Field Programmable Port Extender (FPX)

JPEG Processing Modules for Real-Time Decoding and Recoding

Chris Neely

Washington University, Applied Research Lab

June 19, 2002

Supported by: NSF

http://www.arl.wustl.edu/arl/projects/fpx

Outline

• Introduction
• Motivation of JPEG Recoder Module
• JPEG Encoding Concepts
• FPX JPEG Recoder
  • Up to N coefficients
  • Block Diagram
  • Image Results
  • Thumbnail size Images
  • Network Configuration
  • Hardware Synthesis
Hardware JPEG Application

- **FPX Video Recoder Module**
  This module was implemented to adjust the bandwidth of Motion JPEG video midway through the network. Control cells are sent to specify the number of coefficients to drop on each luminance/chrominance channel (YCbCr).

Motivation for JPEG Recoding

![Diagram showing MMX, FPX, Network, and NTSC connected to JPEG Compressed Video](image)
Motivation for JPEG Recoding

- Recoded Video Reduces Downstream Bandwidth
- Video continues to play but with less detail
- Congestion

Network

MMX

FPX

MMX

NTSC

Monitor

Flashes or No Video

Congestion

NTSC

Video

Field Programmable Port Extender (FPX)
JPEG Encoding Concepts

- First, encode the information that is the most valuable to human perception.
- Convert from RGB to a luminance chrominance representation to preserve the luminance because that is considered valuable.
- Divide into 8x8 block computational units because they are small and efficient for processing.
- The human eye functions like a low pass filter, so encode low spatial frequencies with more accuracy than high frequencies.
JPEG Encoding Concepts

- The compression ratio is controlled by varying the quantization tables. Fewer/smaller coefficients after quantization means greater compression.
- Use variable length representations for data that assign as few bits as possible to frequently occurring data patterns (entropy encoding).
- 0xFF Marker codes indicate the start and end of images.

Network Data format for Motion-JPEG Video

Motion JPEG Input Cell Format:
Raw ATM cells containing RAW JPEG

AAL0 Cell Header
0xFFE0 = Start of Image
4.1. Follow Zig-Zag Coding Sequence

Data Order is a Zig-Zag Sequence

This pattern begins by mapping the value at the lowest spatial frequency, the DC coefficient.

The first AC coefficient is next.

Now the Zig-Zag pattern

The last values to be coded are those that relate to the highest spatial frequencies.
4.1. Follow Zig-Zag Coding Sequence

Huffman Variable Length Code Tables

<table>
<thead>
<tr>
<th>Run/Sz</th>
<th>Co_len</th>
<th>Co_wrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>0/1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0/2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0/3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>F/6</td>
<td>16</td>
<td>ffffa</td>
</tr>
<tr>
<td>F/7</td>
<td>16</td>
<td>ffffb</td>
</tr>
<tr>
<td>F/8</td>
<td>16</td>
<td>ffffc</td>
</tr>
<tr>
<td>F/9</td>
<td>16</td>
<td>ffffd</td>
</tr>
<tr>
<td>F/A</td>
<td>16</td>
<td>fffe</td>
</tr>
</tbody>
</table>

The Huffman table for the Variable Length Code describes the run-length pairs: the zero-run and the number of zeros in the integer.

Variable Length Integers

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Integer Value</th>
<th>VLI Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>-3, -2, 2, 3</td>
<td>00, 01, 10, 11</td>
</tr>
<tr>
<td>3</td>
<td>-7, -4, 4, 7</td>
<td>000, 011, 100, 111</td>
</tr>
<tr>
<td>4</td>
<td>-15, -8, 8, 15</td>
<td>0000, 01111, 1000, 1111</td>
</tr>
<tr>
<td>5</td>
<td>-31, -16, 16, 31</td>
<td>000000, 0111111, 100000, 111111</td>
</tr>
<tr>
<td>6</td>
<td>-63, -32, 32, 63</td>
<td>00000000, 011111111, 10000000, 111111111</td>
</tr>
<tr>
<td>7</td>
<td>-127, -64, 64, 127</td>
<td>0000000000, 01111111111, 10000000000, 11111111111</td>
</tr>
<tr>
<td>8</td>
<td>-255, -128, 128, 255</td>
<td>00000000000, 011111111111, 100000000000, 111111111111</td>
</tr>
<tr>
<td>9</td>
<td>-511, -256, 256, 511</td>
<td>00000000000000, 0111111111111, 10000000000000, 1111111111111</td>
</tr>
<tr>
<td>10</td>
<td>-1023, -512, 512, 1023</td>
<td>00000000000000000000, 011111111111111, 10000000000000000000, 1111111111111111</td>
</tr>
</tbody>
</table>

The Variable Length Integers can be up to 11 bits for the DC component and 10 bits for the AC component.
Entropy Decoder Module

- Bit-serial shift register
  - Bit-serial data-in
  - Identifies 0xFFEx marker control codes
  - Removes byte stuffing (0xFF00)
  - If (vlc_state = ‘1’), then shifts overflow to VLC register for lookup operation
  - If (vli_state = ‘1’), then shifts overflow to store VLI based on lookup result
- Variable length data output
  - VLI (up to 11 bits) is reverse-sign extended to a fixed 12 bit output
  - VLC (up to 16 bits) is padded with zeros to a fixed 16 bit output
- 4 Huffman Code Tables implemented as Combinational Logic

Entropy Decoder Module

11110110 0000 11111110100111

First shift register is not full yet.
### Entropy Decoder Module

**Huffman Table Specifications**

- **Combinational Logic**
- **DataEN**
- **DATA**
- **RESET_L, CLK, SAMP_MODE**
- **HUFF_TBL_SEL**

**Decoded Data**

- **11110110 0000 1111 111110100111**

**FSM**

- **VLC_OUT**
- **VLI_OUT**
- **POSITION_OUT**
- **BLOCK_NUM_OUT**

**Serial Shift Reg.**

- **VLC**
- **VLI**

AC value (Zero Run = 1, Bits = 4)

- **1 11110110 0000 1111**

Filled first shift register, so shifting overflow to VLC.

Doing Lookup on VLC on each shift.

Field Programmable Port Extender (FPX)
Entropy Decoder Module

**Huffman Table Specifications**

- **Combinational Logic**
  - DataEN
  - DATA
  - RESET, CLK
  - SAMP_MODE
  - HUFF_TBL_SEL

**Serial Shift Reg.**

- **VLC**
  - VLC
  - VLI
  - VLI
  - POSITION_OUT
  - BLOCK_NUM_OUT

**VLC VLI**

- **Entropy Decoder Module**
  - VLC = 11110110
  - AC value (Zero Run = 1, Bits = 4)

**Found VLC = “11110110”**

**AC value (Zero Run = 1, Bits = 4)**

**Field Programmable Port Extender (FPX)**
Entropy Decoder Module

Field Programmable Port Extender (FPX)

Symmetric Entropy FSM

Field Programmable Port Extender (FPX)
Entropy Recoder Module

Found VLC = “11110110”
AC value (Zero Run = 1, Bits = 4)

Found VLI = “0000”,
Integer = -15.

Is the coefficient position <= N?
Perform Byte Stuffing

Byte Shifting Pipeline
- Shift Most significant byte to check for 0xFF’s
- Stuff 0x00 if 0xFF is found that is not a marker code

Looking for 0xFF…
Perform Byte Stuffing

Byte Shifting Pipeline
- Shift Most significant byte to check for 0xFF’s
- Stuff 0x00 if 0xFF is found that is not a marker code

Format INPUT CELLS.DAT

Simulation Testbench

Format INPUT CELLS.DAT
Simulation Waveforms – Finding VLCs and VLIs

Matched VLC
Stored VLI

Reformat SW_CELLSORT.DAT
Simulation JPEG Output – DC Values

Recoded for Greater Compression

(Normal video)
17.14 KB, 0.88 bits-per-pixel

up to 8 Y, 8 Cb, 8 Cr
11.78 KB, 0.61 bits-per-pixel
Recoded for Greater Compression

(Normal video)
17.14 KB, 0.88 bits-per-pixel

up to 4 Y, 4 Cb, 4 Cr
9.29 KB, 0.48 bits-per-pixel

Recoded for Greater Compression

(Normal video)
17.14 KB, 0.88 bits-per-pixel

up to 2 Y, 2 Cb, 2 Cr
5.60 KB, 0.289 bits-per-pixel
Recoded for Greater Compression

(Normal video)
17.14 KB, 0.88 bits-per-pixel

to Y, 1 Cb, 1 Cr
4.35 KB, 0.22 bits-per-pixel

Convert to “Thumbnail”

- 4 Pipeline Conversion Stages
  - Determine VLI size and convert to 12 bit signed integer
  - Add to 12 bit DPCM (DC running sum)
  - Denormalize (convert to 8 bit unsigned integer)
  - Average Blocks Horizontally

Field Programmable Port Extender (FPX)
• 4 Pipeline Conversion Stages
  – Determine VLI size and convert to 12 bit signed integer
  – Add to 12 bit DPCM (DC running sum)
  – Denormalize (convert to 8 bit unsigned integer)
  – Average Blocks Horizontally

Convert to “Thumbnail”
• 4 Pipeline Conversion Stages
  – Determine VLI size and convert to 12 bit signed integer
  – Add to 12 bit DPCM (DC running sum)
  – Denormalize (convert to 8 bit unsigned integer)
  – Average Blocks Horizontally
Sample Baseline Bitmap Output

Sample MMX Bitmap Output
Results

- How fast does it go?
  - 27.443 MHz, RAD_CLKB
- How much logic / memory does it require?
  - 22 logic levels
  - 45% BLOCKRAMs
  - 12% SLICEs

Motivation for JPEG Recoding
NCHARGE is used from a web browser to configure the module.
Module ID: 60 (0x3C)
Opcode: 0x04 Manually specify the number of coefficients per color component (N1 x Y, N2 x Cr, N3 xCb)
Opcode: 0x10 Preset to blocky color (1 x Y, 1 x Cr, 1 xCb)
Opcode: 0x12 Preset to blocky grayscale (1 x Y, 0 x Cr, 0 xCb)

NCHARGE Listing File:
<module>
  Video Recorder
  <input_opcodes>
  0x4,A,3,Lum,Cb,Cr,
  0x10,B,A
  <output_opcodes>
  0x5,A,3,Lum,Cb,Cr,
  0x11,B,0
  0x13,C,0
  <fields>
  Lum,x,1,2,1,16,
  Cb,x,1,4,1,8,
  Cr,x,1,6,1,0
  <fields>
  <help>
  A Manual set coefficients
  B Preset blocky color
  C Preset blocky grayscale
  <help>
# References

**MMX JPEG Format**

- Half-size images of NTSC fields (even or odd scan lines) are encoded as individual JPEG images (60 frames/sec)
- 0xFFxx Marker Codes identify the images:
  - 0xFFE0 = Start of Even Field
  - 0xFFE1 = Start of Odd Field
  - 0xFFE2 = End of Field
- Image Resolution: 640x248

---

**CS535 Project Logical Connection Diagram**

[Image depicting the logical connection diagram for CS535 project]
Connections: MMX Video station to the FPX VR

CS535 “Thumbnail” Image Decoder
1.1. Convert the Color Channels

The conversion from RGB to YCbCr uses the following equations:

1. Separate the Color Channels
1.1. Convert the Color Channels

\[Y = \frac{L_{\text{Red}} \cdot R + L_{\text{Green}} \cdot G + L_{\text{Blue}} \cdot B}{2} \]

\[C_b = \frac{B - Y}{2 - 2 \cdot L_{\text{Blue}}}\]

\[C_r = \frac{R - Y}{2 - 2 \cdot L_{\text{Red}}}\]

where:

\[L_{\text{Red}} = \frac{299}{1000}\]

\[L_{\text{Green}} = \frac{587}{1000}\]

\[L_{\text{Blue}} = \frac{114}{1000}\]

Note: 64-bit arithmetic is the recommended precision for this conversion.

Since most of the perceivable information comes from the luminance, we can sample the chrominance half as often as we sample the luminance with only a minor change in image quality.

Here, the sampling ratio used 4:2:2, which means that there is a 2:1 ratio between luminance to each chrominance channel.

1.2. Partition into 8x8 Blocks

The images is divided below into 8x8 pixel blocks before being encoded in the following MCU data order.
1.3. Form Minimum Coded Units

![Diagram of Luminance (Y) and Chrominance (Cr) blocks]

Chrominance Blue (Cb)

<table>
<thead>
<tr>
<th>M</th>
<th>0</th>
<th>L-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>f</td>
</tr>
<tr>
<td>...</td>
<td>c</td>
<td>g</td>
</tr>
<tr>
<td>N-1</td>
<td>d</td>
<td>h</td>
</tr>
</tbody>
</table>

MCU = a b c d

MCU = e f g h

Note: For an MMX Image M = 40, N = 31

2.2. DCT by Matrix Multiplication

```c
void forward_DCT ()
{
    double temp[N][N];
    double temp1 = 0;
    for (int i = 0; i < M; i++) {
        C[0][j] = 1.0 / sqrt(N);
        Ct[0][j] = C[0][j];
    }
    for (int i = 1; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = sqrt(2.0/N) * cos((2*j+1)*i*pi/(2.0*N));
            Ct[j][i] = C[i][j];
        }
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                temp[i][j] = 0.0;
                for (int k = 0; k < N; k++)
                    temp[i][j] += (Pixels[i][k] - 128) * Ct[k][j];
            }
        }
    }
}
```

Prepare a Cosine matrix, Ct, that is a linear combination of the basis functions for the transformation:

```
Ct:
0.353 0.49 0.461 0.415 0.353 0.277 0.191 0.097
0.353 0.277 -0.191 -0.097 -0.353 -0.49 -0.461 -0.277
0.353 -0.097 -0.461 -0.277 -0.353 -0.49 -0.461 -0.277
0.353 0.49 0.461 0.415 0.353 0.277 0.191 0.097
0.353 0.415 0.191 -0.097 -0.353 -0.49 -0.461 -0.277
0.353 -0.097 -0.461 -0.277 -0.353 -0.49 -0.461 -0.277
0.353 -0.277 -0.191 -0.097 -0.353 -0.49 -0.461 -0.277
0.353 -0.461 -0.415 -0.49 -0.461 -0.277 0.191 -0.097
```

Shift Pixel (source) between (-128) and 127

Matrix-Multiply Shifted Pixels by Ct

Field Programmable Port Extender (FPX)
The position of the values in the matrix after the DCT relates to their spatial frequencies. Increasing Horizontal Frequency

Increasing Vertical Frequency

DC

2-D Representations of the Basis Coverage from the DCT Corner Values

0 - Upper Left

28 - Upper Right

35 - Lower Left

63 - Lower Right

The human eye functions like a lowpass filter, and so some of the high frequency information can be removed without perceiving any loss.

Integer rounding is used for division. Q values (8 bits) increase with spatial frequency.

The human eye functions like a lowpass filter, and so some of the high frequency information can be removed without perceiving any loss.
4. Entropy Encoding

This pattern begins by mapping the the value at the lowest spatial frequency, the DC coefficient.

The first AC coefficient is next.

Now the Zig-Zag pattern

The last values to be coded are those that relate to the highest spatial frequencies.
4.2. Encoding the DC coefficient

After the DCT followed by quantization, the result might look like the matrix shown to the left.

The first value to be encoded in a block is the **DC coefficient**. This is the most important value of the block, so it can have up to 12 bits signed precision. It is coded as the difference from the DC value of the previous block.

4.2.1 Encoding the DC VLC

Huffman entropy coding is used to describe the number of bits used for the integer. This codeword precedes the integer and is called the **VLC**. The Huffman codes are kept in a table similar to the one shown left.
4.3.1 Encoding the AC VLC

The Huffman table for the VLC describes the run-length pairs: the zero-run and the number of zeros in the integer.

<table>
<thead>
<tr>
<th>Run/Sz</th>
<th>Co_len</th>
<th>Co_wrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/0</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>0/1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0/2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0/3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>F/6</td>
<td>16</td>
<td>ffffa</td>
</tr>
<tr>
<td>F/7</td>
<td>16</td>
<td>ffffb</td>
</tr>
<tr>
<td>F/8</td>
<td>16</td>
<td>ffffc</td>
</tr>
<tr>
<td>F/9</td>
<td>16</td>
<td>ffffd</td>
</tr>
<tr>
<td>F/A</td>
<td>16</td>
<td>fffe</td>
</tr>
</tbody>
</table>

4.3.2 Encoding the AC coefficients

The VLIs are encoded using the same representation. However, they range only from 0-10 bits.

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Integer Value</th>
<th>VLI Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>-3, -2, 2, 3</td>
<td>00, 01, 10, 11</td>
</tr>
<tr>
<td>3</td>
<td>-7, -4, 4, 7</td>
<td>000, 011, 100, 111</td>
</tr>
<tr>
<td>4</td>
<td>-15, -8, 8, 15</td>
<td>0000, 01111, 10000, 11111</td>
</tr>
<tr>
<td>5</td>
<td>-31, -16, 16, 31</td>
<td>00000, 011111, 100000, 111111</td>
</tr>
<tr>
<td>6</td>
<td>-63, -32, 32, 63</td>
<td>000000, 0111111, 1000000, 1111111</td>
</tr>
<tr>
<td>7</td>
<td>-127, -64, 64, 127</td>
<td>0000000, 01111111, 10000000, 11111111</td>
</tr>
<tr>
<td>8</td>
<td>-255, -128, 128, -128</td>
<td>00000000, 011111111, 100000000, 111111111</td>
</tr>
<tr>
<td>9</td>
<td>-511, -256, 256, 511</td>
<td>000000000, 0111111111, 1000000000, 1111111111</td>
</tr>
<tr>
<td>10</td>
<td>-1023, -512, 512, 1023</td>
<td>0000000000, 01111111111, 10000000000, 11111111111</td>
</tr>
</tbody>
</table>
4.3. Encoding the AC coefficients

Next, the AC coefficients are encoded in zig-zag order. Only the non-zero coefficients are encoded, so their position is kept track of by recording the run-length of zeros in between.

4.2.2 Encoding the DC VLI

The integer value, called the VLI, is coded as shown in this table. They range from 0-11 bits.

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Integer Value</th>
<th>VLI Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>-3, -2, 2, 3</td>
<td>00, 01, 10, 11</td>
</tr>
<tr>
<td>3</td>
<td>-7, -4, 4, 7</td>
<td>000, 001, 010, 011</td>
</tr>
<tr>
<td>4</td>
<td>-15, -8, 8, 15</td>
<td>0000, 0001, 0010, 0011</td>
</tr>
<tr>
<td>5</td>
<td>-31, -16, 16, 31</td>
<td>00000, 00001, 00010, 00011</td>
</tr>
<tr>
<td>6</td>
<td>-63, -32, 32, 63</td>
<td>000000, 000001, 000010, 000011</td>
</tr>
<tr>
<td>7</td>
<td>-127, -64, 64, 127</td>
<td>0000000, 0000001, 0000010, 0000011</td>
</tr>
<tr>
<td>8</td>
<td>-255, -128, 128, 255</td>
<td>00000000, 00000001, 00000010, 00000011</td>
</tr>
<tr>
<td>9</td>
<td>-511, -256, 256, 511</td>
<td>000000000, 000000001, 000000010, 000000011</td>
</tr>
<tr>
<td>10</td>
<td>-1023, -512, 512, 1023</td>
<td>0000000000, 0000000001, 0000000010, 0000000011</td>
</tr>
<tr>
<td>11</td>
<td>-2047, -1024, 1024, 2047</td>
<td>00000000000, 00000000001, 00000000010, 00000000011</td>
</tr>
</tbody>
</table>
JPEG “Thumbnail” Decoder

JPEG Decoding Example – 0xFF__Marker Codes

Entropy-coded data:

FFE0 3EC1 FF004E1 FF923 FF00 DC55

SOI Marker Code
### JPEG Decoding Example – Serial Bit Shifts

Entropy-coded data:

```
0011111011000001111111010011000011111
1111110010001111111111111111101100101010
```

SOI Marker Code

<table>
<thead>
<tr>
<th>VLC</th>
<th>VLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC value (Bits = 0)</td>
<td>0</td>
</tr>
<tr>
<td>AC value (Zero Run = 1, Bits = 4)</td>
<td>-15</td>
</tr>
<tr>
<td>AC value (Zero Run = 6, Bits = 4)</td>
<td>-15</td>
</tr>
<tr>
<td>AC value (Zero Run = 10, Bits = 4)</td>
<td>-14</td>
</tr>
<tr>
<td>AC value (Zero Run = 14, Bits = 4)</td>
<td>-13</td>
</tr>
</tbody>
</table>

EOB