OL_QOIE
QOI Imge Compressor Core

User's Manual Rev 1.0
Core version 1.0

Overview

This core implements the QOI lossless image compression algorithm producing a raw, header-less file. Simple, fully synchronous design with low gate count.

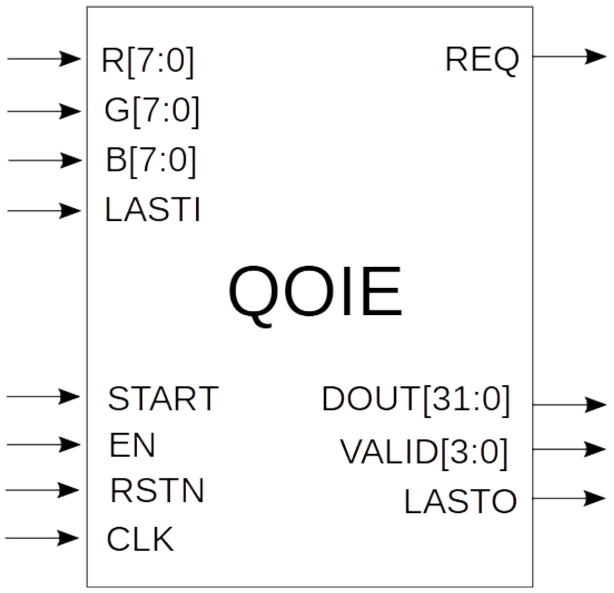
Applications

- ♦ Bandwidth and storage reduction.
- ♦ Lossless compressed frame store.
- ♦ Space imaging applications.
- ♦ SoC to SoC image transmission.

Features

- ♦ Implemented according to the QOI image format.
- ♦ Processes one 24 bits pixel per clock cycle.
- ♦ High throughput : up to 800 Mpixles/s in high end, 4K@30 in low end FPGAs.
- ♦ Optional header processing available.
- ♦ Fully synchronous design.
- ♦ Available as fully functional and synthesizable VHDL or Verilog soft-core.
- ♦ Test benches provided.

Symbol



The QOI Lossless Image Compression Algorithm

QOI stands for Quite OK Image Format and it is a fast, simple lossless image compression algorithm.

As it can be seen in these benchmarks, the performance of the algorithm is usually somewhere in between two well known image compression libraries (like libpng and stb) but at a much lower computational cost.

QOI uses a combination of some well known image compression techniques and innovative ideas.
Pixels are compressed as:

- runs of identical pixels
- an index to a 64 pixel cache of previously encountered pixels
- a difference to the previous pixel
- a full RGB pixel

The QOI algorithm is an excellent compromise between compression performance and low algorithmic complexity.

QOI Lossless Image Compression Core

OL_QOIE is a fast, low complexity implementation of the compression algorithm that accepts one 24 bits RGB pixel per clock cycle and outputs compressed data packed in 32 bits words.

Pin Description

Name	Type	Description
RSTN	Input	Core asynchronous reset, active low.
CLK	Input	Core clock signal.
EN	Input	Synchronous enable signal. When LOW operations stall and the core ignores all its inputs.
START	Input	When HIGH, compression operation is started.
R[7:0]	Input	Red component of the input pixel.
G[7:0]	Input	Green component of the input pixel.
B[7:0]	Input	Blue component of the input pixel.
LASTI	Input	When HIGH, it signals the last input pixel.
REQ	Output	When HIGH, a new pixel is requested.
DOUT[31:0]	Output	Output data.
VALID[3:0]	Output	Output data valid. Each bit, when HIGH, it indicates the valid byte in the output data.
LASTO	Output	When HIGH, it indicates the last compressed data.

Functional description

Rising the START input for one clock cycle activates the core. After 62 cycles, used by the core to initialize the pixel cache, the core will raise the REQ output, accepting one RGB pixel form then on, without any gaps.

If a pixels cannot be provided when the core requests it, then the EN signal must be lowered. This will stall the core as long as EN is LOW. Likewise for the output, if not ready to accept compressed data being output.

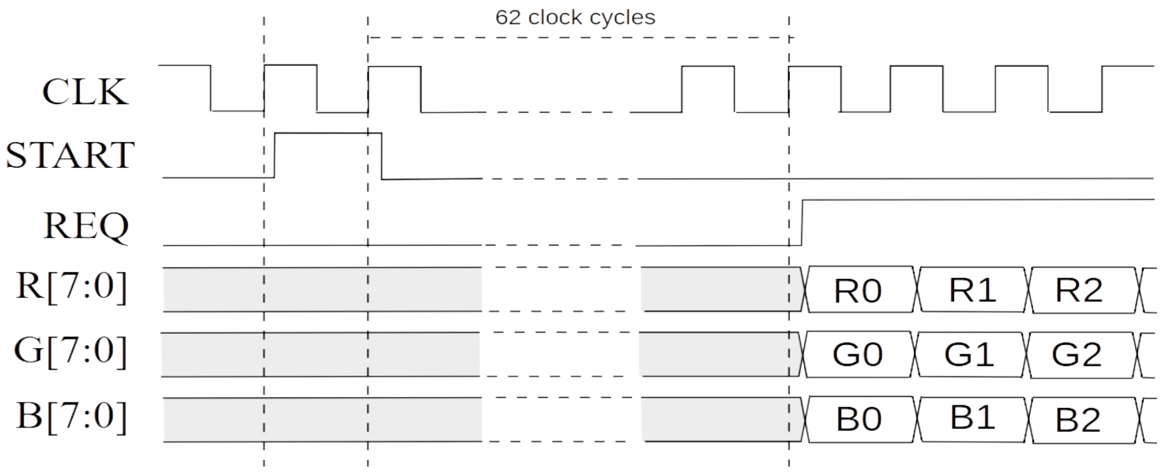


Figure 1: Start core operations.

When limited to RGB, the QOI algorithm can output from 1 to 4 bytes for each input pixel. Therefore, in order to guarantee accepting one pixel input per clock cycle, the core will pack the compressed data in 32 bit words. The core will output a 32 bit word from DOUT[31:0] as soon as at least 4 bytes are accumulated internally. This will be indicated by raising the VALID[3:0] output.

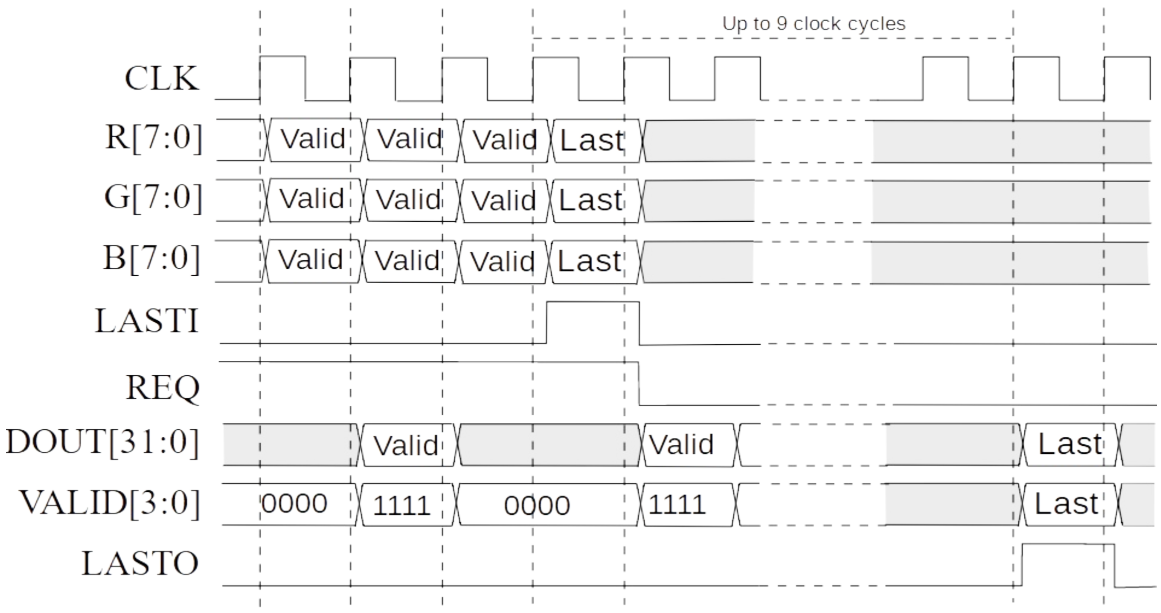


Figure 2: Last pixels input and last compressed data output.

The last pixel fed to the core must be indicated by raising the LASTI signal. As shown above, as soon as the last pixel has been clocked in the REQ signal will be lowered.
Depending on the amount of compressed data that needs to be flushed out, it can take up to 9 cycles for the last 32 bit word to be output. This will be indicated by the LASTO output going HIGH. For the last output data, each bit of the VALID[3:0] output will also indicate which byte of

the DOUT[31:0] is actually valid. This is because the QOI format is byte aligned and the last output might not necessarily contain 4 bytes. Specifically, bit 3 of VALID[3:0] will indicate that bits 31:24 of DOUT[31:0] are valid, bit 2 that bits 23:16 are valid and so on.

C Model and test vectors generation

A C model of the core is provided for test vector generation. The C model provided is based on a the reference implementation, located on GitHub and included in the deliverable in the **qoi-master/** directory.

The test vector generators, **qoie.c** is also provided.
In order to compile this program, make sure that the **qoi.h** file from the **qoi-master/** directoryis visible by the c compiler.
The test vector generator **qoie.c** is compiled with :

gcc –o qoie image-io.c qoie.c

The resulting **qoie** executable usage is:

./qoie <input image in PPM format> <output image in QOI format>

Example :
./qoie image.ppm image.qoi

The resulting image.qoi file is a raw QOI (without header) binary file.
Two other utilities are also provided **bin2hex32.c** and **ppm2hex.c** .
They can be compiled with:
gcc -o bin2hex32 bin2hex32.c
gcc -o ppm2hex ppm2hex.c

bin2hex32.c is used to convert the binary QOI file into a hex text file suitable for the testbench.
Example :
./bin2hex image.qoi qoi.dat

Likewise, **ppm2hex** is used to convert the PPM image file into a hex text file suitable for the testbench.
Example :
./ppm2hex image.ppm pixel.dat

The hex text files qoi.dat and pixel.dat can then be moved to the testbench directory to be used by the simulation.

It is possible to also use **qoie** to generate a QOI binary file that includes the QOI header, suitable for viewing by using the -h option.
Eample :
./qoie image.ppm image.qoi -h

In the Windows environment the free Irfanview can be used to view QOI files. The -h option should not be used to generate files for the testbench as this version of the core does not generate an header.

Test bench

The directory **tb** contains the HDL file **tb.v(hd)**. This files represents the self-checking test benches provided with the core.
The figure below shows a block diagram of the test bench.

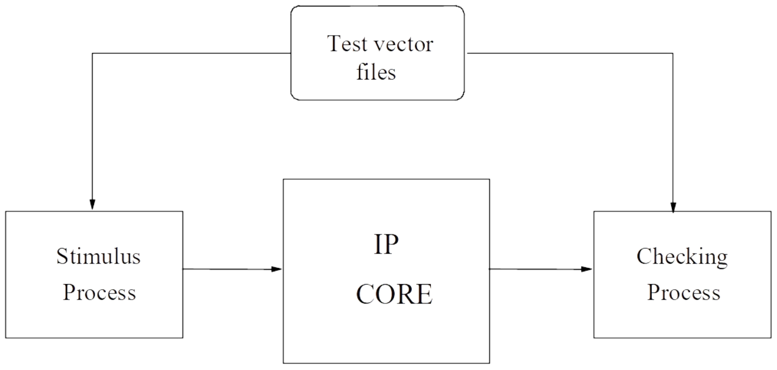


Figure 3 Test bench block diagram.

The Stimulus Process reads the input vectors and passes them to the core. The core results are verified by the Checking Process.

The testbench reads the pixels form the pixel.dat file and feeds them to the OL_QOIE core. It also verifies that the output matches the qoi.dat file.