# Hardware JPEG encoder IP

Martin Strubel // section5.ch

June 7, 2013

**Revision:**
0.2preliminary

# 1 Introduction

## 1.1 Overview

The JPEG encoder IP by www.section5.ch is a toolbox of various FPGA IP modules that allow pipelined or accelerated encoding of monochrome or colour pixel data into the well standardized JPEG format at arbitrary compression rates.

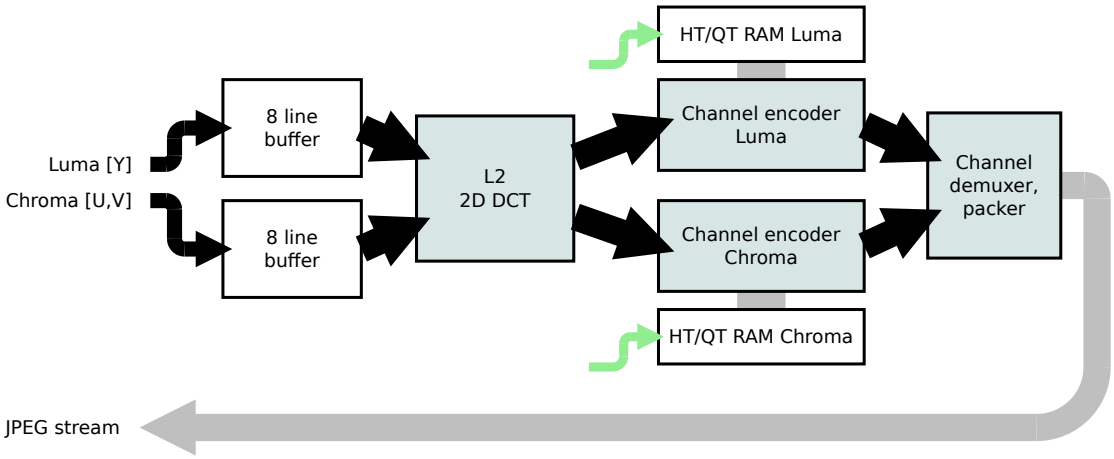


*Figure 1.1: L2 encoder schematic*

It is issued in two top level implementation variants:

- 'L1': A single channel, multiplexing JPEG encoder
- 'L2': Dual channel parallelizing version for fast, synchronous throughput (See block schematic in Fig. 1.1)

To explain functionality and usage of these encoder variants, a first quick introduction to JPEG format characteristics is given.
These encoder IP blocks do not directly emit a complete JPEG that can be opened by a standard JPEG reader. Instead, they strive to provide optimum flexibility and speed while being easy to integrate into existing system on chip (SoC) environments. For example, the developer has the choices of:

- Host CPU interface or soft core
- Full JFIF compliance or 'bare metal' streaming
- Configureable, internal or external memory blocks

A detailed description about the output options is found below in Section 1.4. The full JPEG standard is covered in the ITU/CCITT norm T.81 ([jpeg_t81] Appendix A.3)

## 1.2 JPEG imaging basics

The JPEG format is a wide spread and well established image compression format which typically uses a lossy compression that is barely visible in typical photography images at high

quality settings. There are extensions to the JPEG standard such as JPEG LS which preserve the entire image information during the compression process. This implementation however uses a standard JPEG baseline compression algorithm that is available without third party licensing involved.
JPEGs images are organized in blocks, so called MCUs (minimum coded unit). Depending on the input format, the MCUs can have a resolution of 8x8, 16x8 or 16x16 pixels. Each MCU is, as the term suggests, encoded in the frequency space and quantized such that information that is little relevant to the human perception is stripped. The measure of quantization is a factor that has direct impact on how well the image compresses.

The term MCU might lead to confusion regarding 'micro controller unit'. Therefore, we use the notation '$\mu$C' when referring to a micro controller.

Above a certain compression ratio of input towards output, the block boundaries become visible. These blocky artefacts are a drawback of the JPEG standard encoding, however, the algorithm has a few pros as well:

- Rather easy to implement in hardware
- Can be well pipelined or parallelized
- Uses little resources and does not need external image buffering or caching

## 1.3 Input formats

The JPEG standard, in particular the JFIF specification is fairly flexible, therefore it supports various image formats, and subsampling options for a number of image channels. For monochrome images, the process is straightforward, image data goes in and compressed data comes out.
For colour encoding, there are a few more options. Basically, the RGB information coming from a imaging source needs to be converted to YUV format – or also referred to as Luma and Chroma channels. Typical input formats to JPEG are YUV422 or YUV420. The numbers indicate the subsampling resolution. For further details, please refer to [subsampling] Appendix A.3. Unfortunately, there are various definitions of the YUV format (which is also sometimes referred to as YCbCr-Format).
The input format that is required to the JPEG encoder is however defined clearly by the RGB$\rightarrow$YUV conversion matrix found in the JPEG standard literature. For summary:

- Y ranges from 0..255
- U, V range from -128 to +127
- Additional level shifting takes place inside the decoder for the Y channel, but not for U, V when in the above range

The exact input format and encoder configuration is described in detail in the specific encoder sections.

### 1.3.1 Channels

Monochrome encoding just requires one channel whereas the full RGB information requires three. However, data coming from an imaging sensor might already suffer from redundancies, because the RGB information might be interpolated from a Bayer pattern, for example. It is therefore somewhat plausible that information can be omitted at no further visible quality

loss. The YUV coding plus the subsampling technique elaborated in the next paragraph allows us to use multiplexing of the data in order to reduce the general data amount to be processed. Basically, the JPEG encoder distinguishes between a Luma (Brightness) and a Chroma (Red- and Blueness) channel. Each channel type has its specific quantization and encoding method, according to the weighting of the human eye's perception.

### 1.3.2 Subsampling

Luma/Chroma subsampling is basically a data reduction method. The background idea is, to sample spatial color changes in an image at a lower resolution than the brightness information that is generally perceived in more detail by the human eye. For example, one Chroma value can be stored per 2x2 Luma pixels.
The following input formats are supported:

**YUV400 (monochrome)**
Supported by the L1 encoder or in two taps (double throughput) by the L2 encoder.

**YUV420**
Supported by the L1 encoder at 1.5x processing clock

**YUV422**
Supported by the L2 encoder at full pixel clock

Input format versus supported pixel clock rate is crucial to the selection of the appropriate encoder. The detailed properties are listed below in Chapter 2.
Note there are many variants of the above formats, such as:

- Planar vs. interleaved
- UV vs. VU ordering
- Level shifted or non level shifted

Depending on the subsampling, the input image dimensions are restricted to a integer multiple of the MCU dimensions. Images not matching this format have to be padded up to the next MCU multiple.
For example, in YUV420 format with 16x16 MCU size, 360x240 would not be a valid format, but 352x240 would be.

## 1.4 Output formats

As mentioned in the introduction, the encoder IP does not emit a fully JFIF compliant JPEG, but a compressed stream only. The JFIF header has to be constructed by external 'intelligence', such as a $\mu$C soft core or the receiving host.
Basically, a full JPEG output, for example to a MJPEG capable video display, follows the scheme displayed in Fig. 1.2.
First, the memory tables are initialized according to the quality configuration of the encoder. Then, the $\mu$C initializes the JFIF header on the heap or in a static memory area. Then inside the video encoding main loop, the following sequence takes place:

1. The $\mu$C sends the header to the peripheral
2. The JPEG encoder is enabled to send the compressed actual image data stream
3. The $\mu$C terminates the stream and stores the image size for further transmission

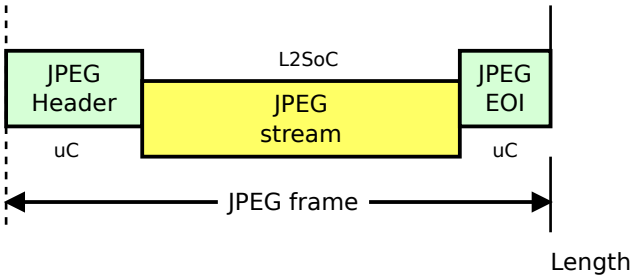


*Figure 1.2: JPEG output scheme*

The hard encoder IP leaves all format specifics to the application. Therefore, the toolkit remains somewhat flexible for future adaptations and format extensions for custom purposes such as medical imaging or machine vision.
The typical reference demo application allows to stream a MJPEG video to a browser or player such as freely available reference applications: mplayer, VLC, etc. (see also [players] Appendix A.3)

## 1.5 Application scenarios

Below, a few possible example scenarios are given, that can be realized using this JPEG encoder IP suite.

1. Standalone JPEG encoder with external CPU. In this case, the FPGA will act as a JPEG accelerator pre- or postprocessor
2. Fully independent, intelligent JPEG processor (Raw colour sensor image in, JPEG image out)
3. Custom multi channel JPEG encoding pipeline with further processing by a DSP

With respect to the output format, the IP cores are flexible but scalable and cooperate easily with embedded processors or built-in soft CPUs such that Header and stream assembly can happen on an external hardware by the user or directly on the FPGA. The controlling firmware is written in portable C code such that it can be run on a PC and on the soft core in the FPGA, likewise.

# 2

# Encoder implementations

## 2.1 Common encoder basics

Both the L1 and L2 encoder rely on external, preinitialized block memory that contains the Huffman and Quantization tables. The data layout is shown in Table 2.1.

| Address offset mnemonic [Bytesize] | Description |
|---|---|
| JPEGMEM_QT_LUMA[128] | Luminance channel quantization table |
| JPEGMEM_QT_CHROMA[128] | Chrominance channel quantization table |
| JPEGMEM_HT_CODE[1024] | Huffman code table |
| JPEGMEM_HT_SIZE[1024] | Huffman size table |

*Table 2.1: External memory map*

The L1 encoder allows full configuration of the above memory banks, whereas the L2 encoder restricts the configuration option by default to the Quantization (QT) memory bank only. However, the L2 can be configured for full configuration access at the cost of higher RAM block usage.
Normally, the Huffman tables are fixed and do not need external configuration as they are preinitialized to default values upon start.

## 2.2 Single channel 'L1' encoder

The first version of this encoder is a fully pipelined encoder that accepts a single channel image source as input. This can be either a monochrome channel, or a interleaved sequence of YUV blocks. A preprocessor must split the source image into the proper MCU sequence and store them in a MCU buffer of the size 64x64 of 16 bit words. The DCT engine picks the data up from this buffer and signals when it has processed the entire MCU. Typically, a ping pong technique is used to swap buffers that were just filled against the ones just processed.
For monochrome encoding, this process is rather straight forward. The user typically implements an input buffer of two MCU sizes that are swapped by a "pingpong" signal. For colour encoding, the process is somewhat more complicated. First it should be noted that the MCU input data sequence must follow a specific order, depending on the input format (YUV420, YUV422, YUV444).
For details on how to use this encoder module, please refer to the developer reference [jpegdoxy] Appendix A.3.

## 2.3 Dual channel 'L2' encoder

This dual channel version uses the full bandwidth of the DCT engine to process two channels (typically Luma/Chroma) in parallel. This avoids the need for overclocking the engine with respect to the input pixel clock, therefore processing can happen entirely synchronously. Instead of one multiplexed input channel, there are two simultaneously operating channel compressors A and B with fixed code selection (typically: a dedicated Luma and Chroma channel). By default, this system supports the YUV422 input mode. The encoder interface is

designed similar as the L1 encoder such that it picks a YUYV block sequence from an external buffer.
Again, detailed information for integration is found in the developer reference ([jpegdoxy] Appendix A.3).

## 2.4 JPEG system on chip ('L2SoC')

The JPEG SoC implementation is based on the L2 encoder and directly accepts Y[UlV] pairs in parallel from an external source such as a Bayer to YUV decoder. For configuration and status query, it supplies a $\mu$C compatible read and write port plus a few direct signal input pins. All input buffering is taken care of by the JPEG L2 SoC. There is no handshaking necessary, data can move at full bandwidth from the input to the compressed data output.

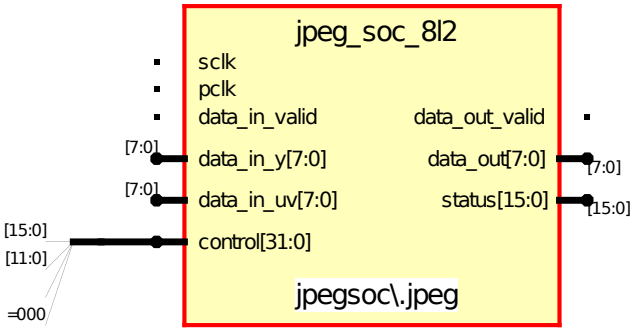


*Figure 2.1: JPEG SoC IP module*

The *L2SoC* is configureable for the following scenarios:

- YUV422 encoding: Simultaneous input of data streams on both channels in the following sequence
  (A,B) := [Y0,U0], [Y1,V0], [Y2,U1], [Y3,V1], ...
- Dual tap monochrome encoding
  (A,B) := [Y0, Y1], [Y2, Y3], ...
  This is suitable for monochrome cameras that operate at two taps. For example, 160 MHz effective pixel throughput can be achieved at 80 MHz pixel clock and two simultaneous even/odd pixel output channels.

The detailed signal description of the L2SoC architecture is found in Section 3.4.1.

# 3

# Usage and register reference

Typically, the high level L2SoC module is used directly by the system integrator. Lower level usage of the L1 and L2 encoders is possible for custom purposes, but not documented here.

## 3.1 Common encoder properties

All SoC encoder versions are configureable via a 16 bit register interface. Table 3.1 lists the currently available SoC configurations of the encoder toolchain. The version code can be used by the firmware to synchronize with the corresponding versions of the IP core.

The version code is always located at the end of the register map and may move to the next power of two minus one address for future expansions (while remaining downward compatible).

| Version code | Description |
|---|---|
| `ff01` | Core test bench encoder version 1 (Section 3.4.2) |
| `0a01` | Custom Alpha version 1 (Section 3.4.3) |

*Table 3.1: Available SoC encoder versions*

## 3.2 Base Register map

The JPEG encoder memory map is split into a register bank ('JPEG') and a memory bank ('JPEGMEM'). The register settings are not hard coded but defined in a XML device description. All code definitions and address decoding is generated from this XML file which is following an internal 'devdesc' standard ([devdesc] Appendix A.3).
To determine the encoder version, the software must read the Version code from the `VERSION` register. All derived encoder versions of the L2SoC class share the Base registers and bits. Extended or custom versions are augmented by a few custom registers or bits. If the register map is changed, an encoder receives a new high byte in the version code.

The detailed implementation and documentation source for the custom registers is always found in the corresponding XML device description file. See Section 3.4 for specific filename (relative to distribution folder)

Make sure to use the **generated** register constant packages (VHDL) and C headers **only**. Only final, frozen versions whose version code does not start from 0xff00 guarantee the correctness of the register map hard copy.
The registers do not have default values on most platforms. It is therefore mandatory to initialize them correctly by the $\mu$C firmware.

### 3.2.1 JPEG

JPEG encoder configuration map

| Offset [Span] | Name(Id) | Access | Description |
|---|---|---|---|
| 0x0000 [2] | CONTROL | WO | Control register |
| 0x0000 [2] | STATUS | RO | Status register |
| 0x0002 [2] | IMG_WIDTH | RW | Image width minus one |
| 0x0004 [2] | NUM_MCUS | RW | Total number of 8x8 MCUs of image frame minus one |
| 0x0006 [2] | MAX_WIDTH | RO | Maximum image width of the input buffer (configured at synthesis) |
| 0x001c [2] | CAPABFLAGS | RO | Capability flags. Reserved for future variations. |
| 0x001e [2] | VERSION | RO | Version ID |

*Table 3.2: Address map JPEG starting at absolute address 0x1000*

### 3.2.2 JPEGMEM

JPEG encoder shared memory table area

| Offset [Span] | Name(Id) | Access | Description |
|---|---|---|---|
| 0x0400 [0] | TABLE_QUANT | WO | Quantization table offset (legacy, do not use for new code ) |
| 0x0400 [128] | QT_LUMA | WO | Quantization table offset Luma |
| 0x0480 [128] | QT_CHROMA | WO | Quantization table offset Chroma |
| 0x0800 [1024] | HT_CODE | WO | Huffman code table offset |
| 0x0c00 [1024] | HT_SIZE | WO | Huffman size table offset |

*Table 3.3: Address map JPEGMEM starting at absolute address 0x1000*

### 3.2.3 Detailed register description

See Table 3.4 and following.

| Bit(s) | Name | Description |
|---|---|---|
| 10:10 | FORMAT | 0: YUV422, 1: YUV400 (monochrome) dual tap |
| 7:7 | START | Toggle 1-0 to reset the engine and start a new frame |
| 6:6 | FIFORESET | HIGH active full reset to clear JPEG engine FIFO overruns |
| 5:5 | LEVELSHIFT_B | Level shift channel B if set (data_b -= 128) |
| 4:4 | LEVELSHIFT_A | Level shift channel A if set (data_a -= 128) |

*Table 3.4: CONTROL (Address: 0x0000)*

| Bit(s) | Name | Description |
|---|---|---|
| 4:5 | DEMUX_OVERRUN | Channel demuxer overrun |
| 3:3 | OVERRUN | FIFO overrun error |
| 1:2 | DCTERR | DCT error in X(0) or Y(1) |
| 0:0 | DONE | 1: JPEG encoding done |

*Table 3.5: STATUS (Address: 0x0000)*

## 3.3 General JPEG SoC usage

Basically, the encoding process is simple: Pass data to the encoder in the correct order, receive encoded data on the output with a certain latency. However, there is some framing to take care of. The detailed configuration sequence is demonstrated in the firmware for the L2SoC $\mu$C (`jpeg.c`)

### 3.3.1 Preconfiguration

Before any data is transmitted to the encoder, configuration needs to take place – the encoder has to know what amount of data to expect. This is taken care of by the following two registers:

**IMG_WIDTH**
: Width of encoded image (the true line width) minus one

**NUM_MCUS**
: Number of total 8x8 blocks minus one. This is calculated using the formula below

$$n_{MCU} = x * y * n_{ch}/64 - 1 \tag{3.1}$$

where

$x, y$
: Resolution of the image

$n_{ch}$ Number of channels, e.g. YUV400: 1, YUV422: 2

Examples:

- For a 640x480 colour YUV422 image, IMG_WIDTH is 639, NUM_MCUS is 9599
- For a 320x200 monochrome image, IMG_WIDTH is 319, NUM_MCUS is 999

Next, the $\mu$C must initialize the quantization tables according to the quality selection factor. The quantization tables are stored in zig zag order and basically contain 1:15 fractional values. The detailed generation of the tables is part of the $\mu$Cs firmware.
For the default configuration (`FORMAT` bit cleared), the Quantization table banks A/B are written with the Luma/Chroma data. For the monochrome dual tap configuration (`FORMAT` bit set), **both** Quantization tables are initialized with the Luma records.

### 3.3.2 Streaming

Prior to each image frame, the engine must be synchronously reset using the `START` flag. This is also separately available as input pin on some control ports (see Section 3.4.1). When the encoder is done encoding the current frame data, it sets the `DONE` bit high, signalling it is ready for a new START pulse. The start signal can be asserted as short as one peripheral clock cycle (`pclk`). When asserting via the START bit, be sure that the system clock (`sclk`) of your $\mu$C is sufficiently high with respect to the `pclk`. For the clock description, see Section 3.4.1 below.
When streaming synchronously from an imaging sensor, it is important to note that the encoder has only restricted buffering capabilities. It is up to the system integrator to make sure that the header preparation and transmission is occuring in time during video blanks.

#### Data input

Data is fed to the encoder by a push scheme, i.e. a 'data valid' signal (H active) and a data port per channel. The data must arrive simultaneously as listed in the detailed, encoder specific input scheme described below. The data valid signal does not necessarily have to be asserted over the entire input period of a image block, so line blanking can occur.
For a partially subsampled Luma/Chroma configuration, the Luma data is typically fed through channel A, the Chroma data (UV) interleaved through channel B as described in Section 2.4.
The L2SoC does an internal level shift (subtraction of 128) depending on whether the according LEVELSHIFT bit is set, see Table 3.4.
Table 3.6 shows the default configuration for the Bayer→YUV422 *Cottonpicken* demosaicer engine.

| Bit | Value |
|---|---|
| LEVELSHIFT_A | 1 |
| LEVELSHIFT_B | 0 |

*Table 3.6: Default level shift configuration*

Note: Some optimized encoder versions may not allow to configure the level shifting. See specific encoder version description.

#### Status and Error handling

During streaming, errors can occur under certain circumstances. These are reported by sticky bits in the `STATUS` register. If any of the error signals are asserted, the JPEG encoder must be reset using `FIFORESET` and restarted using the `START` bit.
A detailed description of the status bits:

**DONE**
This bit is asserted high when the encoding has been completed, according to the data amount specified by the IMG_WIDTH and NUM_MCUS register.

**DCTERR[1:0]**
These bits are sticky HIGH if an overflow in the DCT (0: X, 1: Y) occured due to illegal input values. This can never happen when using 8 bit input data. These bits are reset by toggling the START bit.

**OVERRUN**
This bit is set if a FIFO overrun occured in one of the input channels. This can only happen on incorrect timing of the SoC implementation and should never occur in the final implementation. This bit is reset by toggling the FIFORESET flag.

**DEMUX_OVERRUN[1:0]**
Sticky bits that denote overrun in the output demuxer channel FIFOs. This can only happen on a misconfiguration of the channel encoders. Reset by toggling the FIFORESET flag.

## 3.4 L2SoC family

### 3.4.1 Common signals

All Variants of the L2 encoder SoC supports the full base register set.
The basic hardware pins are described again below:

**sclk**
The $\mu$C system clock for configuration

**pclk**
The pixel clock (processing clock)

**data_in_valid**
High when input data valid

**data_in_a**
Data input channel A (typically: Luma)

**data_in_b**
Data input channel B (typically: Chroma)

**jpeg_valid**
JPEG data out valid

**jpeg_data**
JPEG data stream, 8 bit

**control**
JPEG control input port

**status**
JPEG status output port

The control/status ports are implemented as records and may have varying member sets depending on the encoder version. Please see the detailed up to date encoder implementation documentation (Doxygen generated, typically residing in `doc/html` of your encoder distribution directory).
For all L2 SoC encoders, the common members are:

**Port** `control`**:**

**cfg_we**
Config $\mu$C port write enable (H active)

**cfg_addr**
Configuration address (12 bit)

**cfg_data**
Configuration data (16 bit)

**Port** `status`**:**

**data**
Status output data according to `cfg_addr`

### 3.4.2 Version TB 01 (0xff01)

The test bench version of the L2SoC. This is the version for full compliance tests and verification of the hardware against the simulation. It is featured by a MIPS CPU as test pattern generator. Restrictions:

- HT_CODE and HT_SIZE are not writeable
- The input format is fixed at YUV422 in YU/YV order.

| Supported input format | 8 bit YUV422 or YUV400 |
|---|---|
| XML device description | soc/iomap.xml |
| Configuration Interface | MCU 16 bit data, 32 bit address |
| Throughput latency in PCLK cycles | *TODO* |

*Table 3.7: Properties of TB01 version*

Note that the total throughput latency adds an extra overhead of 8 image scanlines, i.e. the acquisition duration of $8 * (w + b_h)$ cycles, where

$w$ Image width

$b_h$ Horizontal blank time

### 3.4.3 Version alpha 01 (0x0a01)

The alpha01 version is separately documented in [jpegdoxy] Appendix A.3.

| Supported input format | 8 Bit monochrome (YUV400) |
|---|---|
| XML device description | soc/iomap.xml |
| Configuration Interface | $I^2C$ |
| Throughput latency in PCLK cycles | *TODO* |

*Table 3.8: Properties Version alpha 01*

| | |
|---|---|
| Supported input format | 12 bit, arbitrary JFIF compliant YUV sequences |
| XML device description | ghdlex/ghdlsim.xml |
| Configuration Interface | VirtualBus Addr/Data |
| Throughput latency in PCLK cycles | 204 |

*Table 3.9: Properties of L1SoC Demonstrator*

### 3.4.4 Early L1SoC test bench

Not maintained in HW anymore. Simulation only, further undocumented.

# 4

# Reference designs

To demonstrate the capabilities of the IP toolbox, the following hardware and software modules are deployed:

## 4.1 Software: 'VisionKit framework'

The **VisionKit** framework is a high performance embedded solution for imaging solutions where frame loss needs to be under a very tight control or has to be avoided at all. It consists of a real time capable kernel driver and a user space video acquisition system – the **videoserver** – featured by a preprocessor FIFO. Every stage of the processing can be controlled using separately running threads such that the CPU power is fully used.
The videoserver is portable among various platforms (PC, embedded systems). The following reference hardware platforms and demos make use of the videoserver engine.

## 4.2 'gözcu' preprocessing camera

The gözcu camera system is equipped with a fast hybrid host/DSP processor and a preprocessing FPGA module based on a Spartan6 LX45. It is capable of processing data at very high speeds. Demosaicing and JPEG encoding are done on the FPGA side, the host processor is running a Linux OS to support networking and video serving over the portable videoserver software.

## 4.3 Lattice HDR60 eval kit

The HDR60 is a development and evaluation kit for the ECP3 and various pluggable sensor modules plus USB and Ethernet peripherals.
The current HDR60 test environment is featured by a JPEG encoder test bench to stream compressed image data over a isochronous USB connection to a host PC. The JPEG stream can be viewed using a standard browser at slow image rates (~8 fps) or via mplayer or VLC at high frame rates.
On the HDR60 demo environment, the videoserver is running on the host PC, using a specific USB FIFO backend for image acquisition. The MJPEG player connects to the videoserver locally or remotely over a TCP/IP connection to request video transmission.

| Property | Specification |
|---|---|
| Interface | USB2.0 |
| Typ. throughput | ~15 MB/s on Linux videoserver |
| Viewer software | Browser, mplayer, other |
| Sensor | MT9P031, TODO: MT9M024 |
| Tested Pixel clock | 54 MHz (theor. 80 MHz) |

*Table 4.1: Lattice HDR60 based reference design overview*

# 5 Functional and compliance testing

## 5.1 Simulation and debugging environment

A full cycle accurate simulation model exists for all variants of the encoder IP and the SoC including software debugger functionality using gdb. This means, the SoC simulation can be debugged 'live' while the system is processing data. To summarize all simulator capabilities:

1. Debugging using GDB
2. PNG source images can be streamed directly into the simulation using a front end
3. Output analysis of the encoded image using test bench or external tools (JPEGSnoop)

The simulation engine is also used to cover all test cases that are specific to the JPEG channel encoding and byte stuffing in order to guarantee an error free JPEG stream.

## 5.2 Empirical throughput tests

The JPEG compression ratio basically follows the principle that 'natural' transitions that are typical to a photographical image compress well. Completely randomized images with a statistically uniform frequency distribution again compress less well and apply quite some stress to the channel output encoder FIFOs.
The first empirical testing stage was carried out on the hardware test bench using the following approach:

- Generate badly compressable pattern sequences by the embedded $\mu$C
- Push MCUs through JPEG encoder using DMA
- Monitor error flags
- Verify output data using mplayer

These throughput tests are currently verified over several hours with test images of 800x600 @37 fps, which corresponds to a compressed JPEG data rate of maximum 18 MB/s at highest quality (95%)
The overall bottleneck in this case is the USB 2.0 interface which can go at a maximum of 24 MB/s in isochronous mode. Also note that system relevant dropouts (USB packets not picked up in time due to system load) can cause FIFO overflow at high data rates on the HDR60 side.

### 5.2.1 Data rate aspects

The maximum data rate (and compression factor) can is only determined by empirical analysis. It obviously depends on the source image characteristics, the subsampling factors and the quantization factor ('quality'). At the maximum tested quality of 95%, typical high detail generated images with YUV422 encoding do not exceed a factor of 1:2 (source:compressed). The detailed analysis of the data rate is omitted here, as it can be determined easily beforehand using free JPEG software encoders.

## 5.3 Theoretical entropy coding analysis

Finding the worst compressible data pattern plus giving the proof for its maximum entropy is out of the scope of this project for now. However, a statistical analysis could be carried out according to the statically stored default Huffman table. This will remain an option for later research when hard requirements or certifications come into play.

# A

# Appendix

## A.1 Known issues

Table A.1 shows known issues and limitations of the encoder IP.

| Short description | Version | Issue | Fix |
|---|---|---|---|
| L1 and L2 encoder do not support restart marker | xx01 | Feature | no |
| L1 encoder may emit several ZRL before EOL | xx01 | Feature | no |
| jpeg_l1 module requires minimum time for deasserted 'ce' | xx00 | Feature | yes |

*Table A.1: Issues*

**L1 and L2 encoder do not support restart marker**
Restart markers that allow to resume image decoding when JPEG data is missing are not supported by this version. There is no workaround. It is mandatory for this version that all data is transmitted.

**L1 encoder may emit several ZRL before EOL**
The compression may not be the full optimum under the occurence of a MCU containing little high frequencies, due to the RLE encoder not buffering the data for a 'look ahead to the end' feature. Therefore, a number of ZRLs can be emitted before the finaly EOB. The result is, that images with little detail have a slightly less optimum compression. Typically, this is barely noticeable.

**jpeg_l1 module requires minimum time for deasserted 'ce'**
The jpeg_l1 state machine requires a certain blank time when ce goes low in order to resynchronize. When feeding MCU blocks to the encoder, 'ce' must stay high over the entire period and, if it goes low, remain low for at least 16 cycles. This is fixed from xx01.

## A.2 Resource usage of L2SoC v1

The raw (non-optimized) resource usage of the L2SoC core is shown in the tables below.

| Resource/Timing | Number (%) |
|---|---|
| Register bits | 1900 (3%) |
| DSP units | 16 (12%) |
| Block RAMs (EBR) | 45 (19%) |
| Slices (LUTs) | 2918 (9%) |
| I/O | 71 (25%) |
| Pixel clock (tested) | 72 MHz |
| Maximum listed clock | 108 MHz |

*Table A.2: Usage/Timing on Lattice ECP3 LFE3_70EA*

## A.3 Links and further documentation

A list of documents and further pointers:


[devdesc] **The device description XML dialect**
04/2005, section5::ms <hackfin@section5.ch>
*A set of schema description files, part of the netpp distribution*
*URL: http://section5.ch/netpp*

[jpeg_t81] CCITT Recommendation T.81 from 1992
**Information Technology Digital Compression And Coding Of Continuous-Tone Still Images — Requirements And Guidelines**
Filename: `JPEG_itu-t81.pdf`

[jpegdoxy] JPEG encoder library package
*Doxygen generated developer reference*
**JPEG encoder developer documentation**

[players] JPEG players:
**mplayer** : *URL: http://www.mplayerhq.hu*
**VLC player** : *URL: http://www.videolan.org*

[subsampling] **Wikipedia: chroma subsampling**
*URL: http://en.wikipedia.org/wiki/Chroma_subsampling*