6.375 Project
JPEG decoder

Saajan S. Chana

Spring 2008

Contents

1 Introduction

The JPEG image standard is by far the most popular method for storing and exchanging photographic image files. The term ‘JPEG’ actually covers a wide range of related methods developed by the Joint Photographic Expert Group in the late 1980s. JPEG includes standards for both lossless and the more common lossy compression, as well as progressive rendering techniques for encoding images to be sent over slow transmission lines (such as the Internet).

JPEG is designed specifically for photographic or photorealistic images. As such it takes advantage of the tendency for such images to have most of their energy concentrated at low frequencies (i.e. they consist mainly of smooth gradients rather than sharp edges), so it produces better compression ratios on such images than other standards such as GIF. By contrast, images with lots of high-frequency features (such as text, or computer-generated images) are compressed comparatively poorly by JPEG, and furthermore must be encoded at high quality rates to avoid visible artifacts.

The core JPEG algorithms were designed with hardware implementation in mind. Therfore, significant efficiency and power gains could be realized by using a hardware accelerator for the computationally-intensive decoding process. This is particularly beneficial considering the plethora of battery-powered devices on the market which display JPEG images.

Part I
The JPEG standard

2 Variations

As mentioned above, JPEG is an extremely wide-ranging standard. See [3] for more details. It has provision for both 8-bit and 12-bit images, with 8-bit typically used for 24-bit colour photographs and 12-bit for greyscale medical images which require higher fidelity.

There are also several different possible orders for data encoding. The simplest is simply to encode pixel blocks left-to-right and top-to-bottom. If such an image is incrementally rendered, (for example if it is being rendered in a web browser while data is still being received over a slow uplink) it appears to be drawn row-by-row from the top to the bottom.

An alternative is to encode the image 'progressively', in several sweeps. The first sweep contains the lowest-frequency components, with successive sweeps containing progressively higher frequencies. Such an image can be rendered incrementally in a more satisfactory manner: after only a few bytes have been received, a first-pass render of the whole image can be achieved. This first pass appears blurred, with successive passes adding higher frequencies and more detail until the entire image has been received and rendered. This leads to a more satisfactory user experience than watching the image load line-by-line.

Various colourspaces are also supported. Images displayed on a computer screen must be converted to the RGB colourspace, where the colour of each pixel is specified as amounts of red, green and blue. However, this is not necessarily the best way to store an image. The human eye is much more sensitive to variations in brightness (luminance) than variations in colour (chrominance). Therefore, if a representation is used where these can be separated out, the chrominance components can be stored at a lower resolution, saving space. One such representation is YCbCr where Y is the luminance of each pixel, and Cb and Cr specify the 'redness' and 'blueness' respectively.

Finally, standards for both lossless and lossy compression are specified.

3 The Baseline Standard

Clearly a decoder which could deal with all the possible permutations would be extremely complex and time-consuming to design. Fortunately, the Standard specifies a subset which is sufficient for most applications. This is called the Baseline Standard, and is subject to the following limitations:

- Only Huffman coding, not the more complex arithmetic coding is used for compression[1]
- The decoder need only store two sets of Huffman coding tables
- Only the YCbCr colour space is supported
- Only 8 bits-per-pixel (per component) images are supported.
- Only sequential (row-by-row) encoding order is supported.
- Only lossy compression is used.

4 Method of operation

The encoding process is discussed here so as to make clear the purpose of each operation. To decode, the order of operations given here must be reversed. See [4] for a more in-depth discussion.

The fundamental unit of JPEG is the 8×8 block or tile. Unlike newer standards such as H.264, the tile sized is fixed regardless of the image content, simplifying implementation. If the image height and width are not multiples of 8, filler rows or columns are added by the encoder; this process is not discussed here.

The general principle of operation is that each block is subjected to a discrete cosine transform (DCT). This is an operation closely related to the DFT, with the difference that all the coefficients are real, so only half the number of real multiplications need be performed. The result of this is a matrix showing the signal power at each frequency. This representation is more amenable to compression.

The operations performed on each block are, in summary:

[1]See entropy encoding discussion, page 6

- DCT - see page 8.
- Quantization. The coefficient matrix is divided, element by element, by a quantization matrix. This serves two purposes: the DCT produces output coefficients of up to 12 bits – post-quantization these are reduced to 8 bits. Furthermore, higher frequencies are generally less important (and there is less power there anyway) – so they are divided by larger numbers so the post-quantization matrix contains smaller numbers (requiring less space) or, zeros for high-frequency components
- Reorder matrix elements. As many of the high frequency components are zero, it makes sense to keep them close together to provide longer zero-run- lengths which can be very compactly encoded. So the matrix is stored from low to high frequency as shown in figure 2.
- Entropy encoding - see page 6.

See figure 1 for a decoder block diagram.

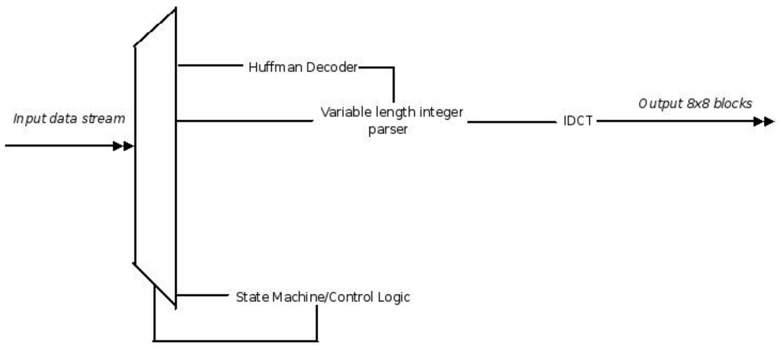

Figure 1: Decoder block diagram

5 Entropy Decoder

Each pixel value is preceded by an 8-bit token, with the first nybble containing the number of of preceding zeros and the second containing the number of bits required to encode the pixel value, which is stored as a variable-length signed integer (see below). A special token (0, 0) is used to signal end-of-block if there are no nonzero elements remaining.

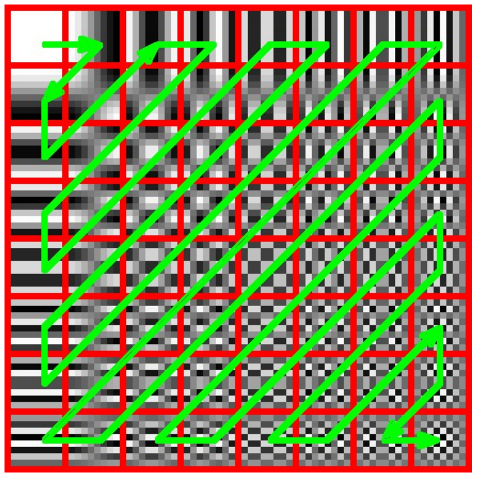

Figure 2: Coefficient Reordering (adapted from http://en.wikipedia.org/wiki/JPEG)

The tokens are compressed either using Huffman compression or arithmetic coding. Arithmetic coding has the advantages that no tables are required, and it automatically adapts to the statistics of the image, resulting in file sizes up to 5% smaller. However, arithmetic coding is rarely used for two reasons: firstly, it is more complicated to implement, both to encode and to decode; secondly and more importantly, it is (or may be) patent-encumbered.

The DC coefficients (the first element of each 8x8 block) are further compressed by making them relative to the last DC coefficient to be read in. This takes advantage of the fact that generally changes in average brightness are gradual in photographs, so the number of bits required to encode this delta is generally low. The DC coefficient of the first block in the image is stored relative to zero.

5.1 Variable-length integer (varint) decoding

If the number of bits is known, small integers can be encoded very compactly. For example, only one bit is required to encode -1 and $+1$; 2 bits can encode -3, -2, $+2$ and $+3$, and so on.

The algorithm for decoding a varint v of length n is as follows:

- If bit $n-1$ is set, return v
- Otherwise, return $v-(2^n-1)$

6 IDCT

Note that the output of the dequantizer is a 12-bit signed integer. The IDCT takes these as inputs and produces an 8-bit signed integer as an output. Adding 128 ($= 2^7$) to these produces the original 8-bit unsigned pixel values.

The 2-dimendional $n \times n$ IDCT is defined by the following formula:

$$f(x,y) = \frac{1}{4} \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} C(u)C(v)A(x,u)A(y,v)F(u,v)$$

where

$$\begin{aligned} n &= 8 \\ C(u) &= \begin{cases} 1\sqrt{2} & u = 0 \\ 0 & \text{otherwise} \end{cases} \\ A(x,u) &= \cos\frac{(2x+1)u\pi}{16} \end{aligned}$$

A direct implementation of this algorithm is obviously $O(n^4)$. Algorithms based on the FFT are available which run in $O(n^2 \log n^2)$ time.

7 Tile ordering

JPEG encodes each component of a colour image separately. This leads to the question of how to arrange tiles from different components in the bitstream. One solution is to encode all of the Y tiles, then all of the Cb tiles, then all of the Cr tiles. This is known as non-interleaved encoding. This is unsatisfactory because it makes incremental rendering impossible - you need to read two thirds of the file in before you can begin rendering.

The alternative is interleaved encoding, where the three components are placed alternately in the file. There is a slight subtlety here as each component may well use different Huffman tables and quantization matrices. By its nature, it is impossible to tell from the bitstream where the tile boundaries are except by actually decoding it, so the decoder itself must take care of switching tables as and when required.

This situation is complicated somewhat by the possibility that the horizontal and/or vertical resolutions of the components need not be the same. The chrominance components often have either their horizontal or vertical resolutions or both being half those of the luminance component. Obviously in this case the tiles can no longer be simply ordered Y – Cb – Cr.

To get around this problem, JPEG introduces the concept of a *Minimum Coded Block* (MCB). This is the smallest repeating unit. For example, an image containing Cb and Cr components with half the horizontal resolution of the Y component would have an MCB containing 5 tiles: two Y, one Cb and one Cr. The frame header and scan header (see page 9) specify the Huffman and quantization table to use for each tile in the MCB. There can be a maximum of 10 tiles in each MCB; in theory this limits the configurations available to encoders, but it does accomodate all configurations used in practice.

8 File format

There are many different container formats used to hold JPEG image data, as no single file format is specified in the standard [1]. The most common is known as JFIF (Jpeg File Interchange Format), but JPEG images can also be encapsualted (for example) in TIFF or EPS files. Many digital cameras and newer image editing packages use Exif (Extended Interchange Format) files, which are able to store additional metadate such as exposure time, rotation information and so on. However, Exif and JFIF are mutually compatible in that it is possible to construct a file that can be read by both Exif and JFIF readers.

JFIF files contain all the information needed to decode an image, including Huffman tables (if appropriate) and quantization tables. The various headers specifiy the size of the image and the exact method used to encode it, such as the number and resolution of components and the algorithm used for entropy coding.

See [2] for a full treatment of the JFIF format.

A JFIF file consists of a series of markers (introduced by the byte 0xFF, followed by a second byte specifying the marker type). Some markers specify parameters of the image (e.g. size), some indicate the presence of metadata, and some introduce blocks of binary data – such as Huffman tables or image data.

Unlike (for example) TIFF files, JFIF is designed to be read sequentially. So all tables (Huffman and quantization) must be defined before they can be used. The table definition specifies the slot in the decoder where they should be installed. It is valid to redefine an existing table; the new definition will be used for subsequent decoding.

Each image consists of exactly one *frame*[2]. The frame header specifies which quantization table to use for each component. Within the frame are one or more scans. For a sequential image, there is exactly one scan; for progressively encoded images there will be several scans, the first encoding only the low-frequency DCT coefficients and subsequent scans containing progressively more detail. The scan header specifies the Huffman tables to be used for the DC and AC components of each component.

Note that the scan header does not specify the length of the entropy-coded data to follow: the decoder is expected to read up to the next marker. As markers are indicated by the presence of an 0xFF bytes, any incidences of this byte in the entropy-coded stream must be protected. The encoder inserts a null (0x00) byte, which is not a valid marker ID, after each 0xFF byte. The decoder is expected to strip out and ignore these guard bytes.

[2]Other file formats such as MJPEG allow more than one frame per image

Part II
Implementation & Testing

9 Design scope and interface

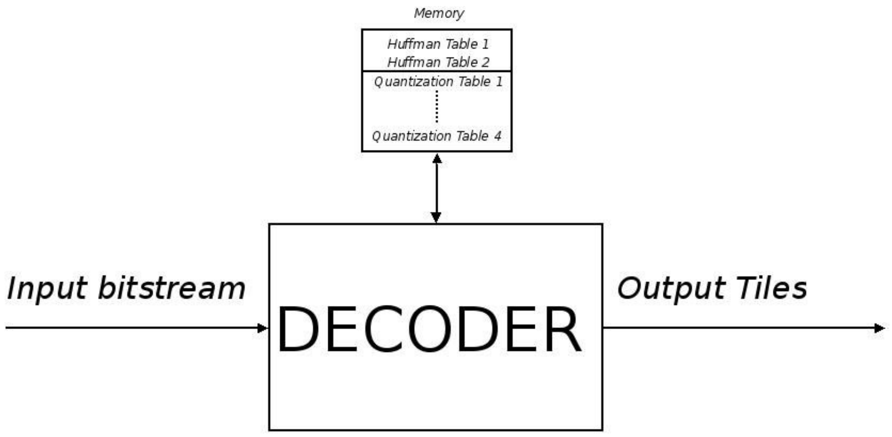

Figure 3: Implementation scope

See Figure 3. The datapath interfaces are via `Get` and `Put` interfaces. The input is raw words (the input bus width is paramaterizable) while the output is a 64-element `Vector` of 8-bit unsigned integers representing pixel intensities. It is expected that external logic or software will decode the actual file to extract the tables and set up the decoder before piping the raw image data in.

The module also exposes a `Client` interface to the internal memory used for storing Huffman and quantization tables. It is expected that these tables will be placed in memory by external logic or by software before the `JPEGDecoder` is used.

The `JPEGDecoder` module provides an `initialize` method in order to set up the table addresses. It takes a `UInt#(4)` specifying the number of components in each MCB, and three 10-element `Vector`s specifying the addresses of the DC Huffman tables, AC Huffman tables and quantization tables respectively for each component.

10 Test plan

A C++ program has been written which decodes the markers in a JFIF file and writes them out in human-readable format, as well as the quantization and Huffman tables and the raw image data. The output is formatted so that it can easily be pasted into a Verilog memory file for testing.

For testing, a testbench is used. This instantiates the decoder and streams in data loaded from a memory file (note that this is separate from the internal memory used to hold tables).

11 Implementation

11.1 General

Communication between the various decoder stages is acheived using the `GetPut` interface. At present each module implements the `Get::get()` and `Put::put()` methods to achieve this, but these could be replaced by standard FIFOs on the input to each module. The current method has the advantage of making the guards of each module explicit, so it is easy to see the conditions which lead to each one being ready.

11.2 Memory Arbiter

Both the dequantizer and the entropy decoder require memory access. This is mediated by an arbiter based on the one used for the SMIPS implementation in Lab 2. It is currently using a round-robin arbiter; it may be worthwhile prioritizing requests from the dequantizer in order to move data at the end of the pipeline as quickly as possible and to take advantage of data locality for fast memory transfers, if applicable.

As the dequantizer only requires 64 cycles of memory access this would be unlikely to detrimentally affect the entropy decoder.

11.3 Entropy Decoder

This module reconstitutes an 8×8 tile of DCT coefficients from the input bitstream. It consists of a state machine which alternately parses Huffman-coded tokens and variable length signed integers (varints), consuming one bit at a time. The compression scheme has been described previously[4].

11.3.1 Huffman decoding

Huffman codes are variable-length, with the most common input tokens being assigned shorter codes. A common hardware implementation of Huffman decoding uses a binary tree stored in memory with the left and right children of node n located at indices $2n$ and $2n + 1$ respectively. The value of each bit determines whether the decoder traverses the left or right branch; each memory address contains either the decoded token or an 'invalid' marker.

Unfortunately, while this is easy to implement, it does not fit well with the requirements for JPEG. Such a representation of the Huffman tree is not efficient for fixed-length tokens (as used in JPEG), as it would be very sparse; JPEG allows for codes of up to 16 bits in length, so 64KB of memory would be required to hold a maximum of 256 tokens (actually there are even fewer valid tokens than this). Considering that the only other memory requirement is for the four quantization tables, totalling 256 bytes, this is obviously impractical.

Instead, an approach more closely mirroring the way the Huffman tables are specified in the JPEG file format[2] is used. The codes are very succintly specified using a 16-byte array with each element specifying how many codes of that length are in use. The codes are assigned in ascending numerical order. For example, consider the following simple Huffman table:

Length	Number of Codes
1	1
2	0
3	2
4	2

etc. The first valid code (assigned to the most common token) is '0'. As there is only one token of length 1, a '1' in the bitstream indicates a longer code. There are no tokens of length 2, so the next valid code is '100' followed by '101', '1100', '1101' etc.

From this information, the controller can generate a table containing, for each length, the first invalid token (i.e. the first token which requires at least one more bit to be complete). For the above table, the generated table would look like:

Length	Num Codes	Max
1	1	1
2	0	10
3	2	110
4	2	1110

etc. To use this information, the decoder needs to keep track of all bits until it finds a valid token (i.e., less than the number recorded in the table entry for the appropriate length); at this point it computes an offset into a flat array of tokens.

In this implemenation, the 16 cutoffs and offsets are stored in registers inside the `EntropyDecoder` module. This avoids them being fetched from memory each time they are required; otherwise, at least one memory access would be required for each bit, and the final bit would require three more transfer. If the frequently-accessed cutoffs and offsets are stored locally, this is cut down to only a single transfer for the final bit in a token, and none otherwise.

The actual decompressed tokens are stored in memory, as there are up to 256 of them.

At the end of each tile, the `EntropyDecoder` module swaps in new tables as necessary.

11.3.2 Varint decoding

DC coefficient The first (top-left) element of each tile is the DC coefficient. This is specified as a difference from that of the last tile. The token specifies how many bits were required to encode the number. If the token is zero, it means an empty block, i.e. the DC coefficient is the same as the previous tile and all the other coefficients are zero

AC coefficients The high-order nybble of the token specifies the number of zero coefficients preceding this one. The low-order nybble specifies the length of the varint. A token of zero indicates end-of-block, i.e. all remaining coefficients are zero.

Decoding algorithm For an n-bit varint v:

- If $v[n-1] = 1$: return $v[n-2:0]$
- Else: return $v[n-2:0] - 2^n$

11.4 Dequantizer

This is implemented simply by fetching quantization coefficients from memory and multiplying each element of the decompresed tile by the corresponding element from the quantization matrix.

As with the entropy decoder, the dequantizer matrix takes care of swappping out quantization tables as required for each component.

11.5 IDCT

This is currently implemented naively, using an $O(n^2)$ algorithm. FFT-like $O(n \log n)$ algorithms also exist which would significantly improve the performance at the expense of an area increase. Such an improvement may well be profitable, as this is currently the rate-limiting step, taking 4096 cycles (but no memory accesses).

Pixel values are normalized to have zero mean before the forward DCT is run during the encoding process; so 128 is added to the (signed) output of the IDCT so that the final output is a 64-element vector of unsigned 8-bit integers, just like the original input.

12 Outcomes and Further Work

The code is currently close to working, but the entropy decoder is still buggy. Debugging is tricky due to its complexity. Synthesis has not yet been attempted.

Enhancing the efficiency of the IDCT would be useful, as this is currently the bottleneck in the pipeline. A related point is that Verilog multiplications are used in both the `IDCT` and `Dequantizer` modules, causing a single-cycle multiply to be inferred which incurs a substantial area penalty. Exploring different implementations for these multiplier could reduce the area, at the expense of slowing down the decode.

Looking ahead, potential improvements could include the ability to decode ‘progressive’ (rather than sequential) images; note that this requires the decoder to have access to a memory large enough to store the entire image in. This would allow it to read the vast majority of JPEG files in circulation.

Also, logic to decode the marker headers could be developed, allowing an entire JFIF file to be streamed in producing a bitmap on the output.

References

[1] Jpeg faq. http://www.faqs.org/faqs/jpeg-faq/.

[2] Eric Hamilton. Jpeg file interchange format v1.02. http://www.w3.org/Graphics/JPEG/jfif3.pdf, 1992.

[3] International Telecoms Union. Jpeg specification. http://www.w3.org/Graphics/JPEG/itu-t81.pdf, 1992.

[4] Gregory K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 1991.