by Evgeni Stavinov

A Practical Parallel CRC Generation Method

Do you understand the mechanics of the cyclic redundancy check (CRC) well enough to build a customized parallel CRC circuit described by an arbitrary CRC generator polynomial? This article covers a practical method of generating Verilog or VHDL code for the parallel CRC. The result is the fast generation of a parallel CRC code for an arbitrary polynomial and data width.

Most electrical and computer engineers are familiar with the cyclic redundancy check (CRC). Many 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. Some have 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 likely used an online tool or an existing example to generate parallel CRC code for a 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. What about you?

In this article, I'll present a practical method for generating Verilog or VHDL code for the parallel CRC. This method allows for the fast generation of a parallel CRC code for an arbitrary polynomial and data width. I'll also briefly describe other interesting methods and provide more information on the subject.

So why am I covering parallel CRC? There are several existing tools that can generate the code, and a lot of examples for popular CRC polynomials. However, it's often beneficial to understand the underlying principles in order to implement a customized circuit or make optimizations to an existing one. This is a subject every practicing logic design engineer should understand.

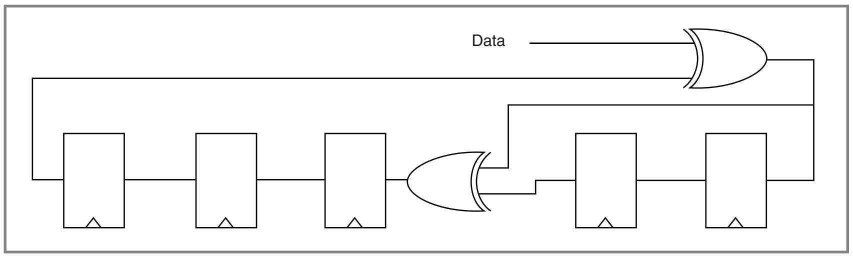

Figure 1—This is a USB CRC5 implementation as LFSR using generator polynomial $G(x) = x^5 + x^2 + 1$.

CRC OVERVIEW

Every modern communication protocol uses one or more error-detection algorithms. CRC is by far the most popular. CRC properties are defined by the generator polynomial length and coefficients. The protocol specification usually defines CRC in hex or polynomial notation. For

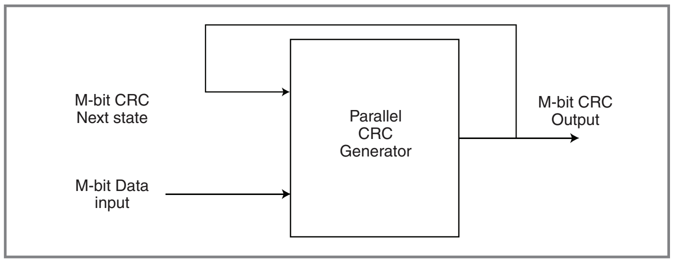

Figure 2—This is a parallel CRC block. The next state CRC output is a function of the current state CRC and the data.

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 (see Figure 1).

In many cases the 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 an 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 has to do with calculating 64-bit CRC on data written and read from a 288-bit-wide memory controller (two 64-bit DDR DIMMs with ECC bits). To achieve higher throughput, the CRC's serial LFSR implementation must be converted into a parallel N-bit-wide circuit, where N is the design datapath width, so that N bits are processed in every clock. This is a parallel CRC implementation, which is the subject of this article. Figure 2 is a simplified block diagram of the parallel CRC.

Even though the CRC was invented almost half a century ago and has gained 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. I was searching available literature and web resources about parallel CRC calculation methods for hardware description languages (HDL) and found a handful of papers. (Refer to the Resources section at the end of this article.) However, most were academic and focused on the theoretical aspect of the parallel CRC generation. They were 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 for the method is that the parallel CRC generator must be able to accept any data width (not only power-of-2) to be useful. Going back to the USB 2.0 CRC5

Listing 1—This Verilog module implements 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
     I       f(rst) begin
             crc5 <= 5'h1F;
         end
         else begin
             crc5 <= crc_iteration(crc5,data_in);
         end
     end
endmodule
//=============================================================================
```

example, a convenient data width to use for the parallel CRC of polynomial width 5 is 11 because USB packets using CRC5 are 16 bits. Another example is the 16-lane PCI Express with a 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 bits wide.

Before going any further into the topic of parallel CRC, I'll briefly review modulo-2 polynomial arithmetic. A polynomial is a value expressed in the following form:

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

where p(i) = {0,1}.

Polynomial addition and subtraction operations use bitwise XOR. Here is an example:

$$P(x) = x^3 + x^2 + 1$$
$$Q(x) = x^2 + x + 1$$
$$P(x) + Q(x) = x^3 + x^2 + x$$

Polynomial multiplication by two is a left shift, and unsigned division by two 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 mod $2^n - 1$).

Listing 2—This Verilog function implements 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
//=============================================================
```

Listing 3—This pseudocode 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 $M_{out}$

//==============================================================
```

PARALLEL CRC GENERATION

I'll start the discussion with a Verilog module that generates parallel USB CRC5 with 4-bit data (see Listing 1). A synthesis tool will do its magic and produce a 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'll describe a practical method that I use to generate parallel CRC in a number of projects. It works on any polynomial and data size, independent of the target technology. Later I'll present other methods that have some useful properties.

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

Table 1—This is the matrix H1 for USB CRC5 with N = 4.

bit data width. The method—which takes advantage of the theory described in a paper by Guiseppe Campobello et al titled "Parallel CRC Realization," as well as in a paper by G. Albertango and R. Sisto titled "Parallel CRC Generation"—leverages a simple serial CRC generator and the linear properties of the CRC to build a parallel CRC circuit.

In Step 1, denote N = data width and M = CRC polynomial width. For parallel USB CRC5 with a 4-bit datapath, N = 4 and M = 5.

In Step 2, implement a serial CRC generator routine for a given polynomial. It's a straightforward process and can be done using different programming languages or scripts (e.g., C, Java, Verilog, or Perl). You can use the Verilog function `crc5_serial` in Listing 2 for the serial USB CRC5. Denote this routine as CRC_{SERIAL}. You can also build a routine `CRCparallel(Nin, Min)` that simply calls CRC_{SERIAL} N times (the number of data bits) and returns M_{OUT}. The pseudocode in Listing 3 is an example of $CRC_{PARALLEL}$.

In 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. Matrix H1 describes M_{OUT} (next CRC state) as a function of N_{IN} (input data) when M_{IN} = 0. Thus, M_{OUT} = $CRC_{PARALLEL}$ (N_{IN}, M_{IN} = 0), and H1 matrix is the size [NxM]. Matrix H2 describes M_{OUT} (next CRC state) as a function of M_{IN} (current CRC state) when N_{IN} = 0. Thus, M_{OUT} = $CRC_{PARALLEL}$ (N_{IN} = 0, M_{IN}), and H2 matrix is the size [MxM].

In Step 4, build the matrix H1. Using the $CRC_{PARALLEL}$ routine from step 2, calculate the 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 0x1, 0x2, 0x4, 0x8 in hex representation. Table 1 shows matrix H1 values for USB CRC5 with N = 4.

In Step 5, build the matrix H2. Using the $CRC_{PARALLEL}$ 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. Table 2 shows the matrix H2 values for USB CRC5 with N = 4.

In Step 6, you'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 XORed together. For USB CRC5 with N = 4, the parallel CRC equations are as follows:

$$M_{OUT}[0] = M_{IN}[1] \wedge M_{IN}[4] \wedge M_{IN}[0] \wedge M_{IN}[3]$$
$$M_{OUT}[1] = M_{IN}[2] \wedge N_{IN}[1]$$
$$M_{OUT}[2] = M_{IN}[1] \wedge M_{IN}[3] \wedge M_{IN}[4] \wedge N_{IN}[0] \wedge N_{IN}[2] \wedge N_{IN}[3]$$
$$M_{OUT}[3] = M_{IN}[2] \wedge M_{IN}[4] \wedge N_{IN}[1] \wedge N_{IN}[3]$$
$$M_{OUT}[4] = M_{IN}[0] \wedge M_{IN}[3] \wedge N_{IN}[2]$$

M_{OUT} is the parallel CRC implementation. I used Table 1 and Table 2 to derive the equations.

The reason this method works is 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:

$$CRC(A + B) = CRC(A) + CRC(B)$$

The resulting Verilog module generates parallel USB CRC5 with 4-bit data (see Listing 4).

OTHER METHODS

There are many other methods for parallel CRC generation. Each method has advantages and drawbacks. 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 with almost everything else in engineering, you have to make trade-offs to bring your designs to completion.

Let's review the most notable methods. One method 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:

$$X(i + 1) = FX(i) + U(i)$$

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

Nin = 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

Table 2—This is the matrix H2 for USB CRC5 with N = 4.

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

Figure 3—This is matrix F in a formula $X(i + 1) = FX(i) + U(i)$ for recursive parallel CRC method. The values are for USB CRC5 polynomial $G(x) = x^5 + x^2 + 1$.

to the equations of serial LFSR. For example, USB CRC5 $G(x) = x^5 + x^2 + 1$ will produce Figure 3, 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.

$$X(i+m) = F^m X(i) + F^{m-1} U(i) + \ldots + FX(i+m) + U(i+m)$$

m is the desired data width. Each row k of the $X(i + m)$ 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 a parallel CRC circuit from a serial one.

Another method uses two-stage CRC calculation. The idea is that checking and generating CRC is done not with generator polynomial G(x), but with another polynomial $M(x) = G(x) \times P(x)$. M(x) is chosen so that 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.

Calculating the CRC with "byte enable" is another method that is important in many cases. For example, if the data width is 16 bits but a packet ends on an 8-bit boundary, it would require having two separate CRC modules for 8 and 16 bits. The byte enable method allows for the reuse of the 16-bit CRC circuit to calculate an 8-bit CRC.

There is also a 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 look-up tables (LUTs) with precomputed CRC values.

PERFORMANCE RESULTS

The logic use and timing performance of a parallel CRC circuit largely depends on the underlying target FPGA or ASIC technology, data width, and polynomial width. For instance, Verilog or VHDL code will be synthesized differently for the Xilinx Virtex5 and Virtex4 FPGA families because of the differences in the underlying LUT input sizes. Virtex5 has 6-bit LUTs, whereas the Virtex4 has 4-bit LUTs.

In general, the 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's five output bits is a function of four data input bits:

CRCout[i] = Function(CRCin4:0], Data[3:0])

Doubling the data width to 8 bits doubles the number of participating data bits in each CRC5 bit equation. That will make the total CRC circuit size up to 10 times bigger (i.e., 5 × 2). 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—doubles the size of each CRC10 output bit. The number of CRC outputs is also doubled, so the total size increase is up to 4 times (i.e., 2^2). The circuit's timing performance

Listing 4—This is a Verilog module that implements parallel USB CRC5 with 4-bit data using XOR gates.

```
//=================================================================
// 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
//=================================================================
```

a)	
Number of LUTs	5
Number of FFs	5
Number of Slices	2
b)	
Number of LUTs	5
Number of FFs	5
Number of Slices	2
c)	
Number of LUTs	214
Number of FFs	32
Number of Slices	93
d)	
Number of LUTs	161
Number of FFs	32
Number of Slices	71

Table 3a—Logic utilization for USB CRC5,4-bit data using the "for loop" method. b—Logic utilization for USB CRC5, 4-bit data the using "XOR" method. c—Logic utilization for CRC32, 32-bit data using the "for loop" method. d—Logic utilization for CRC32, 32-bit data using the "XOR" method.

decreases because it requires 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 the popular Ethernet CRC32. You can explore the results in the available Verilog code and project files.

Xilinx's Virtex5 LX30 is the target FPGA. Table 3a shows USB CRC5 with 4-bit data using "for loop" Verilog implementation. Table 3b shows USB CRC5 with 4-bit data using "XOR" Verilog implementation. Table 3c shows CRC32 with 32-bit data using "for loop" Verilog implementation. Table 3d shows CRC32 with 32-bit data using "XOR" Verilog implementation. Note that a single Xilinx Virtex5 Slice contains four FFs and four LUTs.

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

These 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.

PARALLEL GENERATION

The parallel CRC generation method leverages a simple serial CRC generator and the linear properties of the CRC to build $H1_{NxM}$ and $H2_{MxM}$ 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 the H1 and H2 matrices contains the polynomial coefficients of the CRC output bit [j].

I've used this method successfully in several communication and test-and-measurement projects. An online parallel CRC generator tool available at OutputLogic.com uses this method to produce Verilog or VHDL code given an arbitrary data and polynomial width. A similar method is also used to generate parallel scramblers. Perhaps I'll cover the topic in a future article. ■

POP QUIZ

If you've read this article carefully, you should be able to solve the following problem.

Problem: Consider the polynomial G(x) = x + 1. What well-known error detection code does this polynomial represent? Derive a parallel equation of this polynomial for 8-bit data input. Hint: Draw a circuit with serial data input and think about how the output depends on the number of "1" bits in the input datastream.

The answer is available on the Circuit Cellar *FTP site.*

Evgeni Stavinov (evgeni@outputlogic.com) is a system design engineer for Xilinx who holds an MSEE from USC and a BSEE from The Technion — Israel Institute of Technology. He has more than 10 years of design experience in the areas of FPGA logic design, embedded software, and networking. Evgeni worked for CATC, LeCroy, and SerialTek designing test and measurement tools for USB, Wireless USB, PCI Express, Bluetooth, SAS, and SATA protocols. He also created OutputLogic.com—a web portal that offers online tools for FPGA and ASIC designers—and serves as its main developer.

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2009/234.

RESOURCES

G. Albertango and R. Sisto, "Parallel CRC Generation," *IEEE Micro*, Vol. 10, No. 5, 1990.

G. Campobello, G. Patane, M. Russo, "Parallel CRC Realization," http://ai.unime.it/~gp/publications/full/tccrc.pdf.

R. J. Glaise, "A Two-Step Computation of Cyclic Redundancy Code CRC-32 for ATM Networks," *IBM Journal of Research and Development*, Vol. 41, Issue 6, 1997.

A. Perez, "Byte-wise CRC Calculations," *IEEE Micro*, Vol. 3, No. 3, 1983.

A. Simionescu, "CRC Tool: Computing CRC in Parallel for Ethernet," Nobug Consulting, http://space.ednchina.com/upload/2008/8/27/5300b83c-43ea-459b-ad5c-4dc377310024.pdf.