A JPEG Decoder in SHIM

Nalini Vasudevan and Stephen A. Edwards
Columbia University, Department of Computer Science
nv2144@columbia.edu, sedwards@cs.columbia.edu

Abstract

Image compression plays an important role in multimedia systems, digital systems, handheld systems and various other devices. Efficient image processing techniques are needed to make images suitable for use in embedded systems.

This paper describes an implementation of a JPEG decoder in the SHIM programming language. SHIM is a software/hardware integration language whose aim is to provide communication between hardware and software while providing deterministic concurrency.

The paper shows that a JPEG decoder is a good application and reasonable test case for the SHIM language and illustrates the ease with which conventional sequential decoders can be modified to achieve concurrency.

1 Introduction

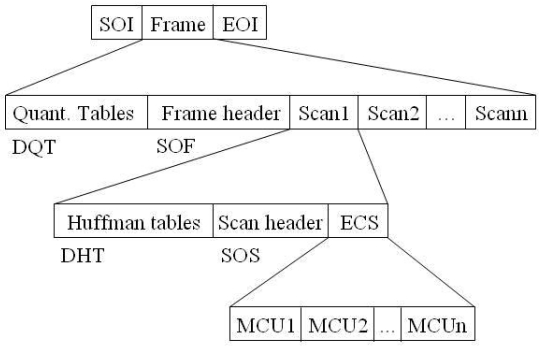
Image compression is vital for reducing storage requirements and minimizing bandwidth requirements. Conventional compression algorithms can be divided into a sequence of stages: at best they can run in a parallel pipeline. SHIM provides hardware/software integration augmented with deterministic concurrency, providing a good programming model for a JPEG decoder, allowing it to be described concurrently.

In this paper we discuss the implementation of a JPEG decoder in SHIM and how SHIM's features can be used to produce efficient, parallel code.

2 Related Work

Various streaming applications have been implemented in a variety of programming environments. Software has been written to target specific architectures, such as floating-point DSPs [6]. On the other hand, implementers also strive to support multiple underlying architectures. Different programming models are being employed. For example, Drake et al. [1] describe an MPEG decoder in the StreamIt programming language. StreamIt is a language used for programming large streaming applications and it facilitates variety of architecture. They show the StreamIt programming model is a good match for decoding an MPEG stream.

In this paper, we discuss a JPEG decoder implemented in the SHIM model [2], a concurrent, asynchronous, deterministic model for specifying, validating, and synthesizing such heterogeneous embedded systems. We show how the parallel nature of the model can be used to capture the concurrent parts of the decoder.



SOI	Start of image
DQT	Defines the Quantization Table
DHT	Defines the Huffman Table
SOF	Start of frame
SOS	Start of scan
EOF	End of file
MCU	Minimum Coded Unit (8×8 pixels)

Figure 1: Structure of a JPEG image (from J. V. Meerbergen, An introduction to the syntax of the JPEG bitstream, http://www.es.ele.tue.nl/~jef/education/)

3 Overview of the JPEG format

To reduce the storage and transmission requirements, the Joint Photographic Experts Group (JPEG) devised a method of compressing continuous-tone images. Figure 1 shows the structure of a JPEG bitstream.

3.1 JPEG Encoding

A JPEG image is composed of 8×8 pixel blocks known as Minimum Coded Units or MCUs. Each unit is converted to a frequency domain representation using the discrete cosine transform. The high frequency components are filtered using a low pass filter characterized by quantization tables, which determine the quality and the compression ratio of the image. Finally the data is coded using Huffman codes to allow more frequent values to be stored as shorter codes. Figure 2 illustrates this process.

3.2 JPEG Decoding

A JPEG decoder (Figure 3, the focus of our work) is the inverse process. The 8×8 units are retrieved from the encoded

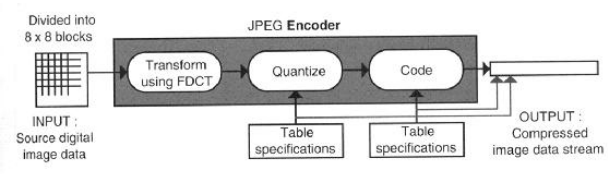


Figure 2: JPEG encoding [3]

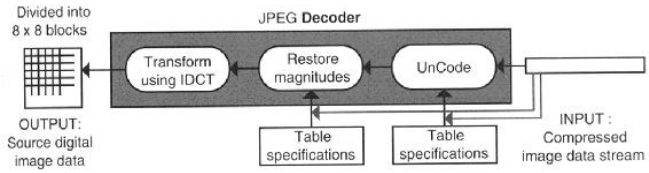


Figure 3: JPEG decoding [3]

data. The coded data is decoded using Huffman's algorithm with the data provided in the Huffman tables. The frequency components can be extracted using the quantization table. The result is a zig-zag (ZZ) vector which is reordered into an 8×8. Finally, the inverse discrete cosine transform (IDCT) is applied to the frequency domain to recover the 8×8 MCU blocks.

4 The SHIM Language

Generally, embedded systems are implemented using a combination of C and an HDL such as Verilog. Combining the two, because their semantics are dramatically different, is much more difficult than using each independently. SHIM [5] follows a C-like syntax that allows local variable declarations and function calls. It does not support pointers but it includes mechanism for concurrent procedure calls through the *par* statement and rendezvous-style interprocess communication through the *next* keyword. The current SHIM compiler generates C code; future compilers will also produce an HDL such as Verilog.

5 A JPEG Decoder in SHIM

5.1 Porting to SHIM

SHIM is a C-like language that supports local declarations and functions. In addition it has two important constructs—*next* and *par*. Records can be created using C-*struct* syntax. However C and SHIM have many differences. As such, the C code we used as a starting point (originally written by Koen van Eijk and Pierre Guerrier) had to be modified to work with the SHIM compiler. The aim was to read a stream of bytes from a JPEG file and produce a stream of bytes in raster format. We encountered the following issues while porting the C code to SHIM:

5.1.1 No global variables

SHIM does not allow variables to be defined globally. This results in better modularization and easy debugging, but means variables used frequently across the code have to be passed by reference to functions using them.

5.1.2 Static allocation

Unlike C, SHIM does not allow dynamic memory allocation and hence does not provide any construct like *malloc* or *new*. Arrays must be allocated statically. For example, the image buffer must be statically allocated; its size cannot depend on size of the image. This leads to wasted space for small images and the inability to handle large images.

5.1.3 No Pointers

SHIM does not support pointers. Tree-like structures cannot be represented with pointers, but an array implementation can be used.

5.1.4 Pass-by-reference

By default, all variables (including arrays and structures) are passed by value. SHIM also allows values to be passed to functions by reference. The original C code used pointers for such arguments.

5.1.5 Integrity of data structures

SHIM currently does not allow an individual element of an array or a struct to be passed by reference. The entire array or struct has to be passed by reference.

5.1.6 No macro substitutions

SHIM does not have a C-like macro preprocessor, so all macros in the C code were either rewritten as functions or directly substituted in the code.

5.1.7 Input/Output

The current SHIM compiler does not have I/O libraries, but it does allow arbitrary C to be included between pairs of dollar signs. Thus, I/O can be performed by wrapper functions such as the following, which reads a single byte from the standard input stream.

```
void input(int8 &val) {
  $$VAL(0) = (int) getchar();$$
}
```

5.1.8 Compiler bugs

During the process of implementing the JPEG decoder, a considerable number of minor compiler bugs were discovered and fixed. Thus, the JPEG decoder proved to be a good test application for the SHIM compiler.

5.2 Prototypes

Table 1 lists variables that are used frequently across the code.

Table 1: Commonly-used variables

MCU_buff[10]	Buffer containing decoded DCT blocks
QTable[4] and QValid[4]	Quantization table information
x_size and y_size	Image frame size
MCU_sx and MCU_sy	MCU size in pixels
n_comp	Number of components: 1 or 3
comp[3]	Descriptors for 3 components
ColorBuffer	MCU Buffer after color conversion
FrameBuffer	Complete final RGB image
FBuff	Scratch frequency buffer
PBuff	Scratch pixel buffer
cur_comp	Component under consideration.
in_frame, MCU_row, and MCU_column	Position within an MCU
stuffers	Stuffed bytes in data segment
passed	Bytes passed when searching for markers
bit_count	# of bits available in current window

Structure of the elements of comp[3]:

```
struct cd_t {
  int8  CID;    /* component ID */
  int8  IDX;    /* index of first block in MCU */

  int8  HS;     /* sampling factors */
  int8  VS;
  int8  HDIV;   /* sample width ratios */
  int8  VDIV;

  int8  QT;     /* QTable index, 2 bits  */
  int8  DC_HT;  /* DC table index, 1 bit */
  int8  AC_HT;  /* AC table index, 1 bit */
  int8  PRED;   /* DC predictor value */
}
```

5.2.1 *Functions*

Table 1 lists variables that occur frequently throughout the code.

Input Reads the input stream

```
int read_input(int32 &c)
```

JPEG Decoder The main function, which executes in parallel with the input module.

```
void jpeg_decoder()
```

Huffman table loader The variable c is shared between the decoder and input reader.

```
int
load_huff_tables(int c, int &HTable[4][384])
```

Quantization table loader

```
load_quant_tables(int32 c, PBlock &QTable[4],
                  int32 &QTvalid[4])
```

MCU processor Processes a block

```
int
process_MCU(int c, int &MCU_column, int &MCU_row,
  int &mx_size, int &my_size, int &in_frame,
  int &MCU_valid[10], int &curcomp, FBlock & FBuff,
  PBlock & MCU_buff[10], cd_t &comp[3],
  PBlock &QTable[4], int & window,   int &bit_count,
  int &stuffers, int &HTable[4][384], int &n_comp,
  int8 &ColorBuffer[60000], int &MCU_sx, int &MCU_sy,
  int& x_size, int& y_size, int& rx_size,
  int& ry_size, int8 & FrameBuffer[60000])
```

MCU unpacker Unpacks MCU, dequantifies and reorders

```
FBlock
unpack_block(int &c, FBlock & T, int &select,
   cd_t &comp[3], PBlock QTable[4],
   int &window, int &bit_count, int &stuffers,
   int& HTable[4][384],int &n_comp)
```

1-D IDCT Performs inverse 1-D discrete cosine transform of Y using original Loeffler's algorithm. Result Y is also scaled up by factor sqrt(8).

```
void idct_1d(int &Y[64], int s)
```

2-D IDCT Inverse 2-D discrete cosine transform (done one block at a time). Calls idct_1d()

```
void IDCT(FBlock  input, PBlock & output);
```

Parallel IDCT Runs n IDCTs in parallel.

```
int
parallelize_IDCT(FBlock input[10],
                 PBlock&  MCU_Buff[10], int n);
```

Colorspace Conversion Y, Rcr, Rcb to RGB conversion

```
void
color_conversion(int8 & ColorBuffer[60000],
     PBlock  MCU_buff[10], cd_t comp[3],
     int MCU_sx, int MCU_sy);
```

Output Reads the values from Framebuffer and puts it out in raster format.

```
void
RGB_save(int8 FrameBuffer[60000],
         int x_size, int y_size, int n_comp);
```

5.2.2 *Order of execution*

The main function is jpeg_decoder(). On reading a DHT_MK marker, it calls load_huff_tables() to load the Huffman tables. When it reads DQT_MK, it loads the quantization tables. On receiving the SOS marker, it calls process_MCU, which in turn calls unpack_block, parallelize_IDCT and color_conversion. Finally, the output module dumps the image in raster format.

5.3 *Modules*

5.3.1 *Huffman Decoding*

Huffman coding is a lossless compression algorithm. The symbols are sorted by weight, the two least weighted subtrees are paired, and this process is repeated until a complete tree is built. Figure **??** illustrates this.

The Huffman decoding algorithm is as follows.

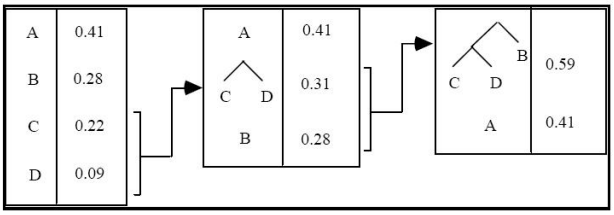


Figure 4: Huffman Coding (after Yusof et al. [6])

1

1. A tree equal to the encoding tree is generated. Since SHIM does not support pointers, we represent this tree using an array.
2. We read an input bit and move to the left subtree if the bit is 0, the right subtree otherwise.
3. If a leaf is encountered, the value is returned. Intermediate nodes in the tree have a value of 256 to distinguish them from leaves (symbols).
4. The last two steps are repeated until the input is empty.

Part of the code for loading the Huffman table is shown below.

```
for (i = 0; i < 16; i++) {
  /* hclass stands for AC or DC */
  LeavesN = HTable[id][MAX_SIZE (hclass) - 2 * i - 1]
          = next c;
  CellsN = 2 * NodesN;  /* nodes are old */
  NodesN = HTable[id][MAX_SIZE (hclass) - 2 * i - 2]
         =  CellsN - LeavesN;
  if (LeavesN) MaxDepth = i;
}
```

5.3.2 *Dequantization*

The quantization table is an array whose entries correspond to the 64 spatial frequencies in the 8×8 MCU block. The frequencies in the MCU block are divided by the values specified in the quantization table.

The quantization table appears in the JPEG stream in the following format:

Number of Bits	Value
u8	0xff
u8	0xde (type of segment)
u16	Length of segment
u4	Precision in bits
u4	Table id
64 elements of u8 or u16	Quant. values in zigzag order

The reverse, i.e., scaling, is done during the decoding process. The SHIM snippet for loading quantization tables is below.

```
QTvalid[id] = 1;
for (x = 0; x < 64; x++)
   QTable[id].linear[x] = next c;
```

5.3.3 *Zig Zag Reordering*

Zig-zag ordering (Figure 5) traverses the 8×8 DCT coefficients in order of increasing spatial frequencies, which helps to bring together similar values that compress better.

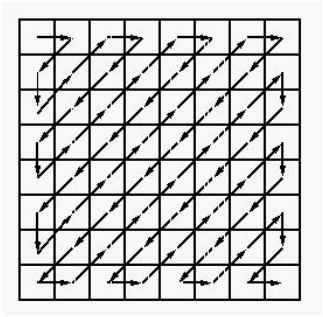


Figure 5: Zig-Zag traversal

Consequently, the vector is sorted on spatial frequency. The first element is the value corresponding to the lowest frequency and is known as DC term. The other components are known as AC terms. The zig-zag vector has to be reordered while decoding and this is done during the reordering phase.

```
/* Dequantify and ZigZag-reorder: */
 T.linear[G_ZZ[i]] = value *
                     QTable[comp[select].QT].linear[i];
/* G_ZZ is the reordering vector */
```

5.3.4 *IDCT: Loeffler's Algorithm*

We use Loeffler's algorithm [4] to compute the IDCT (this was also used in the C code we started from). The implementation in SHIM is below.

```
/* Original Loeffler algorithm.
   Inverse 1-D Discrete Cosine Transform.
   Y is scaled up by factor sqrt(8).
   s is the index to an element in the array. */
void idct_1d(int &Y[64], int s)
{
  int z1[8], z2[8], z3[8], x, y;

  /* Stage 1: */
  but(Y[s + 0], Y[s + 4], x, y);
  z1[1] = x, z1[0] = y;
  z1[2] = CMUL( 8867, Y[s+2]) - CMUL(21407, Y[s+6]);
  z1[3] = CMUL(21407, Y[s+2]) + CMUL( 8867, Y[s+6]);
  but(Y[s+1], Y[s+7], x, y);
  z1[4] = x, z1[7] = y;
  z1[5] = CMUL(23170, Y[s + 3]);
  z1[6] = CMUL(23170, Y[s + 5]);

  /* Stage 2: */
  but(z1[0], z1[3], x, y);
  z2[3] = x, z2[0] = y;
  but(z1[1], z1[2], x, y);
  z2[2] = x, z2[1] = y;
  but(z1[4], z1[6], x, y);
  z2[6] = x, z2[4] = y;
  but(z1[7], z1[5], x, y);
  z2[5] = x, z2[7] = y;

  /* Stage 3: */
  z3[0] = z2[0];
  z3[1] = z2[1];
  z3[2] = z2[2];
  z3[3] = z2[3];
```

```
  z3[4] = CMUL(13623, z2[4]) - CMUL( 9102, z2[7]);
  z3[7] = CMUL( 9102, z2[4]) + CMUL(13623, z2[7]);
  z3[5] = CMUL(16069, z2[5]) - CMUL( 3196, z2[6]);
  z3[6] = CMUL( 3196, z2[5]) + CMUL(16069, z2[6]);

  /* Final stage 4: */
  but(z3[0], z3[7], x, y);
  Y[s + 7] = x, Y[s + 0] = y;
  but(z3[1], z3[6], x, y);
  Y[s + 6] = x, Y[s + 1] = y;
  but(z3[2], z3[5], x, y);
  Y[s + 5] = x, Y[s + 2] = y;
  but(z3[3], z3[4], x, y);
  Y[s + 4] = x, Y[s + 3] = y;
}

int CMUL(int C, int x)
{
   int C_BITS = 14;
   return (C * x + (1 << (C_BITS - 1)))
          >> C_BITS;
}

int but(int a, int b, int &x, int &y)
{
   x =  a - b;
   y =  a + b;
}
```

5.3.5 *Level shifting*

While encoding, the samples are shifted down by 128 to make them signed. While decoding, 128 is added to the all 8-bit signed values in the 8×8 blocks resulting from the IDCT transform.

5.3.6 *[Y,Cb,Cr] to [R,G,B] Conversion*

JPEG stores the image details in [Y,Cb,Cr] format where Y is a weighted-filter with different weights for each spectral component. Cb and Cr are the chrominance values and represent two coordinates in a system which measures the hue and saturation of the color. The snippet for conversion in SHIM is as follows:

```
r = y + ((359 * rcr) >> 8);
g = y - (( 11 * rcb) >> 5) - ((183 * rcr) >> 8);
b = y + ((227 * rcb) >> 7);
```

5.4 *Determining the concurrent units*

Since SHIM is a deterministic concurrent language, the goal is to maximize parallelism in the application. The first step is to identify expensive computations which can be done in parallel. One such computation is the IDCT. The blocks can be processed independently to get the IDCT of each block.

5.4.1 *The Input Stream*

The input to the decoder is a stream of bytes. The input read unit and the processing unit run in parallel, as shown below and in Figure 6.

```
read(c); par decode(c);
```

The input unit reads one byte at a time and passes it to the processing unit. The communication is achieved using the *next c* command.

```
void input_read(int8 & c)
{
    next c = get_next_byte(); /*Sender*/
```

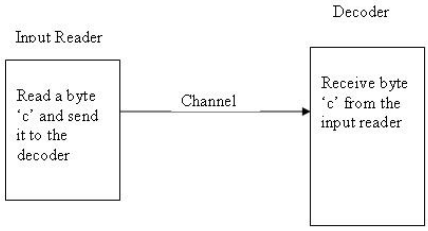


Figure 6: Input Reader and Decoder as two parallel units

```
}

void decode(int8 c)
{
    ..
    temp = next c; /*Receiver*/
    process( temp);
    ..
    ..
}
```

The read function and the decode function are parallel modules but the output module is not parallel with the decode module. The reason is quite simple: the last stage of the decoder sends processed data of dimensions 8×8, rather than a sequential stream. Hence, the block outputs are collected and then converted to a sequential stream. This cannot happen in parallel with the decoder because the output module has to wait for all the data to arrive from the previous stage.

5.4.2 *Hard coding the number of parallel units*

The number of parallel units can be hard-coded to a value k. If there are n blocks and they can be processed in parallel, then each thread becomes the owner of approximately n/k blocks. The computational speed hence expected is k times the speed of a single threaded system. Also, note that the n/k blocks of each unit are processed sequentially. At any given time there can be a maximum of k units running in parallel.

Suppose $k = 3$. We have

```
  idct(input[0], out);
par
  idct(input[1], out);
par
  idct(input[2], out);
```

5.4.3 *Achieving dynamic behavior*

Determining the number of parallel units can be done during runtime using recursion. Given n blocks, n threads can be created on the fly in the following way.

```
void IDCT_parallelize(Block input[10],
                      &output[10], int n)
{
  if (n == -1) return;

  Block out;

    idct(input[n-1], out);
```

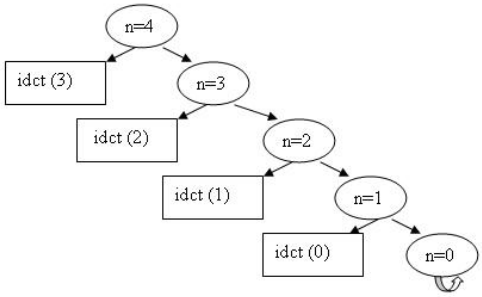


Figure 7: Spawning *n* parallel units using recursion

Table 2: Comparison of Execution times

JPEG Image Size	C	SHIM
0.663KB	0.003s	0.011s
1.883KB	0.003s	0.124s
1.933KB	0.003s	0.270s
3.636KB	0.004s	0.365s
3.736KB	0.004s	0.372s
5.619KB	0.004s	0.488s

```
  par
    IDCT_parallelize(input, output, n-1);

  output[n-1] = out;
}
```

The parallel units are spawned as shown in Figure 7.

If the IDCT computation is expensive enough compared to the expense of creating a new activation record, idct(3), idct(2), idct(1) and idct(0) can be considered to be in parallel.

6 Results

The original JPEG decoder code consisted of 1618 lines across 9 files. Our SHIM version consists of 1092 lines in a single file. The similar size is to be expected as the SHIM syntax is deliberately C-like.

Table 2 lists the execution time of our JPEG decoder on various images running code from the current SHIM compiler and the original C code. The images are fairly small because of the artificial buffer size limit our implementation imposes.

The code from the SHIM compiler has substantially worse execution times compared to its C counterpart. This is disappointing, but mostly illustrates that the syntax-directed translation of the current SHIM compiler is not very efficient. As this is a reasonable application of SHIM, we are happy to have the JPEG decoder example as a benchmark for improving the performance of the SHIM compiler.

7 Conclusions

In this paper, we described the implementation of a JPEG decoder in the SHIM language. We also recognized the possible concurrent stages and took advantage of SHIM"s concurrency feature to parallelize the algorithm, mostly by running multiple copies of the IDCT transform in parallel. This JPEG decoder will be a good benchmark for testing the quality and correctness of the compiler.

Even though the performance results are disappointing, we have shown it is realistic to code a JPEG decoder as a SHIM program and that the SHIM language and compiler is mature enough to handle a reasonably-sized application such as this.

References

[1] Matthew Drake, Henry Hoffman, Rodric Rabbah, and Saman Amarasinghe. MPEG-2 decoding in a stream programming language. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, apr 2006.

[2] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[3] John P. Jones. JPEG decoder design. Technical Report Sr. Design Document EE175WS00-11, Electrical Engineering Dept., University of California, Riverside, 2000.

[4] Christoph Loeffler, Adriaan Ligtenberg, and George S. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *Proceedings of the IEEE International Conference on Acoustics, Speech, & Signal Processing (ICASSP)*, volume 2, pages 988–991, Glasgow, Scotland, May 1989.

[5] Olivier Tardieu and Stephen A. Edwards. R-SHIM: Deterministic concurrency with recursion and shared variables. In *Proceedings of the 4th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, page 202, Napa, California, July 2006.

[6] Zulkalnain Mohd Yusof, Zulkifli Mohamad, Zulfakar Aspar, Ishak Suleiman, Mohd. Helmi Mohd. Saad, and Sheikh Hussain Sheikh Salleh. Real time implementation of baseline JPEG decoder using floating point DSP TMS320C31. In *Proceedings of TENCON*, volume 3, pages 383–386, Kuala Lumpur, Malaysia, September 2000.

Code listing

```
/*************************************************************
*
* JPEG Decoder in SHIM
* Nalini Vasudevan and Stephen A. Edwards
* From an original by by Pierre Guerrier and Koen van Eijk.
*
*************************************************************/

/* Quick Reference
-------------------
* comp[3]- Descriptors for 3 components
* MCU_buff[10]- Decoded DCT blocks buffer
* MCU_valid[10]- Components for the above buffer
* QTable[4] and QValid[4]- Quantization tables
* x-size and y-xize- Video frame size
* rx_size and ry_size- Downrounded video frame size
* MCU_sx and MCU_sy- MCU size in pixel
* n_comp- no. of components 1 or 3
* ColorBuffer- MCU after Color Conversion
* FrameBuffer- Complete Final RGB image
* FBuff- Scratch frequency buffer
* PBuff- Scratch pixel buffer
* in_frame- frame started?
* cur_comp- current component?
* MCU_row, MCU_column- Current position in MCU unit
* stuffers- stuffed bytes in code segment
* passed- bytes passed when searching markers
*/

/* C part - required for debugging (printf statements) */
$$
#include<stdio.h>
int verbose = 0;
$$

/* 1s complement of 1 is FFFE, 2s complement of 1 is FFFF */

int reformat(uint64 S, int good)
{
  int St;
  if (!good) return 0;
  St = 1 << (good - 1);         /* 2^(good-1) */
  if (S < St) return (S + 1 + ((-1) << good));
  else return S;
}

/* Get the huffman flag - mask it with HUFF_FLAG_MASK */
int HUFF_FLAG(int c)
{
  return c & 0x300;
}

/*The size is either 384 or 64 depending on hclass*/
int MAX_SIZE(int hclass)
{
  int temp =((hclass) ? 384 : 64);
  return temp;
}

/*Scale by 2^n*/
int SCALE(int x, int n)
{
  return x << n;
}

int DESCALE(int x, int n)
{
  return (x + (1 << (n - 1)) - (x < 0)) >> n;
}

int CMUL(int C, int x)
{
  int C_BITS = 14;
  return ((C) * (x) + (1 << (C_BITS - 1))) >> C_BITS;
}

/* Butterfly: but(a,b,x,y) = rot(sqrt(2),4,a,b,x,y)
   Add and subtract, store add result in x
   and subtract result in y */
int
but (int a, int b, int &x, int &y)
{
  x = a - b;
  y = a + b;
}

/*Convert the value to bigendian by rotating the bytes*/
uint32 MACHINE_2_BIGENDIAN(uint64 value)
{
  return
    ((value & 0x000000FF) << 24) |
    ((value & 0x0000FF00) <<  8) |
    ((value & 0x00FF0000) >>  8) |
    ((value & 0xFF000000) >> 24);
}

/* Write header information to stdout */
void write_out(uint32 value, int size, int n )
{
 $$fwrite(&VAL(0), VAL(1), VAL(2), stdout);$$
}

/* Write out to stdout from framebuffer */
void write_out_F(int8 value )
{
  $$putchar(VAL(0));$$
}

/* Reads the value from Framebuffer and generates a raster*/
void RGB_save(int8 FrameBuffer[60000],
              int x_size, int y_size, int n_comp)
{
  int i;
  uint32 bigendian_value;

  uint32 FrameHeader_MAGIC      = 0x59a66a95;
                                  /* round to 2 more */
  uint32 FrameHeader_Width      = 2 * ceil_div(x_size, 2);
  uint32 FrameHeader_Heigth     = y_size;
  uint32 FrameHeader_Depth      = (n_comp>1) ? 24 : 8;
  uint32 FrameHeader_Length    = 0; /* unnecessary in v1.0 */
  uint32 FrameHeader_Type      = 0; /* old one */
  uint32 FrameHeader_CMapType  = 0; /* truecolor */
  uint32 FrameHeader_CMapLength= 0; /* none */

  /* Frameheader must be in Big-Endian format */

  bigendian_value = MACHINE_2_BIGENDIAN(FrameHeader_MAGIC);
  write_out(bigendian_value, 4, 1);

  bigendian_value = MACHINE_2_BIGENDIAN(FrameHeader_Width);
  write_out(bigendian_value, 4, 1);

  bigendian_value = MACHINE_2_BIGENDIAN(FrameHeader_Heigth);
  write_out(bigendian_value, 4, 1);

  bigendian_value = MACHINE_2_BIGENDIAN(FrameHeader_Depth);
  write_out(bigendian_value, 4, 1);

  bigendian_value = MACHINE_2_BIGENDIAN(FrameHeader_Length);
  write_out(bigendian_value, 4, 1);

  bigendian_value =MACHINE_2_BIGENDIAN(FrameHeader_Type);
  write_out(bigendian_value, 4, 1);

  bigendian_value =
    MACHINE_2_BIGENDIAN(FrameHeader_CMapType);
  write_out(bigendian_value, 4, 1);

  bigendian_value =
    MACHINE_2_BIGENDIAN(FrameHeader_CMapLength);
  write_out(bigendian_value, 4, 1);

  /* Note that n_comp contains the no. of component colors:
      n_comp is 1 if black and white pic else n_comp is 3 */
  /* FrameHeader_Width is a function of x_size */
  /* Reminder: xsize and ysize hold the size of each Frame */

  for (i=0; i<y_size; i++) {
    int index = n_comp*i*x_size;
    for(int j=0;j<n_comp*FrameHeader_Width;j++) {
      write_out_F(FrameBuffer[index]);
      index++;
    }
```

```
  }
}

/* Inverse 1-D Discrete Cosine Transform.
   Result Y is scaled up by factor sqrt(8).
   Original Loeffler algorithm. */
void idct_1d(int &Y[64], int s)
{
  int z1[8], z2[8], z3[8], x, y;

  /* Stage 1: */
  but (Y[s + 0], Y[s + 4], x, y);
  z1[1] = x, z1[0] = y;
  /* rot(sqrt(2), 6, Y[s+2], Y[s+6], &z1[2], &z1[3]); */
  z1[2] = CMUL( 8867, Y[s + 2]) - CMUL(21407, Y[s + 6]);
  z1[3] = CMUL(21407, Y[s + 2]) + CMUL( 8867, Y[s + 6]);
  but (Y[s+1], Y[s+7], x, y);
  z1[4] = x, z1[7] = y;
  /* z1[5] = CMUL(sqrt(2), Y[s+3]);
     z1[6] = CMUL(sqrt(2), Y[s+5]); */
  z1[5] = CMUL (23170, Y[s + 3]);
  z1[6] = CMUL (23170, Y[s + 5]);

  /* Stage 2: */
  but (z1[0], z1[3], x, y);
  z2[3] = x, z2[0] = y;
  but (z1[1], z1[2], x, y);
  z2[2] = x, z2[1] = y;
  but (z1[4], z1[6], x, y);
  z2[6] = x, z2[4] = y;
  but (z1[7], z1[5], x, y);
  z2[5] = x, z2[7] = y;

  /* Stage 3: */
  z3[0] = z2[0];
  z3[1] = z2[1];
  z3[2] = z2[2];
  z3[3] = z2[3];
  /* rot(1, 3, z2[4], z2[7], &z3[4], &z3[7]); */
  z3[4] = CMUL(13623, z2[4]) - CMUL( 9102, z2[7]);
  z3[7] = CMUL( 9102, z2[4]) + CMUL(13623, z2[7]);
  /* rot(1, 1, z2[5], z2[6], &z3[5], &z3[6]); */
  z3[5] = CMUL(16069, z2[5]) - CMUL( 3196, z2[6]);
  z3[6] = CMUL( 3196, z2[5]) + CMUL(16069, z2[6]);

  /* Final stage 4: */
  but (z3[0], z3[7], x, y);
  Y[s + 7] = x, Y[s + 0] = y;
  but (z3[1], z3[6], x, y);
  Y[s + 6] = x, Y[s + 1] = y;
  but (z3[2], z3[5], x, y);
  Y[s + 5] = x, Y[s + 2] = y;
  but (z3[3], z3[4], x, y);
  Y[s + 4] = x, Y[s + 3] = y;
}

/* Inverse 2-D Discrete Cosine Transform */
void IDCT(FBlock input, PBlock &output)
{
  int S_BITS = 3;
  int Y[64];
  int k, l;

  /* Pass 1: process rows. */
  for (k = 0; k < 8; k++) {
    /* Prescale k-th row: */
    for (l = 0; l < 8; l++)
      Y[8 * k + l] = SCALE(input.linear[k*8+l], S_BITS);
    /* 1-D IDCT on k-th row: */
    idct_1d (Y, 8 * k);
    /* Result Y is scaled up by factor sqrt(8)*2^S_BITS. */
  }

  /* Pass 2: process columns. */
  for (l = 0; l < 8; l++) {
    int Yc[64];

    for (k = 0; k < 8; k++)
      Yc[k] = Y[8 * k + l];
    /* 1-D IDCT on l-th column: */
    idct_1d (Yc, 0);
    /* Result is once more scaled up by a factor sqrt(8). */
    for (k = 0; k < 8; k++) {
      /* includes level shift */
      int r = 128 + DESCALE (Yc[k], S_BITS + 3);

      /* Clip to 8 bits unsigned: */
      r = r > 0 ? (r < 255 ? r : 255) : 0;
      output.linear[k*8+ l] = r;

    }
  }
}

int parallelize_IDCT(FBlock input[10],
                     PBlock& MCU_Buff[10], int n)
{
  PBlock temp1;

  if (n==-1) return 1;

  {
    temp1 = MCU_Buff[n];
    IDCT(input[n], temp1);
  } par {
    parallelize_IDCT(input, MCU_Buff, n-1);
  }
  MCU_Buff[n]= temp1;
}

/* Ensure number is >=0 and <=255  */
int Saturate(int n)
{
  return n > 0 ? (n < 255 ? n : 255) : 0;
}

/*-----------------------------------------*/
/* rules for color conversion:             */
/*  r = y              +1.402  v           */
/*  g = y -0.34414u     -0.71414v          */
/*  b = y +1.772  u                        */
/* Approximations: 1.402 # 7/5     = 1.400  */
/*                .71414 # 357/500 = 0.714  */
/*                .34414 # 43/125  = 0.344  */
/*               1.772 = 443/250           */
/*-----------------------------------------*/
/* Approximations: 1.402 # 359/256 = 1.40234 */
/*                .71414 # 183/256 = 0.71484 */
/*                .34414 # 11/32   = 0.34375 */
/*                 1.772 # 227/128 = 1.7734  */
/*-----------------------------------------*/

/* Color Conversion is happening at once */
void
color_conversion(int8 & ColorBuffer[60000],
   PBlock & MCU_buff[10], cd_t& comp[3],
   int& MCU_sx, int& MCU_sy)
{
  int i, j;
  uint8 y, cb, cr;
  int8 rcb, rcr;
  int r, g, b;
  int offset;

  for (i = 0; i < MCU_sy; i++)  { /* pixel rows */
    int ip_0 = i >> comp[0].VDIV;
    int ip_1 = i >> comp[1].VDIV;
    int ip_2 = i >> comp[2].VDIV;
    int inv_ndx_0 = comp[0].IDX + comp[0].HS * (ip_0 >> 3);
    int inv_ndx_1 = comp[1].IDX + comp[1].HS * (ip_1 >> 3);
    int inv_ndx_2 = comp[2].IDX + comp[2].HS * (ip_2 >> 3);
    int ip_0_lsbs = ip_0 & 7;
    int ip_1_lsbs = ip_1 & 7;
    int ip_2_lsbs = ip_2 & 7;
    int i_times_MCU_sx = i * MCU_sx;

    for (j = 0; j < MCU_sx; j++) {      /* pixel columns */
      int jp_0 = j >> comp[0].HDIV;
      int jp_1 = j >> comp[1].HDIV;
      int jp_2 = j >> comp[2].HDIV;

      y = MCU_buff[inv_ndx_0 + (jp_0 >> 3)].
            linear[(ip_0_lsbs)*8+(jp_0 & 7)];
      cb = MCU_buff[inv_ndx_1 + (jp_1 >> 3)].
            linear[(ip_1_lsbs)*8+(jp_1 & 7)];
      cr = MCU_buff[inv_ndx_2 + (jp_2 >> 3)].
```

```
            linear[(ip_2_lsbs)*8+(jp_2 & 7)];

      rcb = cb - 128;
      rcr = cr - 128;

      r = y + ((359 * rcr) >> 8);
      g = y - ((11 * rcb) >> 5) - ((183 * rcr) >> 8);
      b = y + ((227 * rcb) >> 7);
      offset = 3 * (i_times_MCU_sx + j);
      /* note that this is SunRaster color ordering */
      ColorBuffer[offset + 2] = Saturate(r);
      ColorBuffer[offset + 1] = Saturate(g);
      ColorBuffer[offset + 0] = Saturate(b);
    }
  }
}

void reset_prediction (cd_t &comp[3])
{
  for (int i = 0; i < 3; i++) comp[i].PRED = 0;
}

int HUFF_VALUE(int c)
{
  return c & ~0x300;
}

uint8 get_symbol(int &c, int select,
                 int & HTable[4][384], int &bit_count,
                 int &window, int &stuffers)
{
  int cellPt;

  cellPt = 0; /* this is the root cell */
  while (HUFF_FLAG(HTable[select][cellPt]) == 0x100)
    cellPt  =
      get_one_bit(c, bit_count, stuffers, window) |
      (HUFF_VALUE(HTable[select][cellPt])<<1);

  switch (HUFF_FLAG(HTable[select][cellPt])) {
  case 0x000:
    return 0;
    break;

  case 0x200:
    return HUFF_VALUE(HTable[select][cellPt]);
    break;

  case 0x300:
    return 0;
    break;

  default:
    break;
  }
  return 0;
}

void clear_bits (int &bit_count)
{
  bit_count = 0;
}

uint8 get_one_bit(int &c, int &bit_count,
                  int &stuffers, int &window)
{
  int newbit;
  uint8 aux, wwindow;

  if (bit_count == 0) {
    wwindow = next c;

    if (wwindow == 0xFF)
      switch (aux = next c) { /* skip stuffer 0 byte */
      case 0xFF:
        break;

      case 0x00:
        stuffers++;
        break;

      default:
        break;
      }
    bit_count = 8;
  } else wwindow = window;

  newbit = (wwindow >> 7) & 1;
  window = wwindow << 1;
  bit_count--;
  return newbit;
}

uint64 get_bits(int &c, int number,
                int &bit_count, int &window, int &stuffers)
{
  int i, newbit;
  uint64 result = 0;
  uint8 aux, wwindow;

  if (!number) return 0;

  for (i = 0; i < number; i++) {
    if (bit_count == 0) {
      wwindow = next c;

      if (wwindow == 0xFF)
        switch (aux = next c) { /* skip stuffer 0 byte */
        case 0xFF:
          break;

        case 0x00:
          stuffers++;
          break;

        default:
          break;
        }
      bit_count = 8;
    }
    else wwindow = window;
    newbit = (wwindow >> 7) & 1;
    window = wwindow << 1;
    bit_count--;
    result = (result << 1) | newbit;
  }
  return result;
}

FBlock unpack_block(int &c, FBlock & T,
 int &select, cd_t &comp[3], PBlock QTable[4],
 int &window, int &bit_count, int &stuffers,
 int &HTable[4][384], int &n_comp)
{
  int G_ZZ[64];
  G_ZZ[0]=  0, G_ZZ[1] = 1, G_ZZ[2] = 8, G_ZZ[3] =16,
  G_ZZ[4]=  9, G_ZZ[5] = 2, G_ZZ[6] = 3, G_ZZ[7] =10,
  G_ZZ[8]= 17, G_ZZ[9] =24, G_ZZ[10]=32, G_ZZ[11]=25,
  G_ZZ[12]=18, G_ZZ[13]=11, G_ZZ[14]= 4, G_ZZ[15]= 5,
  G_ZZ[16]=12, G_ZZ[17]=19, G_ZZ[18]=26, G_ZZ[19]=33,
  G_ZZ[20]=40, G_ZZ[21]=48, G_ZZ[22]=41, G_ZZ[23]=34,
  G_ZZ[24]=27, G_ZZ[25]=20, G_ZZ[26]=13, G_ZZ[27]= 6,
  G_ZZ[28]= 7, G_ZZ[29]=14, G_ZZ[30]=21, G_ZZ[31]=28,
  G_ZZ[32]=35, G_ZZ[33]=42, G_ZZ[34]=49, G_ZZ[35]=56,
  G_ZZ[36]=57, G_ZZ[37]=50, G_ZZ[38]=43, G_ZZ[39]=36,
  G_ZZ[40]=29, G_ZZ[41]=22, G_ZZ[42]=15, G_ZZ[43]=23,
  G_ZZ[44]=30, G_ZZ[45]=37, G_ZZ[46]=44, G_ZZ[47]=51,
  G_ZZ[48]=58, G_ZZ[49]=59, G_ZZ[50]=52, G_ZZ[51]=45,
  G_ZZ[52]=38, G_ZZ[53]=31, G_ZZ[54]=39, G_ZZ[55]=46,
  G_ZZ[56]=53, G_ZZ[57]=60, G_ZZ[58]=61, G_ZZ[59]=54,
  G_ZZ[60]=47, G_ZZ[61]=55, G_ZZ[62]=62, G_ZZ[63]=63;

  int DC_CLASS = 0;
  int HUFF_EOB = 0x00;
  int HUFF_ZRL = 0xF0;
  uint i, run, cat;
  int value;
  uint8 symbol;
  int AC_CLASS = 1;
  /* Init the block with 0's: */
  for (i = 0; i < 64; i++) T.linear[i] = 0;

  /* First get the DC coefficient: */
  symbol =
    get_symbol(c, HUFF_ID (DC_CLASS, comp[select].DC_HT),
               HTable, bit_count, window, stuffers);
```

```
  value =
    reformat(get_bits(c, symbol, bit_count, window,
                      stuffers), symbol);

  value += comp[select].PRED;
  comp[select].PRED = value;
  T.linear[0] = value * QTable[comp[select].QT].linear[0];

  /* Now get all 63 AC values: */
  for (i = 1; i < 64; i++) {
    symbol =
      get_symbol(c, HUFF_ID(AC_CLASS, comp[select].AC_HT),
                 HTable, bit_count, window, stuffers);

    if (symbol == HUFF_EOB) break;
    if (symbol == HUFF_ZRL) {
      i += 15;
      continue;
    }
    cat = symbol & 0x0F;
    run = (symbol >> 4) & 0x0F;
    i += run;

    value = reformat(get_bits(c, cat, bit_count,
                              window, stuffers), cat);

    /* Dequantify and ZigZag-reorder: */
    T.linear[G_ZZ[i]] =
      value * QTable[comp[select].QT].linear[i];
  }
  return T;
}

void memmove(int8& ColorBuffer[60000],
             PBlock src)
{
  int i, j;
 for (i=0;i<64;i++) {
   ColorBuffer[i]= src.linear[i];
 }
}

int memmove2(int8 &FrameBuffer[60000],
             int frame_start,
             int8 ColorBuffer[60000],
             int color_start, int n_comp,
             int good_cols)
{
  int i, j;
  for(i=0;i<good_cols*n_comp;i++)
    FrameBuffer[frame_start++] = ColorBuffer[color_start++];
}

int
process_MCU(int c, int &MCU_column, int &MCU_row,
            int &mx_size, int &my_size,
            int &in_frame, int &MCU_valid[10],
            int &curcomp, FBlock & FBuff,
            PBlock & MCU_buff[10], cd_t &comp[3],
            PBlock &QTable[4], int & window,
            int &bit_count, int &stuffers,
            int &HTable[4][384], int &n_comp,
            int8 &ColorBuffer[60000], int &MCU_sx,
            int &MCU_sy, int& x_size,
            int& y_size, int& rx_size, int& ry_size,
            int8 & FrameBuffer[60000])
{
  int i;
  int64 offset;
  int goodrows, goodcolumns;
  FBlock IDCT_buff[10];

  if (MCU_column == mx_size) {
    MCU_column = 0;
    MCU_row++;
    if (MCU_row == my_size) {
      in_frame = 0;
      return 0;
    }
  }

  for (curcomp = 0; MCU_valid[curcomp] != -1; curcomp++) {
    int temp = MCU_valid[curcomp];
    unpack_block(c, FBuff, temp, comp, QTable, window,
                 bit_count, stuffers, HTable, n_comp);
    MCU_valid[curcomp]= temp;
    IDCT_buff[curcomp] = FBuff;
    /* pass index to HT,QT,pred */
  }
  parallelize_IDCT(IDCT_buff, MCU_buff, curcomp-1);

  /* YCrCb to RGB color space transform here */
  if (n_comp > 1)
    color_conversion(ColorBuffer, MCU_buff, comp,
                     MCU_sx, MCU_sy);
  else memmove(ColorBuffer, MCU_buff[0]);

  /* cut last row/column as needed */
  if ((y_size != ry_size) && (MCU_row == (my_size - 1)))
    goodrows = y_size - ry_size;
  else
    goodrows = MCU_sy;

  if ((x_size != rx_size) && (MCU_column == (mx_size - 1)))
    goodcolumns = x_size - rx_size;
  else
    goodcolumns = MCU_sx;

  offset = n_comp * (MCU_row * MCU_sy * x_size +
                     MCU_column * MCU_sx);
  for (i = 0; i < goodrows; i++)
    memmove2(FrameBuffer, offset + n_comp * i * x_size,
             ColorBuffer,  n_comp * i * MCU_sx,
             n_comp , goodcolumns);

  MCU_column++;
  return 1;
}

/* Available cells, top of storage: */
int MAX_CELLS(int hclass)
{
  return MAX_SIZE(hclass) - 32;
}

int load_huff_tables(int c, int &HTable[4][384])
{
  int8 aux;
  int size, hclass, id;
  int LeavesN, NodesN, CellsN;
  int MaxDepth, i, k, done;
  int NextCellPt;  /* where shall we put next cell */
  int NextLevelPt; /* where shall node point to */
  int64 flag;

  uint8 size_aux;
  size_aux = next c;
  size = (size_aux << 8) | (next c);    /* big endian */
  size -= 2; /* size is the tables' size */

  while (size > 0) {
    aux = next c;
    hclass = first_quad (aux);  /* AC or DC */
    id = second_quad (aux);     /* table no */

    if (id > 1) return -1;
    id = HUFF_ID (hclass, id);
    size--;
    CellsN = NodesN = 1;        /* the root one */
    LeavesN = 0;

    for (i = 0; i < MAX_CELLS (hclass); i++)
      HTable[id][i] = 0; /* secure memory with crash value */

    /* first load the sizes of code elements */
    /* and compute contents of each tree level */
    /* Adress   Content         */
    /* Top              Leaves 0        */
    /* Top-1    Nodes  0        */
    /* ......   .......        */
    /* Top-2k   Leaves k        */
    /* Top-2k-1 Nodes  k        */
    MaxDepth = 0;
    for (i = 0; i < 16; i++) {
      LeavesN = HTable[id][MAX_SIZE (hclass) - 2 * i - 1] =
        next c;
```

```
      CellsN = 2 * NodesN;      /* nodes are old */
      NodesN = HTable[id][MAX_SIZE (hclass) - 2 * i - 2] =
        CellsN - LeavesN;
      if (LeavesN)
        MaxDepth = i;
    }
    size -= 16;
    /* build root at address 0, then deeper levels at */
    /* increasing addresses until MAX_CELLS reached */

    /* GOOD_NODE_FLAG;  points to cell _2_ !*/
    HTable[id][0] = 1 | 0x100;
    /* we give up address one to keep left children
       on even adresses */
    NextCellPt = 2;
    i = 0;      /* this is actually length 1 */

    done = 0;

    while (i <= MaxDepth) {
      /* then load leaves for other levels */
      LeavesN = HTable[id][MAX_SIZE (hclass) - 2 * i - 1];
      for (k = 0; k < LeavesN; k++)
        if (!done) {
          int temp = next c;
          /* Good Leaf Flag */
          HTable[id][(NextCellPt++)] = temp | 0x200;
          if (NextCellPt >= MAX_CELLS (hclass))
            done = 1;
        } else
          next c; /* throw it away, just to keep file sync */
      size -= LeavesN;

      if (done || (i == MaxDepth)) {
        i++;
        continue;
      }
      /* skip useless node building */

      /* then build nodes at that level */
      NodesN = HTable[id][MAX_SIZE (hclass) - 2 * i - 2];

      NextLevelPt = NextCellPt + NodesN;
      for (k = 0; k < NodesN; k++) {
        if (NextCellPt >= MAX_CELLS (hclass)) {
          done = 1;
          break;
        }

        flag = (((NextLevelPt | 1) >=
                 MAX_CELLS (hclass))) ? 0x300 : 0x100;
        /* we OR by 1 to check even right brother
           within range */
        HTable[id][(NextCellPt++)] =
          (NextLevelPt / 2) | flag;
        NextLevelPt += 2;

      }

      i++; /* now for next level */
    }   /* nothing left to read from file after maxdepth */
  }
  return 0;
}

struct PBlock {
  uint8 linear[64]; /* block of pixel-space values */
};

struct cd_t {
  int8 CID;   /* component ID */
  int8 IDX;   /* index of first block in MCU */

  int8 HS;    /* sampling factors */
  int8 VS;
  int8 HDIV;  /* sample width ratios */
  int8 VDIV;

  int8 QT;    /* QTable index, 2bits  */
  int8 DC_HT; /* DC table index, 1bit */
  int8 AC_HT; /* AC table index, 1bit */
  int8 PRED;  /* DC predictor value */
};
```

```
struct FBlock {
  int linear[64]; /* block of frequency-space values */
};

/* Returns ceil(N/D). */
int ceil_div(int N, int D)
{
  int i = N / D;
  if (N > D * i) i++;
  return i;
}

/* Returns floor(N/D). */
int floor_div(int N, int D)
{
  int i = N / D;
  if (N < D * i) i--;
  return i;
}

bool RST_MK(int32 x)
{
  return ((0xFFF8 & (x)) == 0xFFD0);
}  /* is x a restart interval ? */

int32 first_quad(int32 c)
{
  return c >> 4;
}

/* first 4 bits in file order */
int32 second_quad(int32 c)
{
  return c & 15;
}

int32 HUFF_ID(int32 hclass, int32 id)
{
  return 2 * hclass + id;
}

void read_file(int32 &c)
{
  int32 i = 0;

  while (1) {
    $$ VAL (0) = (int) getchar (); $$
    if (c == -1)
      return;
    next c;
  }
}

int32 load_quant_tables(int32 c,
      PBlock &QTable[4], int32 &QTvalid[4])
{
  int8 aux;
  int32 size, n, i, id, x;

  int32 temp;
  temp = next c;
  size = (temp << 8) | (next c); /* big endian */
  /* size is the tables' size */
  n = (size - 2) / 65;

  for (i = 0; i < n; i++) {
    aux = next c;
    if (first_quad (aux) > 0) return -1;
    id = second_quad (aux);

    QTvalid[id] = 1;
    for (x = 0; x < 64; x++)
      QTable[id].linear[x] = next c;
  }
  return 0;
}

/* initialise MCU block descriptors */
int init_MCU(int &MCU_valid[10], cd_t &comp[3],
             int &n_comp, int &mx_size,
             int &my_size, int &rx_size,
             int &ry_size, int &MCU_sx, int &MCU_sy,
```

```
            int &x_size, int &y_size)
{
  int i, j, k, n, hmax = 0, vmax = 0;

  for (i = 0; i < 10; i++)
    MCU_valid[i] = -1;

  k = 0;

  for (i = 0; i < n_comp; i++) {
    if (comp[i].HS > hmax)
      hmax = comp[i].HS;
    if (comp[i].VS > vmax)
      vmax = comp[i].VS;
    n = comp[i].HS * comp[i].VS;

    comp[i].IDX = k;
    for (j = 0; j < n; j++) {
      MCU_valid[k] = i;

      k++;
      if (k == 10) return -1;
    }
  }

  MCU_sx = 8 * hmax;
  MCU_sy = 8 * vmax;
  for (i = 0; i < n_comp; i++) {
    /* if 1 shift by 0 */
    comp[i].HDIV = (hmax / comp[i].HS > 1);
    /* if 2 shift by one */
    comp[i].VDIV = (vmax / comp[i].VS > 1);
  }

  mx_size = ceil_div (x_size, MCU_sx);
  my_size = ceil_div (y_size, MCU_sy);
  rx_size = MCU_sx * floor_div (x_size, MCU_sx);
  ry_size = MCU_sy * floor_div (y_size, MCU_sy);

  return 0;
}

void skip_segment(int32 c) /* skip a segment we don't want */
{
  uint32 size;
  int32 i;

  int32 aux;

  aux = next c;
  size = (aux << 8) | (next c); /* big endian */

  for (i = 0; i < size - 2; i++)
    next c;
}

int32 get_size(int32 c)
{
  uint32 aux;

  aux = next c;
  return (aux << 8) | (next c); /* big endian */
}

void get_next_MK(int32 c, int32 & passed, uint32 & ret_val)
{
  int32 ffmet;
  int32 locpassed;
  ffmet = 0;
  locpassed = -1;

  passed--;                     /* as we fetch one anyway */
  for (;;) {
    next c;
    if (c == -1) return;
    if (c == 0xFF) {
      ffmet = 1;
    } else if (c == 0x00) {
      ffmet = 0;
    } else {
      if (ffmet)            {
        ret_val = (0xFF00 | c);
        return;
      } else {
        ffmet = 0;
        if (c == -1) return;
      }
    }
    locpassed++;
    passed++;
  }
}

void jpeg_decoder()
{
  uint32 aux;
  int32 mark;
  int32 in_frame, x_size, y_size;
  int32 passed;
  int32 n_restarts, restart_interval, leftover;
  int32 j;
  int32 c, d;
  int n_comp;
  int stuffers = 0;

  passed = 0;

    read_file (c);
  par {
    int32 MCU_valid[10], MCU_sx, MCU_sy, MCU_row, MCU_column;
    int32 debug, mx_size, my_size, rx_size;
    int32 ry_size, bit_count;
    int32 QTvalid[4];
    FBlock FBuff;               /* Frame Buffer */
    PBlock PBuff;               /* Pixel Buffer */
    PBlock QTable[4], MCU_buff[10];
    cd_t comp[3];

    int32 HTable[4][384];
    /*Allocate storage*/
    int8 ColorBuffer[60000];
    int8 FrameBuffer[60000];

    int curcomp;
    uint8 window;

    get_next_MK (c, passed, aux);
    in_frame = 0;

    restart_interval = 0;
    for (int i = 0; i < 4; i++)
      QTvalid[i] = 0;

    while (1) {
      get_next_MK (c, passed, mark);

      switch (mark) {
      case 0xFFC0:
        in_frame = 1;
        get_size (c);   /* header size, don't care */

        /* load basic image parameters */
        next c;         /* precision, 8bit, don't care */
        y_size = get_size (c);
        x_size = get_size (c);

        n_comp = next c;

        for (int i = 0; i < n_comp; i++) {
          /* component specifiers */
          comp[i].CID = next c;
          aux = next c;
          comp[i].HS = first_quad (aux);
          comp[i].VS = second_quad (aux);
          comp[i].QT = next c;
        }

        init_MCU(MCU_valid, comp, n_comp, mx_size, my_size,
          rx_size, ry_size, MCU_sx, MCU_sy, x_size, y_size);

        break;
      case 0xFFC4: /* DHT_MK */
        load_huff_tables (c, HTable);
        break;

      case 0xFFDB:              /*DQT_MK */
```

```
  load_quant_tables (c, QTable, QTvalid);
  break;

case 0xFFDD:              /*Restart Interval */
  get_size (c);   /* skip size */
  $$printf("Restart Interval");$$
  restart_interval = get_size (c);
  break;

case 0xFFFE:              /*Comment Segment */
  skip_segment(c);
  break;

case 0xFFDA:              /*SOS_MK */
  get_size(c);  /* don't care */
  aux = next c;
  if (aux != n_comp) return;
  for (int i = 0; i < n_comp; i++) {
    aux = next c;
    if (aux != comp[i].CID) return;
    aux = next c;
    comp[i].DC_HT = first_quad (aux);
    comp[i].AC_HT = second_quad (aux);
  }
  get_size (c);
  next c;         /* skip things */

  MCU_column = 0;
  MCU_row = 0;
  clear_bits (bit_count);
  reset_prediction (comp);

  /* main MCU processing loop here */
  if (restart_interval) {
    n_restarts =
     ceil_div(mx_size * my_size, restart_interval) - 1;
    leftover = mx_size * my_size -
      n_restarts * restart_interval;
    /* final interval may be incomplete */

    for (int i = 0; i < n_restarts; i++) {
      for (int j = 0; j < restart_interval; j++)
        process_MCU(c, MCU_column, MCU_row, mx_size,
                    my_size, in_frame, MCU_valid,
                    curcomp, FBuff, MCU_buff, comp,
                    QTable, window, bit_count,
                    stuffers, HTable, n_comp,
                    ColorBuffer, MCU_sx, MCU_sy,
                    x_size, y_size, rx_size,
                    ry_size, FrameBuffer);
      /* proc till all EOB met */

      get_next_MK (c, passed, aux);

      if (!RST_MK (aux)) return;
      else reset_prediction (comp);
      clear_bits (bit_count);
    }             /* intra-interval loop */
  } else
    leftover = mx_size * my_size;
  /* process till end of row without restarts */
  for (int i = 0; i < leftover; i++)
    process_MCU(c, MCU_column, MCU_row, mx_size,
                my_size, in_frame, MCU_valid, curcomp,
                FBuff, MCU_buff, comp, QTable, window,
                bit_count, stuffers, HTable, n_comp,
                ColorBuffer, MCU_sx, MCU_sy, x_size,
                y_size, rx_size, ry_size, FrameBuffer);
  in_frame = 0;
  break;

case 0xFFD9:
  RGB_save(FrameBuffer,x_size, y_size, n_comp);
  return;
  break;

default:
  if ((mark & 0xFFF0) == 0xFFE0) {
    skip_segment (c);
    break;
  }
  if (RST_MK(mark)) {
    //printf( "Yes resetting");;
      reset_prediction(comp);
      break;
    }
    /* if all else has failed ... */
    break;
  }
 }
}

void main()
{
  int a=10;
  jpeg_decoder();
}
```