jpegenc Documentation

Release 0.0.1

Merkourious and Vikram

October 23, 2016

Contents

CHAPTER 1

Overview

The *jpegenc* package is a JPEG encoder implemented in MyHDL. The JPEG encoder is intended to be flexible with reusable subblocks. This project also includes a verification environment for the encoder.

The following figure outlines the main subblocks in the system.

Fig. 1.1: The JPEG encoder system diagram

The subblocks were designed to be independent and process an image stream.

1.1 Uses

- (M)JPEG real-time video compression.
- Framework for investigation image and video compression.

1.2 Goals

- Easy to use and understand JPEG encoder implementation.
- Flexible (modular) and reusable subblocks.
- Base set of blocks to build various image and video encoders.

1.3 Measurements

1.3.1 Coverage Results for Frontend Part Modules

Module	Coverage
Color Space Conversion	100
Color Space Conversion v2	89
1D-DCT	100
2D-DCT	100
Zig Zag Scan	100
Frontend Part	91

1.3.2 Coverage Results for Backend Modules

Module	Coverage
Quantizer	100
Quanizer_Core	100
Divider	100
RLE	100
RLE_Core	100
EntropyCoder	100
RLE_Doublebuffer	100
Huffman	99
Huffman_Doublebuffer	100
ByteStuffer	100
Backend	99

1.3.3 Hardware Implementation Results

The Frontend Part was converted to VHDL and synthesized using Vivado 2014.4. The stategy which used for the implementation run was the ExplorePostRoutePhysOpt. The above results are from the Post-Route report.

Utilization

Device	FF(%)	LUT(%)	MEMORY LUT(%)	I/O(%)	DSP48(%)
xc7kc325tffg900-2	0.41	0.43	0.01	7.8	9.05
xc7vx690tffg1761-2	0.19	0.20	0.01	4.59	2.11
xc7z020clg484-1	1.58	1.65	0.01	19.50	34.55
xc7a200tsbg484-2	0.41	0.43	0.01	7.8	9.05

Max Clock Frequency

Device	Frequency (MHz)
xc7kc325tffg900-2	322
xc7vx690tffg1761-2	320
xc7z020clg484-1	180
xc7a200tsbg484-2	221

Quick start

The easiest path to install the package is to use *pip*:

```
>> pip install git://github.com/cfelton/test_jpeg.git#egg=test_jpeg
```

Once the package is installed the test suite can be run:

```
>> cd tests
>> py.test
```

CHAPTER 3

JPEG encoder reference

3.1 Interfaces

Todo

This sections needs to be completed, the interfaces are in a state of change.

3.2 Frontend

3.2.1 Interfaces

Color Space Conversion Intefaces

Inputs Interface

class `jpegenc.subblocks.common.`**`RGB`**(*nbits=8*)
Red, Green, Blue Signals with nbits bitwidth for RGB input

Ouputs Interface

class `jpegenc.subblocks.common.`**`YCbCr`**(*nbits=8*)
Y, Cb, Cr output signals

class `jpegenc.subblocks.common.`**`YCbCr_v2`**(*nbits=8*)
Data_out Signal. According to color mode the data_out could be Y, Cb or Cr for color mode = 0, 1, 2

1D-DCT Intefaces

Inputs Interface

class `jpegenc.subblocks.common.`**`input_1d_1st_stage`**(*nbits=8*)
Input interface for the 1st stage 1D-DCT

Ouputs Interface

class jpegenc.subblocks.common.**output_interface**(*out_precision=10*, *N=8*)
Output interface for the 1D-DCT module

2D-DCT Intefaces

Inputs Interface

class jpegenc.subblocks.common.**input_interface**(*nbits=8*)
Input Interface for the 2D-DCT module

Ouputs Interface

class jpegenc.subblocks.common.**outputs_2d**(*precision_factor=10*, *N=8*)
Output interface for the 2D-DCT module. It is used also as input/output interface for the zig-zag scan module

Zig Zag Scan Intefaces

Inputs Interface

class jpegenc.subblocks.common.**outputs_2d**(*precision_factor=10*, *N=8*)
Output interface for the 2D-DCT module. It is used also as input/output interface for the zig-zag scan module

Ouputs Interface

class jpegenc.subblocks.common.**outputs_2d**(*precision_factor=10*, *N=8*)
Output interface for the 2D-DCT module. It is used also as input/output interface for the zig-zag scan module

Frontend Part Intefaces

Inputs Interface

class jpegenc.subblocks.common.**inputs_frontend_new**(*nbits=8*)
Red, Green, Blue Signals with nbits bitwidth for RGB input

Ouputs Interface

class jpegenc.subblocks.common.**outputs_frontend_new**(*precision_factor=10*)
Output signals of the frontend part

3.2.2 Subblocks

Color Space Conversion Module

Color Space Conversion Module

jpegenc.subblocks.color_converters.rgb2ycbcr.**rgb2ycbcr**

Color Space Conversion module

This module is used to transform the rgb input to an other representation called YCbCr. The input interface is the RGB and the output is the Ycbcr. The inputs and outputs are parallel.

Inputs: Red, Green, Blue, data_valid clock, reset

Outputs: Y, Cb ,Cr, data_valid

Parameters **num_fractional_bits** –

class jpegenc.subblocks.color_converters.rgb2ycbcr.**ColorSpace**(*red=0*, *green=0*, *blue=0*)

Color Space Conversion Class

It is used to derive the integer coefficients and as a software reference for the conversion

_set_jfif_coefs()

The YCbCr special constants

The JFIF YCbCr conversion requires "special" constants defined by the standard. The constants are describe in a Wikipedia page: https://en.wikipedia.org/wiki/YCbCr

get_jfif_ycbcr()

RGB to YCbCr Conversion. Used as software reference.

get_jfif_ycbcr_int_coef(*precision_factor=0*)

Generate the integer (fixed-point) coefficients

Color Space Conversion Module v2

Color Space Conversion Module v2

jpegenc.subblocks.color_converters.rgb2ycbcr_v2.**rgb2ycbcr_v2**

Color Space Conversion Module v2

This module is used to transform the rgb input to an other representation called YCbCr. The input interface is the RGB and the output is the Ycbcr_v2. The inputs and outputs are serial. According to the signal color_mode one of the 3 transformations occurs. If color_mode equals to 0 then the rgb to Y transformation occurs and so on.

Inputs: Red, Green, Blue, data_valid, color_mode, clock, reset

Outputs: data_out, data_valid

Parameters **num_fractional_bits** –

class jpegenc.subblocks.color_converters.rgb2ycbcr.**ColorSpace**(*red=0*, *green=0*, *blue=0*)

Color Space Conversion Class

It is used to derive the integer coefficients and as a software reference for the conversion

_set_jfif_coefs()

The YCbCr special constants

The JFIF YCbCr conversion requires "special" constants defined by the standard. The constants are describe in a Wikipedia page: https://en.wikipedia.org/wiki/YCbCr

get_jfif_ycbcr()

RGB to YCbCr Conversion. Used as software reference.

get_jfif_ycbcr_int_coef(*precision_factor=0*)
Generate the integer (fixed-point) coefficients

1D DCT Module

jpegenc.subblocks.dct.dct_1d.**dct_1d**
1D-DCT Module

This module performs the 1D-DCT Transformation. It takes serially N inputs and outputs parallely the vector of N signals. The parameter num_fractional_bits defines how many bits will be used for the fixed point representation of the dct coefficient. The out_precision parameter defines how many bits will be used for the fixed point representation of the outputs signals. This module is the building block for the 2d-dct module. The input interface is the input_1d_1st_stage and the output interface is the output_interface.

Inputs: data_in, data_valid, clock, reset

Outputs: List of N signals: out_sigs, data_valid

Parameters out_precision, N (*num_fractional_bits,*) –

jpegenc.subblocks.dct.dct_1d.**tuple_construct**(*matrix*)
Construct a tuple from list to use it as a rom

class jpegenc.subblocks.dct.dct_1d.**dct_1d_transformation**(*N*)
1D-DCT Transformation Class

It is used to derive the integer coefficient matrix and as a software reference for the 1D-DCT Transformation.

build_matrix(*N*)
Create the coefficient NxN matrix

dct_1d_transformation(*vector*)
1D-DCT software reference

dct_int_coeffs(*precision_factor*)
Transform coeff matrix to integer coefficients

2D DCT Module

jpegenc.subblocks.dct.dct_2d.**dct_2d**
2D-DCT Module

This module performs the 2D-DCT Transformation. It takes serially the inputs of a NxN block and outputs parallely the transformed block in 64 signals. The row-column decomposition method used to implement the 2d-dct. The parameter num_fractional_bits is used to define the fractional bits for the fixed point representation of the coefficients. The stage_1_prec parameter defines the fractional bits for the fixed point representation of the 1st stage 1d-dct outputs. The out_prec defines the fractional bits for the ouputs of the 2d-dct and the N parameter defines the size of the NxN block.

Inputs: data_in, data_valid, clock, reset

Outputs: NxN signals in the list out_sigs, data_valid

Parameters stage_1_prec, out_prec, N (*num_fractional_bits,*) –

class jpegenc.subblocks.dct.dct_2d.**dct_2d_transformation**(*N*)
2D-DCT Transformation Class

It is used as a software reference for the 2D-DCT Transformation

build_matrix (*N*)
Create the NxN coefficient matrix

dct_2d_transformation (*block*)
2D-DCT software reference

Zig Zag Scan Module

jpegenc.subblocks.zig_zag.zig_zag.**zig_zag**
Zig-Zag Module

This module performs the zig-zag reorderding. According to the zig-zag matrix the input list of signals is reordered. The inputs and the ouputs are parallel. The parameter N defines the size of the NxN block.

Inputs: List of signals of a NxN block

Outputs: Reordered list of signals of a NxN block

Parameters = the size of the NxN block (*N*) –

class jpegenc.subblocks.zig_zag.**zig_zag_scan** (*N*)

static build_zig_zag_matrix (*N*)
Build the zig-zag matrix Code taken from http://paddy3118.blogspot.gr/2008/08/zig-zag.html

zig_zag (*signal_list*)
Zig-zag scan function

Frontend Part Module

jpegenc.subblocks.frontend.frontend_v2.**frontend_top_level_v2**
Frontend Part of the JPEG Encoder

This part combines the color space conversion, 2D-DCT and zig-zag scan modules. It takes serially each input pixel (Red, Green, Blue) and when it computes the first block it takes another two times the same block in order to compute the other components. First outputs the Y block, then the Cb block and last the Cr block. The processing of this part is continuous and it never stops.

Inputs: red, green, blue, data_valid, clock, reset

Outputs: data_out, data_valid

class jpegenc.subblocks.frontend.frontend_v2.**frontend_transform**
Software implementation of the frontend part

Quantizer

divider module

This module contains the HDL for divider used for Quantiser

jpegenc.subblocks.quantizer.divider.**divider**
This module contains the HDL implementation

jpegenc.subblocks.quantizer.divider.**divider_ref** (*dividend, divisor*)
software implementation of divider

quant_rom module

MyHDL implementation of Quantiser ROM

jpegenc.subblocks.quantizer.quant_rom.**build_huffman_rom_tables**(*csvfile*)
build huffman tables

jpegenc.subblocks.quantizer.quant_rom.**quant_rom**
Build Chrominance ROM for Huffman Tables

quantizer module

The above module is the hardware implementation of quantizer top module

class jpegenc.subblocks.quantizer.quantizer.**QuantCtrl**
Bases: object

Control Signals used for quantizer top module

start : signal used to start the processing of block ready : asserts when block is ready to take next input color_components : select Y1 or Y2 or Cb or Cr component

class jpegenc.subblocks.quantizer.quantizer.**QuantIODataStream**(*width_data=12*, *width_addr=6*)
Bases: object

Input datastream into the Quantizer top module

data_in : send input data into the module read_addr : read the data from the input buffer

jpegenc.subblocks.quantizer.quantizer.**quantizer**
The Quantizer module divides the input data and data in the ROM

Arguments: quanti_datastream : Input datastream to the module quant_ctrl : control signals to the module

Returns: quanto_datastream : Output datastream from the module

quantizer_core module

The above module is the hardware implementation of quantizer core module

class jpegenc.subblocks.quantizer.quantizer_core.**QuantDataStream**(*width_data=12*)
Bases: object

Input interface for core module

data : input data to the quantizer core module valid : asserts when input data is valid

jpegenc.subblocks.quantizer.quantizer_core.**quantizer_core**
This Module is the core of the Quantizer

Arguments: quant_input_stream : Input stream to the core module color_component : used to select specific quantizer tables

Returns: quant_output_stream : Output data stream from the Quantizer

ramz module

This ram is used to store quantization values

`jpegenc.subblocks.quantizer.ramz.`**`ramz`**
default addr width 6, data width 12

romr module

This module generates reciprocals for numbers 0-255

`jpegenc.subblocks.quantizer.romr.`**`romr`**
Reciprocals of numbers are generated for quantizer core

RLE Module

doublebuffer module

The above module is a double buffer to store runlength encoded data

`jpegenc.subblocks.rle.doublebuffer.`**`doublefifo`**
I/O ports:

dfifo_bus : A FIFOBus connection interace buffer_sel : select a buffer

Constants :

depth : depth of the fifo used width_data : width of the data to be stored in FIFO

entropycoder module

This module takes a input and returns amplitude of the input and number of bits required to store the input.

`jpegenc.subblocks.rle.entropycoder.`**`bit_length`**(*num*, *maxlen=32*)
Determine the number of bits required to represent a value This functions provides the same functionality as the Python int.bit_length() function but is convertible.

This function generates the combinatorial logic to determine the maximum number of bits required to represent an unsigned value.

Currently the function computes a maximum of maxlen bits.

for values larger than 2**maxlen this function will fail. myhdl convertible

`jpegenc.subblocks.rle.entropycoder.`**`entropy_encode`**(*amplitude*)
Model of the entropy encoding

Parameters **`amplitude`** (`int`) – given an integer generate the encoding

Returns size_ref:

Return type amplitude_ref

`jpegenc.subblocks.rle.entropycoder.`**`entropycoder`**
This module return the amplitude and number of bits required to store input

io ports:

data_in : input data into the entropy coder size : number of bits required to store amplitude amplitude : amplitude of the input

constants:

width_data : width of the input data

`jpegenc.subblocks.rle.entropycoder.`**`two2bin`**(*num*)
converts negative number to positive

rle module

This module is the MyHDL implementation of run length encoder top module

class `jpegenc.subblocks.rle.rle.`**`BufferDataBus`**(*width_data*, *width_size*, *width_runlength*)
Bases: *`jpegenc.subblocks.rle.rlecore.RLESymbols`*

Connections related to output data buffer

Amplitude : amplitude of the number size : size required to store amplitude runlength : number of zeros dovalid : asserts if ouput data is valid buffer_sel : select the buffer in double buffer read_enable : read data from the output fifo fifo_empty : asserts if any of the two fifos are empty

`jpegenc.subblocks.rle.rle.`**`rlencoder`**
The top module connects rle core and rle double buffer

I/O Ports:

datastream : input datastream bus buffer data bus : output data bus rleconfig : configuration bus

Constants:

width_data : input data width width_addr : address width width_size : width of register to store amplitude size max_addr_cnt : maximum address of the block being processed width_runlength : width of runlength value that can be stored limit : value of maximum runlength value width_depth : width of the FIFO Bus

rlecore module

This module if the core of the run length encoder module

class `jpegenc.subblocks.rle.rlecore.`**`Component`**
Bases: `object`

Select the color component

class `jpegenc.subblocks.rle.rlecore.`**`DataStream`**(*width_data=12*, *width_addr=6*)
Bases: `object`

Input data streams into Rle Core

data_in : input to the rle module read_addr : address of input data from the input ram

class `jpegenc.subblocks.rle.rlecore.`**`RLEConfig`**
Bases: `object`

RLE configuration Signals are the generic signals used in the block

color_component [select the color component] to be processed(Y1, Y2, Cb or Cr)

start : start signal triggers the module to start processing data sof : start of frame asserts when next frame is ready

class `jpegenc.subblocks.rle.rlecore.`**`RLESymbols`**(*width_data=12*, *width_size=6*, *width_runlength=4*)

Bases: `object`

Output symbols generatred by RLE Core

Amplitude : amplitude of the number size : size required to store amplitude runlength : number of zeros dovalid : asserts if ouput is valid

`jpegenc.subblocks.rle.rlecore.`**`rle`**

This is the RLE Core module

IO Ports:

datastream : input data and address to the input bus rlesymbols : output generated by core module rleconfig : configuration ports for rle core

constants:

width_data : input data width width_addr : address width width_size : width of register to store amplitude max_addr_cnt : maximum address of the block being processed width_runlength : width of runlength value that can be stored limit : value of maximum runlength value

`jpegenc.subblocks.rle.rlecore.`**`sub`**(*num1*, *num2*)

subtractor for Difference Encoder

Huffman Module

Submodules

ac_cr_rom module

MyHDL implementation of AC Chrominance ROM

`jpegenc.subblocks.huffman.ac_cr_rom.`**`ac_cr_rom`**

build ac ROM for chrominance

ac_rom module

MyHDL implementaton of Luminance AC ROM

`jpegenc.subblocks.huffman.ac_rom.`**`ac_rom`**

Build AC ROM here

dc_cr_rom module

MyHDL implementation of Chrominance DC ROM

`jpegenc.subblocks.huffman.dc_cr_rom.`**`dc_cr_rom`**

Build Chrominance ROM for Huffman Tables

dc_rom module

MyHDL implementation of DC ROM used for Huffman Encoder

`jpegenc.subblocks.huffman.dc_rom.`**`dc_rom`**
build dc rom here

doublebuffer module

The above module is a double buffer to store huffman encoded data

`jpegenc.subblocks.huffman.doublebuffer.`**`doublefifo`**
I/O ports:

dfifo_bus : A FIFOBus connection interace buffer_sel : select a buffer

Constants :

depth : depth of the fifo used width_data : width of the data to be stored in FIFO

huffman module

MyHDL implementation of Huffman Encoder Module

class `jpegenc.subblocks.huffman.huffman.`**`HuffBufferDataBus`**(*width_packed_byte*)
Bases: `object`

Output Interface of the Huffman module read_req : access to read the output data stored in FIFO fifo_empty : output fifo is empty buffer_sel : select a buffer from Double Fifo huf_packed_byte : Huffman Encoded Output

class `jpegenc.subblocks.huffman.huffman.`**`HuffmanCntrl`**
Bases: `object`

These are the control signals for Huffman block start : start sending block ready : request for next block color_component : select the component to be processed sof : start of frame

class `jpegenc.subblocks.huffman.huffman.`**`HuffmanDataStream`**(*width_runlength*, *width_size*, *width_amplitude*, *width_addr*)
Bases: `object`

Input interface bus to the Huffman module runlength : runlength of the data byte vli_size : number of bits required to store vli vli : amplitude of the data data_valid : input data is valid

class `jpegenc.subblocks.huffman.huffman.`**`ImgSize`**(*width=8*, *height=8*)
Bases: `object`

Indicates dimensions of the Image width : width of the image height : height of the image

class `jpegenc.subblocks.huffman.huffman.`**`VLControl`**
Bases: `object`

Contains the four states in which the FSM Operates

`jpegenc.subblocks.huffman.huffman.`**`huffman`**
HDL Implementation of Huffman Module. This module takes Variable Length Encoded Inputs and serialise them to VLC using Huffman Rom Tables

Args: huffmancntrl : control signals interface huffmandatastream : Input Interface img_size : Image data class rle_fifo_empty : asserts when Input buffer is empty

Returns: bufferdatabus : Output FIFO Interface

Constants: block_size : size of each block vlcontrol : contains the states used to run huff_fsm image_size.width : width of image image_size.heigth : height of image bits_block_count : width to store number of blocks in image width_word : maximum width of the word register

tablebuilder module

Used to build Huffman Tables

`jpegenc.subblocks.huffman.tablebuilder.`**`build_huffman_rom_tables`**`(`*csvfile*`)`
build huffman tables

ByteStuffer Module

bytestuffer module

This module is MyHDL implementation of Byte Stuffer used for JPEG Encoder

class `jpegenc.subblocks.bytestuffer.bytestuffer.`**`BSInputDataStream`**`(`*width_data*`)`
Bases: `object`

Input interface for the Byte Stuffer

data_in : Input data to Byte Stuffer read : read signal sent to input FIFO fifo_empty : asserts if input FIFO is empty

class `jpegenc.subblocks.bytestuffer.bytestuffer.`**`BSOutputDataStream`**`(`*width_data*, *width_addr_out*`)`
Bases: `object`

Output Interface for the Byte Stuffer

byte : output byte from the Byte Stuffer addr : output address to the RAM data_valid : asserts when output data is valid

class `jpegenc.subblocks.bytestuffer.bytestuffer.`**`BScntrl`**
Bases: `object`

Control Interface for Byte Stuffer

sof : start of frame start : send input frame when start asserts ready : ready to access next frame

`jpegenc.subblocks.bytestuffer.bytestuffer.`**`bytestuffer`**
Byte stuffer checks for 0xFF byte and adds a 0xFF00 Byte

Constants:

width_addr_out : maximum adress width of the output RAM width_out : width of the data in the ouput RAM

I/O Ports :

bs_in_stream : input interface to the byte stuffer bs_cntrl : control interface to the byte stuffer bs_out_stream : output interface to the byte stuffer num_enc_byte : number of bytes encoded to output RAM

Backend Module

backend module

MyHDL implementation of Backend Module

`jpegenc.subblocks.backend.backend.`**`backend`**

Constants:

width_data : width of the input data width_addr : width of the address accessed by a module width_runlength : width of the runlength value width_size : width of the size value width_out_byte : width of output byte width_num_bytes : max encoded bytes width

backend_soft module

software prototype for backend module

`jpegenc.subblocks.backend.backend_soft.`**`backend_ref`**(*block*, *prev_dc_0*, *prev_dc_1*, *prev_dc_2*, *register*, *color_component*, *pointer*)

backend reference module

`jpegenc.subblocks.backend.backend_soft.`**`build_huffman_rom_tables`**(*csvfile*)

build huffman tables

`jpegenc.subblocks.backend.backend_soft.`**`build_rom_tables`**(*csvfile*)

build huffman tables

`jpegenc.subblocks.backend.backend_soft.`**`bytestuffer`**(*block*)

bytestuffer reference module

`jpegenc.subblocks.backend.backend_soft.`**`divider`**(*block*, *color_component*)

divider reference module

`jpegenc.subblocks.backend.backend_soft.`**`divider_ref`**(*dividend*, *divisor*)

software implementation of divider

`jpegenc.subblocks.backend.backend_soft.`**`entropy_encode`**(*amplitude*)

Model of the entropy encoding

Parameters **`amplitude`** (*`int`*) – given an integer generate the encoding

Returns size_ref:

Return type amplitude_ref

`jpegenc.subblocks.backend.backend_soft.`**`huffman_final`**(*register*, *pointer*)

divide huffman code into bytes

`jpegenc.subblocks.backend.backend_soft.`**`huffman_ref`**(*runlength_block*, *amplitude_block*, *size_block*, *color_component*, *register*, *pointer*)

reference model for huffman encoder

`jpegenc.subblocks.backend.backend_soft.`**`runlength`**(*block*, *color_component*, *prev_dc_0*, *prev_dc_1*, *prev_dc_2*)

reference for runlength encoder module

`jpegenc.subblocks.backend.backend_soft.`**`table_huff_gen`**(*filename*, *base*)

huffman table generator

dualram module

`jpegenc.subblocks.backend.dualram.`**`dram`**

default addr width 6, data width 12

3.2.3 Test units

Color Space Conversion Test

Color Space Conversion v2 Test

1D-DCT Test

2D-DCT Test

Zig Zag Scan Test

Frontend Part Test

Quantizer Test

Quantizer-Top Test

Quantizer_core Test

Divider Test

RLE Test

RLE-Top Test

RLE_core Test

Entropy Coder Test

RLE Double Buffer Test

Huffman Test

Huffman Test

Huffman Double Buffer Test

Bytestuffer Test

Backend Test

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Symbols

A

B

C

D

S

T

V

Y

Z