

A Practical Parallel CRC Generation Method

By Evgeni Stavinov, OutputLogic.com



Introduction

Most electrical and computer engineers are familiar with the Cyclic Redundancy Check or CRC. They know that it's used in communication protocols to detect bit errors, and that it's essentially a remainder of the modulo-2 long division operation.

Many of the engineers had closer encounters with the CRC and know that it's implemented as a Linear Feedback Shift Register (LFSR) using Flip Flops and XOR gates. They've likely used an online tool or an existing example to generate parallel CRC code for their design.

But very few engineers understand the mechanics of the CRC well enough to build a customized parallel CRC circuit described by an arbitrary CRC generator polynomial.

This is the topic of this article. It presents a practical method of generating Verilog or VHDL code for the parallel CRC. This method allows fast generation of a parallel CRC code for an arbitrary polynomial and data width. The article also briefly describes some other interesting methods to provide a broader perspective on the subject.

So why writing an article on parallel CRC ? There are several existing tools that can generate the code, and a lot of examples for popular CRC polynomials. However, oftentimes it's beneficial to understand the underlying principles in order to implement a customized circuit or make optimizations to an existing one. In general, it's a useful knowledge to have in a toolbox of a practicing logic design engineer.

CRC Overview

Every modern communication protocol uses one or more error detection algorithms. Cyclic Redundancy Check, or CRC, is by far the most popular one. CRC properties are defined by the generator polynomial length and coefficients. The protocol specification usually defines CRC in hex or polynomial notation. For example, CRC5 used in USB protocol is represented as 0x5 in hex notation or as $G(x)=x^5+x^2+1$ in the polynomial notation:

$$\text{Hex notation } 0x5 \leftrightarrow \text{Polynomial notation } G(x) = x^5+x^2+1$$

This CRC is typically implemented in hardware as a Linear Feedback Shift Register (LFSR) with a serial data input as shown in the following figure.

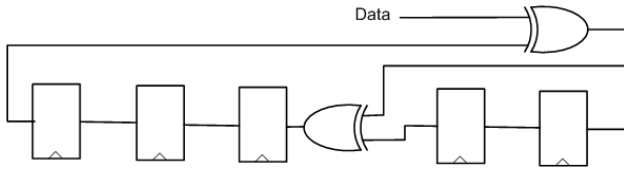


Figure 1: USB CRC5 implementation as LFSR using generator polynomial $G(x)=x^5+x^2+1$

In many cases serial LFSR implementation of the CRC is suboptimal for a given design. Because of the serial data input, it only allows the CRC calculation of one data bit every clock. If a design has an N-bit datapath, meaning that every clock CRC module has to calculate CRC on N bits of data, serial CRC will not work.

One example is USB 2.0, which transmits data at 480 MHz on the physical level. A typical USB PHY chip has 8 or 16-bit data interface to the chip that does protocol processing. A circuit that checks or generates CRC has to work at that speed.

Another, more esoteric application I've encountered with was calculating 64-bit CRC on a data written and read from 288-bit wide memory controller (two 64-bit DDR DIMMs with ECC bits).

To achieve higher throughput, serial LFSR implementation of the CRC has to be converted into a parallel N-bit wide circuit, where N is the design datapath width, so that every clock N bits are processed. This is a parallel CRC implementation, which is the subject of this article. A simplified block diagram of the parallel CRC is shown in Figure 2.

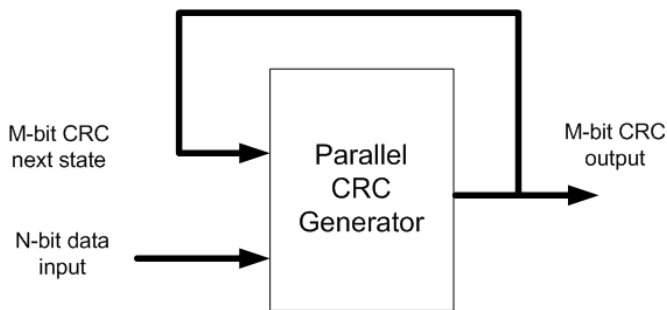


Figure 2: Parallel CRC block diagram. The next state CRC output is a function of the current state CRC and the data.

Even though CRC was invented almost half a century ago and has gained a widespread use, it still sparks a lot of interest in the research community. There is a constant stream of research papers and patents that offer different parallel CRC implementation with speed and logic area improvements.

Searching available literature and web resources on parallel CRC calculation methods for hardware description languages (HDL) produced a handful of papers, most notably most notably [\[1\]](#), [\[2\]](#), and [\[3\]](#). However, most found sources are academic and focus on the theoretical aspect of the parallel CRC generation. They are too impractical to implement in software or hardware for a quick HDL code generation of CRC with arbitrary data and polynomial widths.

An additional requirement on the method is that parallel CRC generator has to be able to accept any data width, not only power-of-2, to be useful. Going back to the USB 2.0 CRC5 example, a convenient data width to use for parallel CRC of polynomial width 5 is 11, since USB packets using CRC5 are 16 bit. Another example is 16-lane PCI Express with 128-bit datapath (16 8-bit symbols). Because the beginning of a packet is a K-code symbol and doesn't participate in the CRC calculation, the parallel CRC data is 120-bit wide.

Before getting further into parallel CRC discussion, I'd like to provide a brief overview of a modulo-2 polynomial arithmetic.

A polynomial is a value expressed in a form:

$$P(x) = \sum_{i=0}^N p(i)x^i = p(0) + p(1)x + \dots + p(N)x^N$$

where $p(i) \in \{0,1\}$

Polynomial addition and subtraction operations use bitwise XOR.

$$\text{Example: } P(x) = x^3 + x^2 + 1 \quad Q(x) = x^2 + x + 1 \quad P(x) + Q(x) = x^3 + x^2 + x$$

Polynomial multiplication by 2 is a left shift, and unsigned division by 2 is the right shift.

Modulo-2 polynomial division is realized the same way as long division over integers.

Cyclic left and right shifts are multiplication and division by $(2 \bmod 2^n - 1)$.

Parallel CRC Generation

I'll start off the discussion with providing a Verilog module that generates parallel USB CRC5 with 4-bit data.

```
//=====
// Verilog module that implements parallel USB CRC5 with 4-bit data
//=====
module crc5_parallel(
    input [3:0] data_in,
    output reg[4:0] crc5,
    input rst,
    input clk);

    // LFSR for USB CRC5
    function [4:0] crc5_serial;
        input [4:0] crc;
        input data;

        begin
            crc5_serial[0] = crc[4] ^ data;
            crc5_serial[1] = crc[0];
            crc5_serial[2] = crc[1] ^ crc[4] ^ data;
            crc5_serial[3] = crc[2];
            crc5_serial[4] = crc[3];
        end
    endfunction

    // 4 iterations of USB CRC5 LFSR
    function [4:0] crc_iteration;
        input [4:0] crc;
        input [3:0] data;
        integer i;

        begin
            crc_iteration = crc;

            for(i=0; i<4; i=i+1)
                crc_iteration = crc5_serial(crc_iteration, data[3-i]);
        end
    endfunction

    always @(posedge clk, posedge rst) begin
        if(rst) begin
            crc5 <= 5'h1F;
        end
        else begin
            crc5 <= crc_iteration(crc5,data_in);
        end
    end
endmodule
//=====
```

A synthesis tool will do its magic and produce some circuit depending on the target FPGA or ASIC technology. However, the purpose of this article is to explain how to get a parallel CRC circuit using XOR gates and flip-flops.

Next I'm going to describe a practical method that I've been using to generate parallel CRC in a number of projects. It works on any polynomial and data size, and independent on the target technology. Later on I'll mention other methods that have some useful properties.

The method description is step-by-step and is accompanied by an example of parallel CRC generation for the USB CRC5 polynomial $G(x)=x^5+x^2+1$ with 4-bit data width.

The method takes advantage of the theory published in [\[1\]](#) and [\[2\]](#), and leverages a simple serial CRC generator and linear properties of the CRC to build a parallel CRC circuit.

Step 1: Let's denote N =data width, M =CRC polynomial width. For parallel USB CRC5 with 4-bit datapath, $N=4$ and $M=5$.

Step 2: Implement a serial CRC generator routine for a given polynomial. It's a straightforward process and can be done using different programming language or scripts, for instance C, Java, Verilog or Perl. Verilog function *crc5_serial* shown below could be used for the serial USB CRC5.

```
//=====
// Verilog function that implements serial USB CRC5
//=====
function [4:0] crc5_serial;
    input [4:0] crc;
    input data;

    begin
        crc5_serial[0] = crc[4] ^ data;
        crc5_serial[1] = crc[0];
        crc5_serial[2] = crc[1] ^ crc[4] ^ data;
        crc5_serial[3] = crc[2];
        crc5_serial[4] = crc[3];
    end
endfunction
//=====
```

Denote this routine as CRC_{serial} .

Let's also build a routine $CRC_{parallel}(N_{in}, M_{in})$, which simply calls CRC_{serial} N times (the number of data bits) and returns M_{out} . The following pseudo-code is an example of $CRC_{parallel}$:

```
//=====
routine CRC_parallel(N_in, M_in)
  M_out = M_in
  for(i=0;i<N;i++)
    M_out = CRC_serial(N_in , M_out)
  return M_out
//=====
```

Step 3: Parallel CRC implementation is a function of N -bit data input and M -bit current CRC state, as shown in the Figure 2. We're going to build two matrices:

- a) Matrix H1. It describes M_{out} (next CRC state) as a function of N_{in} (input data) when $M_{in}=0$:

$$M_{out} = CRC_{parallel}(N_{in}, M_{in}=0)$$

H1 matrix is of size $[N \times M]$

- b) Matrix H2. It describes M_{out} (next CRC state) as a function of M_{in} (current CRC state) when $N_{in}=0$:

$$M_{out} = CRC_{parallel}(N_{in}=0, M_{in})$$

H2 matrix is of size $[M \times M]$

Step 4: Build the matrix H1. Using $CRC_{parallel}$ routine from **step 2** calculate CRC for the N values of N_{in} when $M_{in}=0$. The values are one-hot encoded. That is, each of the N_{in} values has only one bit set. For $N=4$ the values are $0 \times 1, 0 \times 2, 0 \times 4, 0 \times 8$ in hex representation.

Figure 3 shows matrix H1 values for USB CRC5 with $N=4$.

Min=0

	Mout[4]	Mout[3]	Mout[2]	Mout[1]	Mout[0]
Nin[0]	0	0	1	0	1
Nin[1]	0	1	0	1	0
Nin[2]	1	0	1	0	0
Nin[3]	0	1	1	0	1

Figure 3: Matrix H1 for USB CRC5 with $N=4$

Step 5: Build the matrix H2. Using CRC_{parallel} routine from **step 2** calculate CRC for the M values of M_{in} when N_{in}= 0. The values are one-hot encoded. For M=5 M_{in} values are 0x1,0x2,0x4,0x8,0x10 in hex representation.

Figure 4 shows the matrix H2 values for USB CRC5 with N=4

N_{in}=0

	Mout[4]	Mout[3]	Mout[2]	Mout[1]	Mout[0]
Min[0]	1	0	0	0	0
Min[1]	0	0	1	0	1
Min[2]	0	1	0	1	0
Min[3]	1	0	1	0	0
Min[4]	0	1	1	0	1

Figure 4: Matrix H2 for USB CRC5 with N=4

Step 6: Now we're ready to construct the parallel CRC equations. Each set bit *j* in column *i* of the matrix H1, and that's the critical part of the method, participates in the parallel CRC equation of the bit M_{out}[i] as N_{in}[j]. Likewise, each set bit *j* in column *i* of the matrix H2 participates in the parallel CRC equation of the bit M_{out}[i] as M_{in}[j].

All participating inputs M_{in}[j] and N_{in}[j] that form M_{out}[i] are XOR-ed together.

For USB CRC5 with N=4 the parallel CRC equations are as follows:

$$M_{out}[0] = M_{in}[1] \wedge M_{in}[4] \wedge N_{in}[0] \wedge N_{in}[3]$$

$$M_{out}[1] = M_{in}[2] \wedge N_{in}[1]$$

$$M_{out}[2] = M_{in}[1] \wedge M_{in}[3] \wedge M_{in}[4] \wedge N_{in}[0] \wedge N_{in}[2] \wedge N_{in}[3]$$

$$M_{out}[3] = M_{in}[2] \wedge M_{in}[4] \wedge N_{in}[1] \wedge N_{in}[3]$$

$$M_{out}[4] = M_{in}[0] \wedge M_{in}[3] \wedge N_{in}[2]$$

M_{out} is the parallel CRC implementation.

We used tables in Figures 3 and 4 to derive the equations.

The reason why this method works lies in the way we constructed matrices H1 and H2, where rows are linearly independent. We also used the fact that CRC is a linear operation:

$$\text{CRC}(A + B) = \text{CRC}(A) + \text{CRC}(B)$$

The resulting Verilog module that generates parallel USB CRC5 with 4-bit data is as follows:

```
//=====
// Verilog module that implements parallel USB CRC5 with 4-bit data using XOR gates
//=====
module crc5_4bit(
  input [3:0] data_in,
  output [4:0] crc_out,
  input rst,
  input clk);

  reg [4:0] lfsr_q,lfsr_c;
  assign crc_out = lfsr_q;

  always @(*) begin
    lfsr_c[0] = lfsr_q[1] ^ lfsr_q[4] ^ data_in[0] ^ data_in[3];
    lfsr_c[1] = lfsr_q[2] ^ data_in[1];
    lfsr_c[2] = lfsr_q[1] ^ lfsr_q[3] ^ lfsr_q[4] ^ data_in[0] ^ data_in[2] ^ data_in[3];
    lfsr_c[3] = lfsr_q[2] ^ lfsr_q[4] ^ data_in[1] ^ data_in[3];
    lfsr_c[4] = lfsr_q[0] ^ lfsr_q[3] ^ data_in[2];
  end // always

  always @(posedge clk, posedge rst) begin
    if(rst) begin
      lfsr_q <= 5'h1F;
    end
    else begin
      lfsr_q <= lfsr_c;
    end
  end // always

endmodule // crc5_4
//=====
```

Other Parallel CRC Generation Methods

There are many other methods for parallel CRC generation. Each method offers unique advantages, but has drawbacks as well. Some are more suitable for high-speed designs where logic area is less of an issue. Others offer the most compact designs, but for lower speed. As almost everything else in engineering, it's a matter of tradeoff to select the best one for a given design.

I'll mention the most notable methods.

The method described in [1] derives a recursive formula for parallel CRC directly from a serial implementation.

The idea is to represent an LFSR for serial CRC as a discrete-time linear system:

$$X(i+1) = FX(i) + U(i) \quad [\text{eq1}]$$

Where vector $X(i)$ is the current LFSR output, $X(i+1)$ is the output in the next clock, vector $U(i)$ is the i -th of the input sequence, F is a matrix chosen according to the equations of serial LFSR.

For example, USB CRC5 $G(x)=x^5+x^2+1$ will produce the following:

$$F = \begin{bmatrix} p(4) & 1 & 0 & 0 & 0 \\ p(3) & 0 & 1 & 0 & 0 \\ p(2) & 0 & 0 & 1 & 0 \\ p(1) & 0 & 0 & 0 & 1 \\ p(0) & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Where $p(i)$ are polynomial coefficients.

Addition and multiplication operations are bitwise logic XOR and AND respectively.

After m clocks the state is $X(i+m)$, and the solution can be obtained recursively from [eq1]

$$X(i+m) = F^m X(i) + F^{m-1} U(i) + \dots + FX(i+m) + U(i+m) \quad [\text{eq2}]$$

m is the desired data width.

Each row k of $X(i+m)$ solution is a parallel CRC equation of bit k .

An important result of this method is that it establishes a formal proof of solution existence. It's not immediately obvious that it's possible to derive parallel CRC circuit from a serial one.

The method described in [5] uses 2-stage CRC calculation. The idea is that checking and generating CRC is done not with generator polynomial $G(x)$, but with another polynomial $M(x) = G(x) * P(x)$. $M(x)$ is chosen such as it has fewer terms than $G(x)$ to simplify the complexity of the circuit that realizes the division. The result of the division by $M(x)$, which has a fixed length, is divided again by $G(x)$ to get the CRC.

The method described in [4] allows calculating CRC with byte enable. In many cases it's important. For example, if the data width is 16-bit but a packet ends on an 8-bit boundary it'd require having two separate CRC modules: for 8 and 16-bit. The byte enable method allows reusing 16-bit CRC circuit to calculate 8-bit CRC.

The method described in [2] uses DSP unfolding technique to build a parallel CRC. The idea is to model an LFSR as a digital filter and use graph-based unfolding to unroll loops and obtain the parallel processing.

Other methods include using Lookup Tables (LUTs) with pre-computed CRC values [\[3\]](#).

Performance Results

Logic utilization and timing performance of a parallel CRC circuit largely depends on the underlying target FPGA or ASIC technology, data width, and the polynomial width. For instance, Verilog or VHDL code will be synthesized differently for Xilinx Virtex5 and Virtex4 FPGA families, because of the difference in underlying LUT input size. Virtex5 has 5-bit LUTs whereas Virtex4 has 4-bit.

In general, logic utilization of a parallel CRC circuit will grow linearly with the data width. Using the big-O notation, logic size complexity is $O(n)$, where n is the data width. For example, each of the CRC5 5 output bits is a function of 4 data input bits:

$$CRC_{out}[i] = Function(CRC_{in}[4:0], Data[3:0]).$$

Doubling the data width to 8 bit will double the number of participating data bits in each CRC5 bit equation. That will make the total CRC circuit size up to $5 \times 2 = 10$ times bigger. Of course, not all bits will double – that depends on the polynomial. But the point is that the circuit size will grow linearly.

Logic utilization will grow as a second power of the polynomial width, or $O(n^2)$.

Doubling the polynomial width in CRC5 from 5 to 10 (let's call it CRC10, which has different properties), will double the size of each CRC10 output bit. Since the number of CRC outputs is also doubled, the total size increase is up to $2^2 = 4$ times.

Timing performance of the circuit will decrease because it will require more combinational logic levels to synthesize CRC output logic given the wider data and polynomial inputs.

I used free Xilinx WebPACK tools to simulate and synthesize parallel CRC circuits for USB CRC5 and popular Ethernet CRC32. You can explore the results in the accompanied Verilog code and project files.

Target FPGA is Xilinx Virtex5 LX30.

1. USB CRC5 with 4-bit data using “for loop” Verilog implementation

Number of LUTs	5
Number of FFs	5
Number of Slices*	2

2. USB CRC5 with 4-bit data using “XOR” Verilog implementation

Number of LUTs	5
Number of FFs	5
Number of Slices	2

3. CRC32 with 32-bit data using “for loop” Verilog implementation

Number of LUTs	32
Number of FFs	214
Number of Slices	93

4. CRC32 with 32-bit data using “XOR” Verilog implementation

Number of LUTs	32
Number of FFs	161
Number of Slices	71

Notes:

- A single Xilinx Virtex5 Slice contains 4 FFs and 4 LUTs

As expected, the number of FFs is 5 and 32 for CRC5 and CRC32. For small CRC5 circuit there is no difference in the logic utilization. However, for larger CRC32 the code using “XOR” method produces more compact logic than the “for loop” approach.

Note that those synthesis results should be taken with a grain of salt. The results are specific to the targeted technology (ASIC or FPGA family) and synthesis tool settings.

Conclusion

The parallel CRC generation method described in the article leverages a simple serial CRC generator and linear properties of the CRC to build $H1_{N \times M}$ and $H2_{M \times M}$ matrices.

Row [i] of the H1 matrix is the CRC value of N_{in} with a single bit [i] set, while $M_{in}=0$.

Row [i] of the H2 matrix is the CRC value of M_{in} with a single bit [i] set, while $N_{in}=0$.

Column [j] of H1 and H2 matrices contains the polynomial coefficients of the CRC output bit [j].

This method has been successfully used by the author in several communication and test-and-measurement projects. There is an online Parallel CRC Generator tool at OutputLogic.com that takes advantage of this method to produce Verilog or VHDL code given an arbitrary data and polynomial width.

A similar method is also used in generating parallel scramblers. Perhaps it could be a topic of another article.

About the Author

Evgeni Stavinov has over 10 years of diverse design experience in the areas of FPGA logic design, embedded software and networking. He worked for CATC, LeCroy, and SerialTek designing test and measurement tools for USB, Wireless USB, PCI Express, Bluetooth, SAS, and SATA protocols. He is currently a system engineer with Xilinx. Evgeni holds MSEE from USC - University of Southern California and BSEE from Technion – Israel Institute of Technology. For more information contact evgeni@outputlogic.com

About OutputLogic.com

OutputLogic.com is a web portal that offers online tools for FPGA and ASIC designers.

References

- [1] G. Campobello, G Patane, M Russo, "[Parallel CRC Realization](http://ai.unime.it/~gp/publications/full/tccrc.pdf)" (<http://ai.unime.it/~gp/publications/full/tccrc.pdf>)
- [2] G. Albertango and R. Sisto, "Parallel CRC Generation", *IEEE Micro*, Vol. 10, No. 5, October 1990, pp. 63-71.
- [3] A. Perez, "Byte-wise CRC Calculations", *IEEE Micro*, Vol. 3, No. 3, June 1983, pp. 40-50
- [4] Adrian Simionescu, Nobug Consulting , <http://space.ednchina.com/upload/2008/8/27/5300b83c-43ea-459b-ad5c-4dc377310024.pdf>
- [5] R. J. Glaise, "A two-step computation of cyclic redundancy code CRC-32 for ATM networks", *IBM Journal of Research and Development* Volume 41 , Issue 6 (November 1997) pp 705 - 710
- [6] [CRC on Wikipedia](#)

Accompanied Code:

C code for parallel CRC generation on OpenCores.org
Verilog code, Xilinx project and simulation files for parallel CRC5 and CRC32 implementations on OutputLogic.com.