Center for Embedded and Cyber-physical Systems
University of California, Irvine

A SystemC Model of a PNG Encoder

Vivek Govindasamy and Rainer Dömer

Technical Report CECS-TR 20-02

December 18, 2020

Center for Embedded and Cyber-physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

+1 (949) 824-8919

A SystemC Model of a PNG Encoder

Vivek Govindasamy and Rainer Dömer

Technical Report CECS-TR 20-02

December 18, 2020

Center for Embedded and Cyber-physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

+1 (949) 824-8919

Abstract

Portable Network Graphics (PNG) is the most frequently used lossless compression standard for images. Due to PNG's widespread use, most cameras have inbuilt ASIC to perform PNG encoding on images. Most images are automatically compressed using PNG encoders and transmitted. We present here a SystemC model to get a better understanding of its working and intended structure.

In this technical report, we demonstrate a parallelized SystemC model of a PNG encoder capable of achieving high quality lossless compression with small file sizes. We first discuss the original specification of the PNG encoder application and then present its SystemC model, followed by initial experimental results that demonstrate its functionality.

Contents

A SystemC Model of a PNG Encoder

Vivek Govindasamy and Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

Abstract

Portable Network Graphics (PNG) is the most frequently used lossless compression standard for images. Due to PNG's widespread use, most cameras have inbuilt ASIC to perform PNG encoding on images. Most images are automatically compressed using PNG encoders and transmitted. We present here a SystemC model to get a better understanding of its working and intended structure.

In this technical report, we demonstrate a parallelized SystemC model of a PNG encoder capable of achieving high quality lossless compression with small file sizes. We first discuss the original specification of the PNG encoder application and then present its SystemC model, followed by initial experimental results that demonstrate its functionality.

1 Introduction

PNG images [1] are ubiquitously used to store screenshots and other pixel-based images [2]. However, the interior working is not known as widely as other image compression formats like JPEG [3]. PNG encoders have two main working components which perform the actual compression, the filters and the *DEFLATE* algorithm. Filters are of five types, and they are mainly used to reduce pixel values so that they occupy less space. The reduced pixel values require fewer number of bits to transmit, providing some compression. Filtering also creates *runs* of data, and they are compressed in a similar way as run length encoding.

The filtered values are sent in as inputs to the *DEFLATE* algorithm, which uses a combination of Lempel–Ziv–Storer–Szymanski (LZSS) and Huffman encoding to perform lossless compression. *DEFLATE* works better on values which are highly correlated to each other, which will be explained later in this report.

PNG image data is then stored as chunks such as Image Header (IHDR), Image Data (IDAT), Image Palette (PLTE), Image Time (tIME), Image End (IEND) and so on. Here we will only be focusing on IHDR, IDAT and IEND, as these are called critical chunks. The other chunks are known as ancillary chunks and are only required in specific situations.

Our SystemC model is structured as a parallel filtering architecture, followed by a comparator to choose the optimal filtered result. Filters operate row-wise and each row is sent to *DEFLATE* serially. SystemC modules then create chunks to be written to the final PNG file. A simplified PNG encoder is shown in Figure 1. The pixels are filtered and then restructured as a 1 dimensional array. The change in color is due to filtering and the subsequent reduction in pixel intensities. The first element of the array represents the filtering method used. Every row is joined as one large array and compressed to obtain the PNG data. Chunks are then created to provide information to the decoder. The chunks are not shown in the figure.

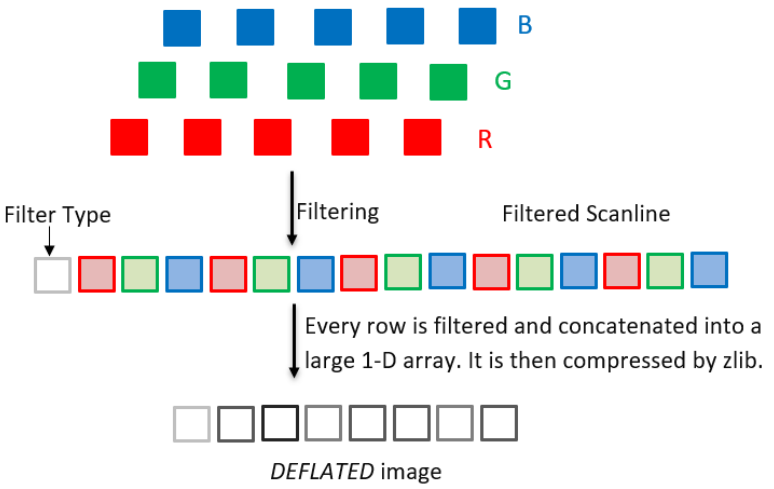

Figure 1: The general PNG encoder model.

[4] provides more information on encoders, and multiple encoder implementations in a variety of programming languages.

2 The PNG Algorithm

We describe the technical details of filtering here. The filters operate on every row individually [5]. Filters are of five different types, namely *None*, *Sub*, *Up*, *Average* and *Paeth*. Figure 2 shows the filtering arithmetic used.

Type	Name	Filter function	Reconstruction Function
0	*None*	Filt(x) = Orig(x)	Recon(x) = Filt(x)
1	*Sub*	Filt(x) = Orig(x) – Orig(a)	Recon(x) = Filt(x) + Recon(a)
2	*Up*	Filt(x) = Orig(x) – Orig(b)	Recon(x) = Filt(x) + Recon(b)
3	*Average*	Filt(x) = Orig(x) – floor((Orig(a) + Orig(b))/2)	Recon(x) = Filt(x) + floor((Recon(a) + Recon(b)/2)
4	*Paeth*	Filt(x) = Orig(x) – PaethPredictor (Orig(a), Orig(b), Orig(c))	Recon(x) = Filt(x) + PaethPredictor(Recon(a), Recon(b), Recon(c))

Figure 2: The filter types and arithmetic. x,a,b and c are shown in Figure 3

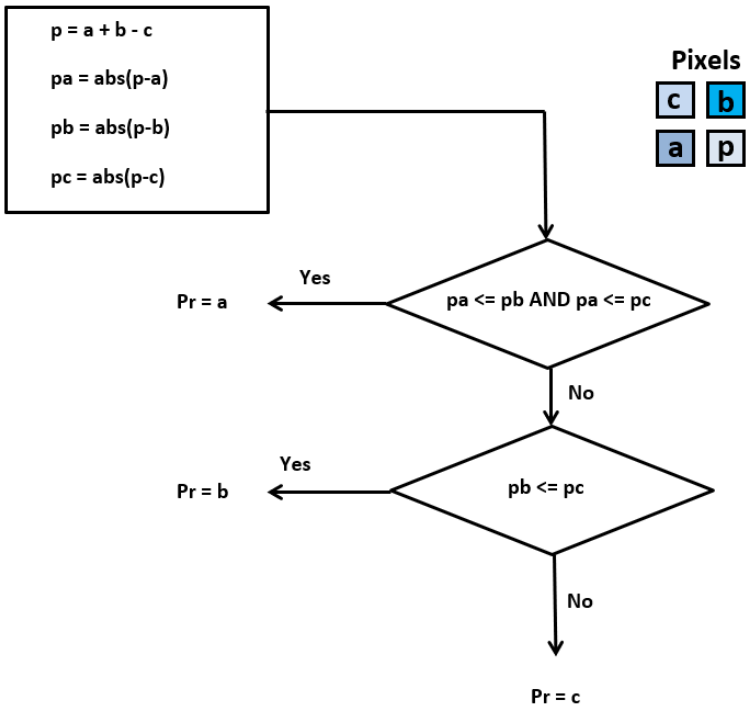

Figure 3: The *Paeth* filter, the most critical PNG filter. The x pixel on the bottom left is to be predicted by the *Paeth* filter

The filters find similarities between adjacent pixels. This helps predict the next pixel and it is a form of delta encoding. The '*Sub*' filter correlates with the left pixel value, the '*Up*' filter correlates with the previous row and the *Paeth* filter correlates with the diagonal values. The detailed algorithm is shown in Figure 3. Pr is the PaethPredicted output which is needed compute the *Paeth* filtered value in Figure 2. The *Paeth* filter is not as straight forward as the other filters. However experimental results show that it is often better than other filters when there is a high degree of variance among pixels in the current row.

The *mean* filter is not implemented in our SystemC model as it has the possibility of generating floating point values, and doing so would require additional ancillary chunks to convey such information. The four implemented filters operate in parallel. The filtered outputs are almost always stored as 8 bit characters (so that they fit in a byte), unless specified otherwise in ancillary chunks. If the filtered value is negative, 256 is added to it. The decision to choose the filtered output is dependent on the encoder and the choice is not clear from the references. We have followed reference [5] which recommends computing the absolute sum of the filtered row, and selecting the row with the least sum. We call this method **filter method 1**.[6] suggests using the '*Sub*' filter for the first row and every row after the first uses the *Paeth* filter. This method will be called as **filter method 2**.

While viewing the hexadecimal file of compressed PNG images, we observed that images which were generated by computer tools used the **first method** of filter selection. These images used every filter available. However, images which were captured with a camera used the **second method**. A possible explanation for this is that even though camera captured images may have regions which seem to be of the same color (like an image of the sky), they are actually pixels of different intensities and the Paeth filter works best for them.

3 The *DEFLATE* algorithm

DEFLATE is an algorithm originally developed in 1996 by Phil Katz [7]. Using a combination of LZSS [8] and Huffman Coding [9], it compresses data losslessly. It is widely used, and a common example is the ZIP file format in computers. While many different implementations of the *DEFLATE* algorithm exist (i.e. pkzip, zlib, gzip), it is important to note that PNG uses only zlib [10]. The implementations have different headers, and zlib, which starts with 78, is the only recognizable format by PNG decoders.

We provide brief descriptions of the two main coding algorithms used in *DEFLATE*.

3.1 LZSS Dictionary Coding

Dictionary coding is a lossless compression technique where matches are searched for in a data set. The data set can be predefined or generated. LZSS creates a new dictionary entry every time it encounters a new character or string. When the same string is repeated further in the array, it is substituted with the distance from the starting point and the length of the string [8]. Here is an example. If we have a string [I am Sam Sam I am], it is transformed to [I am Sam (5,3) (0,4)]. The strings 'I am' and 'Sam' have not been encountered before, so they are stored in the dictionary, and as they are repeated we can directly substitute numbers for them which saves characters.

DEFLATE will always store dictionary strings as hash tables. *DEFLATE* also uses a sliding window of 2^{15}=32768 characters. This means that it will retain up to 32768 possible hash tables, and characters which are behind the current character by 32768 positions will be discarded. This is done so that *DEFLATE* does not need to search the entire dictionary for each string, which significantly saves compression time.

3.2 Huffman Coding

Huffman coding is a lossless prefix coding technique, which tries to encode frequently repeated characters with the least number of bits [9]. Prefix coding ensures that there exists no other code segment with the same initial segment as another code word. It employs a greedy strategy and generates a near optimal code, which is unique. Huffman coding generates binary characters only.

Here is a simple example to demonstrate Huffman coding. Consider an output from the LZSS encoder as (5,3) with a 60% chance of appearing, (0,4) with 30% chance and (0,1) with a 10% chance. We would encode (5,3) as 0, (0,4) as 10 and (0,1) as 11. No two code segments have the same prefix here. If an output stream is 0001110, the only possible decoded output is (5,3), (5,3), (5,3), (0,1), (0,4). The code segments and their literal values are transmitted along with the Huffman stream.

There is one important factor in zlib's Huffman coding. This is called the number of bytes for a chunk. It roughly translates to the byte size of a Huffman coded stream with unique code segments. The default value is 16834 bytes. After this value is exceeded, new Huffman codes are generated. This value can be changed to powers of 2, and higher values increase compression speed. This chunk should not be confused with the PNG chunks in the following section, as they are completely different. *DEFLATE* has other technical details which can be found on the official page [11].

4 PNG Chunks

PNG chunks carry information about the image or the pixel data. In our SystemC model we use only critical chunks to keep the model simple. The following information can help users encode their images in the PNG format. Figure 4 shows how every chunk is stored.

Figure 4: The format for every chunk

4.1 The Image Header Chunk (IHDR)

The IHDR is the chunk which contains most of the important specifications about the image. It is exactly 13 bytes long. To convey its length, the '0D' hex value is always present before IHDR. Table 1 shows the specifications we are using.

Table 1: Table for the Header Chunk

Specification	**Number of Bytes**
Width	4 bytes
Height	4 bytes
Bit depth	1 byte
Colour type	1 byte
Compression method	1 byte
Filter method	1 byte
Interlace method	1 byte

The bit depth is the number of bits per pixel color. We are using 8, so for RGB it will be 24 bits per pixel. Color type conveys the color scheme being used. We are using Truecolor in our model. Using palette can save additional space, providing the same image quality in some cases. Compression method will always be 0, which means that the data was compressed using *DEFLATE*. Filter method will also always be 0, which implies that the

standard five filters were applied. We are using no interlace so it is set to 0. Interlace is useful when we need to render some rows or columns before the others, but it isn't required in our model. A CRC check is present at the end with a length of 4 bytes.

4.2 The Image Data Chunk (IDAT)

The IDAT chunk contains the image data. Before IDAT is present, its length will be specified in a hexadecimal value. In our model this value is C5EC02 which implies that the IDAT chunk is 12971010 bytes long. The IDAT contains the data compressed with DEFLATE, along with a zlib wrapper. The first two bytes of IDAT are always 78, which specifies zlib. The end of IDAT contains a CRC which is automatically computed by zlib. The next section provides a detailed diagram of the hexadecimal file of `fungus.png` to provide users with an example.

4.3 The Image End Chunk (IEND)

IEND is the final chunk in any PNG image. It always has the following values- **49 45 4E 44 AE 42 60 82**. The first four values translate to the numerical characters IEND when converted to ASCII. The data field is empty. The last four values are the CRC check values.

5 The SystemC Model

Our SystemC model is written for Accellera SystemC version 2.3.3. Figure 5 shows the structure of the model. The stimulus module reads a bitmap image (`.bmp`) of size 3216x2136 called as `fungus.bmp`. The image is read into a one dimensional array which stores the pixel values in the BGR format (`fungus.bmp` stores pixel data in the order of blue, green and red). The pixel intensities are then stored in separate R, G and B arrays. Multiple ports and channels are created to send the pixel intensities to the four filter modules we are using. This increases the level of parallelism to 12 parallel filtering instances. Filtered values are then sent to the comparator module through channels. We use the filter selection function which chooses the best filtered output for *DEFLATE*. We create a buffer array called '**inputbuffer**', which stores the row wise filtered image. The total length of the array is 3217x2136, with the first pixel of every row being the filter method used as shown in the table.

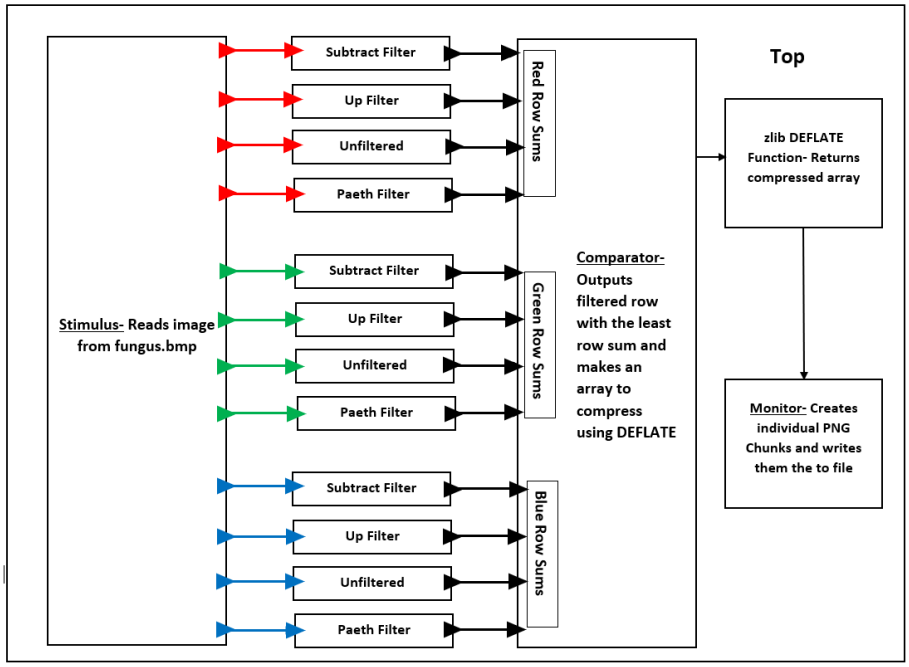

Figure 5: The SystemC block diagram of the modules and ports

Now the PNG header chunk (IHDR) is created and written to the output file. Figure 6 shows the IHDR chunk and the beginning of IDAT for the `fungus.png` image. To use other images of the same resolution the same header will work. Using different resolutions will require a modification of the image width and height hexadecimal values, and a re computation of the cyclic redundancy check (CRC) value.

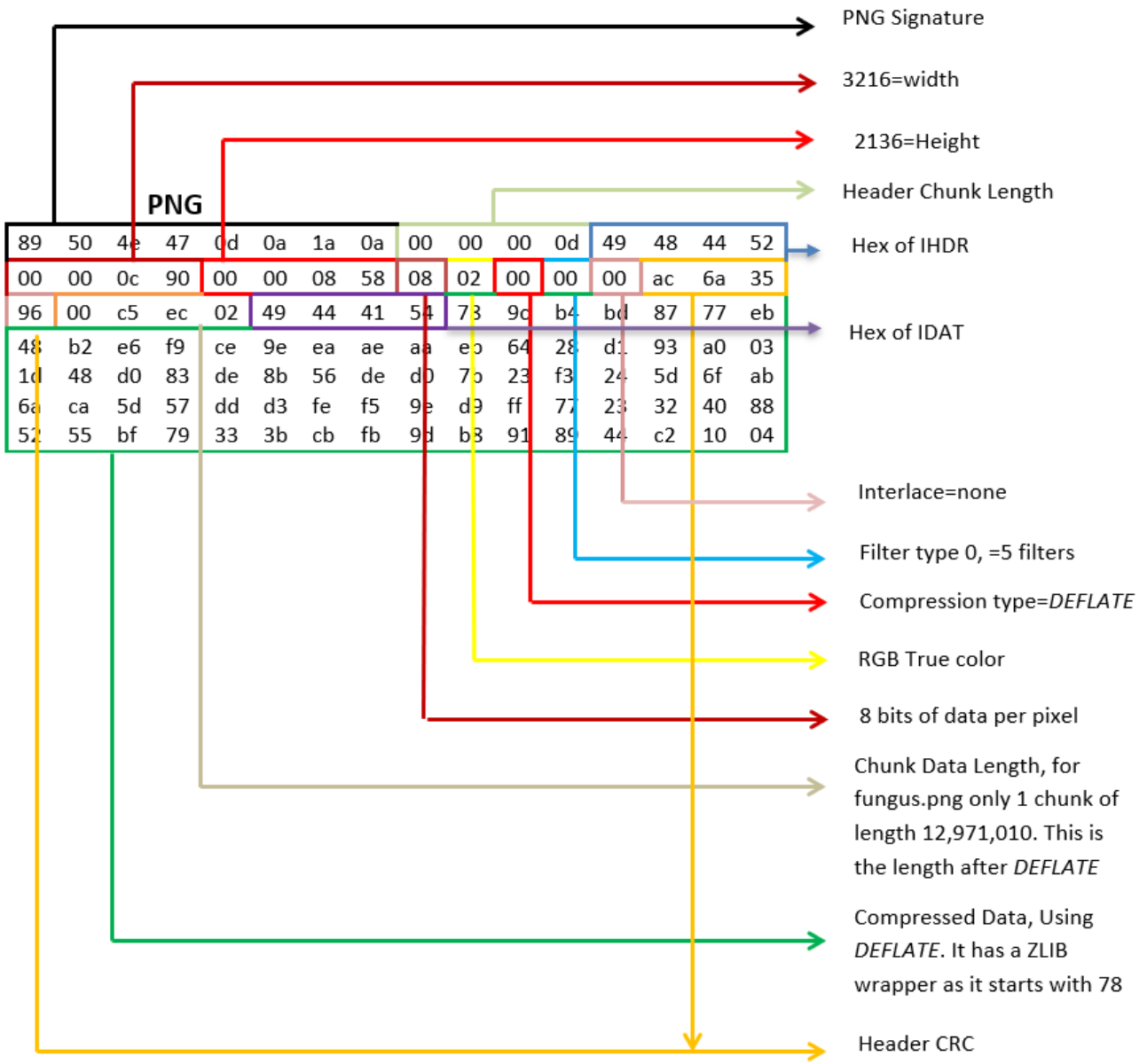

Figure 6: Header and Data chunk for the `fungus.png` image

The buffer array is compressed using the *DEFLATE* function with only a single IDAT chunk. It is also possible to split the IDAT chunk into smaller individual chunks, but this will increase the compressed size. Finally, an IEND chunk is written to the output file.

Table 2: Experimental results

PNG Encoder	Size of the Image in MB
SystemC model with no filters	13.0MB
SystemC model with filter selection method 1	11.2MB
SystemC model with filter selection method 2	8.3MB
Matlab	8.0MB
Microsoft Paint	13.2MB

5.1 Experimental Results

The PNG encoder model is available as GIT checkout from our internal server. To use it the zlib library must be installed. In the Makefile the only option that requires changing is the location of the SystemC installation directory.

Table 2 shows the experimental results for the `fungus.bmp` image. The original BMP image size is 27.5 MB. We are able to achieve a compressed size of only 40% of the original image when using **filter selection method 1**, which gives us a compression ratio of approximately 2.5. When using **filter selection method 2**, we can achieve a compressed size of 30% of the original image, which gives us a compression ratio of 3.3. Paeth filtering seems to be the best option for the `fungus.bmp` image. Matlab provides the best compression, but our **filtering method 2** provides high compression as well. Our SystemC encoder is also significantly more space efficient than the inbuilt Microsoft Paint encoder. Thus, our PNG compression works well and our SystemC model achieves good compression rates.

5.2 Known Limitations and Future Work

There exist a few limitations in this SystemC model. The first one is the inability to change the image size easily. Every array must be changed manually and the IHDR block will be completely changed. The second limitation is that not every BMP image can be easily read by the stimulus module. This is because BMP images have different options when they are encoded. The stimulus module may be further improved to fix this issue.

References

[1] Iso functional specification. https://www.iso.org/standard/29581.html.

[2] Android mobile phone camera specifications. https://developer.android.com/guide/topics/media/media-formats.

[3] W3 functional specification. https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[4] Offical libpng webpage. http://www.libpng.org/.

[5] Filter selection method 1, official w3 png webpage. https://www.w3.org/TR/2003/REC-PNG-20031110/#12Filter-selection.

[6] John Miano. *Compressed image file formats: Jpeg, png, gif, xbm, bmp*. Addison-Wesley Professional, 1999.

[7] Official pkzip webpage. https://www.pkware.com/pkzip.

[8] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[9] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[10] Official zlib webpage. https://zlib.net/.

[11] Official *DEFLATE* webpage. https://tools.ietf.org/html/rfc1951#section-Abstract.