cosine transform on the 8x8 pixels matrix. The fdct_2d function in turn calls the fdct_1d function on each row of the 8x8 matrix. The result of this step is an 8x8 matrix

of the original data, but in the frequency domain. This transformation results in an ordered matrix where the most basic picture information is in the upper-left corner of the matrix and the more detailed information from the picture is in the lower-right corner of the matrix. This step is completely lossless - all the original information is still there - we've just transformed it. We can take advantage of this ordering when compressing the data in the subsequent steps of quantizing and encoding.

The next step is to quantize the newly transform data. In a nutshell, quantizing is achieved by dividing the newly transformed matrix by some number, preferably by some factor of 2. Remember that diving by a factor of 2 can be done by simply shifting bits. Because of the way the DCT transformed the picture, this quantizing step can remove a lot of data that is difficult for humans to see, allowing the picture to be compressed to very small sizes. Thus, this is the step where the JPEG algorithm loses some fine details of the picture in order to allow high compression ratios.

The last step is to encode the quantized data; this step is where the formal compression takes place. We don't need to worry too much with the details of the encoding step. Suffice it to say that quantized data gets serialized by traversing the matrix, from the more frequent pixels to the least frequent ones. Then, applying Huffman encoding, the data gets encoded into binary codes. Huffman encoding assigns shorter codes to the more frequent pixels, while the least frequent pixels get longer codes⁶. It is from this encoded data that the final JPEG image is built.



Now that you have a basic understanding of the digital camera application, go through the profiling results again and answer the following questions.

- **Mention at least two functions which are good candidates for overall performance improvement. Justify your selections with the profile data you collected.**
- **Of these functions which one would be a good candidate to move into hardware? Remember what you did in Lab 3; think about the reasons why the simple multiplication was a bad choice.**

⁶ Huffman encoding is just one of the encoding schemes that are use on JPEG compression. You can find more about Huffman and other codes on: http://en.wikipedia.org/wiki/Huffman_coding