CRC Tool

Computing CRC in Parallel for Ethernet

by **Adrian Simionescu**, Design Engineer

The **CRC** (Cyclic Redundancy Check) is a sophisticated checksum that has long been the most common means of testing data for correctness. Every Ethernet frame has a CRC of the data stored, so the remote system will be aware of data dropouts. A CRC doesn't identify which byte is in error, but it pretty much guarantees that you'll be alerted to at least single bit errors. All CRCs are binary polynomials that are divided into the data stream.

The CRC polynomial used for **Ethernet\AAL5** (and many other applications which use CRC32) is:

$$P(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$$

This means that the input data stream (the Ethernet frame, for example) is exclusive ORed into a 32 bit shift register that has feedback terms at the bit locations with coefficients in the formula P(x), one bit at a time. Each register bit is a function of the CRC to that point, the input data and the feedback taps.

In most systems data is transferred as a serial bit stream. Floppy and hard disks, as well as the newer optical disks, all write a single bit at a time to the medium. Modem applications are also bit oriented. This is fortunate, since the CRC is particularly well suited to serial data transfers.

If the peripheral is a parallel device, the problem becomes much more complex: for example for Ethernet 10/100 Mbps the data is nibble oriented, for Ethernet 1 Gbps it is 16 bits oriented and for Ethernet 10 Gbit is needed a CRC algorithm over 64 bits data width. The quest for speed is bringing more parallel devices into the mainstream. How do you implement a CRC in a purely parallel interface? An obvious approach is to convert the data to serial, compute the CRC, and convert it back to parallel. Although conceptually easy, fast data transfers will require a CRC clock rate equal with data rate multiplied by data width, which is, let's be realistic, impossible...

I first take the Ethernet CRC polynomial and try to see how things work. The CRC polynomial for Ethernet is a 32-degree polynomial:

$$P(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$$

Then I assume that the frame polynomial is M(x). Mathematically speaking the CRC is computed following the next 3 steps:

- M(x) is multiplied by x^m (where m=32, the degree of P(x))
- The first bits of M(x)*x^m are complemented (it means that the start value of the CRC is 32'hffffffff instead of zero).
- The remainder of M(x)*x^{32}/P(x) is complemented (this is the CRC(x)<degree m-1) and appended at the end of the frame: frame = M(x)*x^{32}+~(remainder of M(x)*x^{32}/P(x))

Compute CRC Method 1:

1. Multiply the frame polynomial by x^m and divide by P(x). The bits of the M(x) are shifted 1 by 1 from the MSB as input in the following function, which is the Verilog description of the 32 classically shift register method for computing the CRC.

```
function [31:0] next_crc32_data1; //remainder of M(x)*x^32/P(x)
  input [31:0] crc;                           // previous CRC value
  input B;                                        // input data bit (MSB first)
  begin
    next_crc32_data1 = {crc[30:0], 1’b0} ^ ({32{(crc[31]^B)}} &
      32’b00000100__1__10000010001___1___10__1__10_1__10_1__1__1);
        //      26 ^23   ^22 ^16 ^12 ^11 ^10 ^8 ^7 ^5 ^4 ^2 ^1 ^0
  end
endfunction
```

2. Assuming that the input data has 32 bits width, then the CRC function is:

```
function [31:0] next_crc32_data32;
  input [31:0] crc;
  input [31:0] inp;
  integer i;
  begin
    next_crc32_data32  = crc;
    for(i=0; i<32; i=i+1)
      next_crc32_data32  =
          next_crc32_data1(next_crc32_data32, inp[31-i]);
  end
endfunction
```

This function computes the CRC if it has 32 bits input data, but in the real Ethernet case it may have at the end of the frame only 8, 16 or 24 valid bits (meaning 1, 2 or 3 valid bytes). The function described before cannot compute the CRC in this case. If the first 1 byte is valid, then the function must be:

```
function [31:0] next_crc32_data32_be1;
  input [31:0] crc;
  input [31:0] inp;
  integer i;
  begin
    next_crc32_data32_be1 = crc;
    for(i=0; i<8; i=i+1)    // 8 instead of 32
      next_crc32_data32_be1 =
          next_crc32_data1(next_crc32_data32_be1,inp[31-i]);
  end
endfunction
```

Now, let’s consider the function which computes the CRC for all the byte enable values:

```
function [31:0] next_crc32_data32_be;
  input [31:0] crc;
  input [31:0] inp;
  input [1:0] be;  // 0 for all valid, 1 for data[31:8]
                  // (3 valid bytes).
  integer i;
  begin
    next_crc32_data32_be = crc;
    for(i=0; i<32-8*be; i=i+1)
      next_crc32_data32_be =
          next_crc32_data1(next_crc32_data32_be,inp[31-i]);
 end
endfunction
```

It may seem easy until now, but, as you already observed, this a software description and the synthesis result of this function is a complicated, slower and big area function. So, I must have convinced you not to use this function.
3. When taking a simple example:

```
// Start CRC value is 32'bffffffff. This means the first 32 bits of M(x) are
complemented.
crc = 32'hffffffff;
// Input data:
data = 32'h1c3de547;
// The new CRC:
crc = next_crc32_data32(crc, data);
// and of course the same result if I negate the data: crc =
// next_crc32_data32(32'h00000000, crc ^ data);
// Input data:
data = 32'h546092da;
// Final CRC:
crc = next_crc32_data32(crc, data);
```

Practically this function has an interesting property:

```
next_crc32_data32(32'h00000000, crc ^ data) = next_crc32_data32(crc, data);
```

That's why, instead of negating the first 32 bits of data and start with the crc = 32'h00000000 we can use the start crc = 32'hffffffff and data "as is".

Compute CRC Method 2:

1. Divide the polynomial frame by P(x) without multiplying the frame by x^m. The bits of the M(x) are shifted 1 by 1 from the MSB as input in the following function, which is the Verilog description of the 32 shift register Method 2. As you remember, to obtain the CRC we need extra shifted in m zero bits (32) to get to the final CRC.

```
function [31:0] next_div32_data1; // remainder of M(x)/P(x)
  input [31:0] crc;               // previous CRC value
  input B;                        // input data bit (MSB first)
  begin
    next_div32_data1 = {crc[30:0], B} ^ ({32{crc[31]}} &
      32'b00000100__1__10000010001___1___10__1__10_1__10_1__1__1);
        //     ^26   ^23^22   ^16 ^12 ^11 ^10 ^8 ^7 ^5 ^4 ^2 ^1 ^0
  end
endfunction
```

2. Assuming again that the input data has 32 bits width, the CRC function will be:

```
function [31:0] next_div32_data32;
  input [31:0] crc;
  input [31:0] inp;
  integer i;
  begin
    next_div32_data32  = crc;
    for(i=0; i<32; i=i+1)
      next_div32_data32  =
          next_div32_data1(next_div32_data32,inp[31-i]);
  end
endfunction
```

3. Because this function doesn't have the same property as the one described before, it means that: next_div32_data32(crc, data) is not equal with next_div32_data32(32'h00000000, crc ^ data), using the same example as in Method 1. It is obvious that, for computing the CRC, we must use this second expression.
When taking a simple example, the CRC value is:

```
// Start CRC value:
crc = 32'hffffffff;
// Input data:
data = 32'h1c3de547;
// New remainder:
crc = next_div32_data32(32'h00000000, crc ^ data); // remainder of
                                                   // M(x)/P(x)
// New CRC:
crc = next_div32_data32(crc, 32'h00000000); // remainder of
                                            // M(x)*(x^32)/P(x)
// Input data:
data = 32'h546092da;
// New remainder:
crc = next_div32_data32(32'h00000000, crc ^ data); // remainder of
                                                   // M(x)/P(x)
// Final CRC:
crc = next_div32_data32(crc, 32'h00000000); // remainder of
                                            // M(x)*(x^32)/P(x)
```

As you can see the second method has doubled the steps from the first method, but you can build a function that makes these two steps in one cycle:

```
// Input data:
data = 32'h1c3de547;
// New CRC:
crc = next_crc32_data64(32'h00000000, {crc ^ data, 32'h00000000});
// remainder of M(x)*(x^32)/P(x)
// Input data:
data = 32'h546092da;
// Final CRC:
crc = next_crc32_data64(32'h00000000, {crc ^ data, 32'h00000000});
// remainder of M(x)*(x^32)/P(x)
```

Where:

```
function [31:0] next_crc32_data64;
  input [31:0] crc;
  input [63:0] inp;
  integer i;
  begin
    next_crc32_data64  = crc;
    for(i=0; i<64; i=i+1)
      next_crc32_data64  =
          next_div32_data1(next_crc32_data64, inp[63-i]);
  end
endfunction
```

One may think the area can be doubled in this case, but if the last 32 bits will always be 0, the synthesis will minimize the function and we will obtain the same result as with the synthesis result of the function in Method 1. So you may wonder, what is the use of this Method? Before considering it foolish, check this:

```
// If only 3 bytes of data are valid data[32: 8]:
crc = next_crc32_data64(32'h00000000, {8'h00, (crc^{data[32: 8], 8'h00}),
24’h000000});
// If only 2 bytes of data are valid data[32:16]:
crc = next_crc32_data64(32’h00000000, {16’h00, (crc^{data[32: 16], 16’h00}),
16’h0000});
// If only 1 bytes of data is valid data[32:24]:
crc = next_crc32_data64(32’h00000000, {24’h00, (crc^{data[32: 24], 24’h00}),
8’h00});
```

How is that possible? The answer is simple: in the case at the end of the frame there are only 3 data valid bytes, then the frame will be multiplied only with x^{16} if I have 2 bytes valid and, of course, with x^8 if I have only 1 byte valid. So, I build a single function that computes the CRC for all bytes enable ONLY by shifting the input data and the previous CRC.

In the end, I want to offer you a Two-Step-CRC computation solution described by R.J.Glaise in the article: "A two-step computation of cyclic redundancy code CRC-32 for ATM networks" (http://www.research.ibm.com/journal/rd/416/glaise.html).

$$P123(x) = x^{123} + x^{111} + x^{92} + x^{84} + x^{64} + x^{46} + x^{23} + 1$$

Build the function:

```
function [122:0] next_div123_data1; // remainder of M(x)/P(x)
  input [122:0] crc;                // previous CRC value
  input B;                          // input data bit (MSB first)
  begin
    next_div123_data1 = {crc[121:0], B}^({123{crc[122]}}&
                        123’h0008000101000010000400000800001);
  end
endfunction

function [122:0] next_div32_data32;
  input [122:0] crc;
  input [31:0] inp;
  integer i;
  begin
    next_div32_data32 = crc;
    for(i=0; i<32; i=i+1)
      next_div32_data32 =
          next_div123_data1(next_div32_data32, inp[31-i]);
  end
endfunction
```

And for obtaining the final CRC:

```
function [31:0] next_crc32_data122;
  input [122:0] inp;
  integer i;
  begin
    for(i=0; i<123; i=i+1)
      next_crc32_data122 =
          next_crc32_data1(32’h00000000, inp[122-i]);
  end
endfunction
```

Example:

```
crc_123 = {123{1’b0}}; // start CRC value
data = 32’h1c3de547;
// First 32 bits are complemented
```

```
crc_123 = next_div32_data32(crc_123, ~data);
data = 32’h546092da;
crc_123 = next_div32_data32(crc_123, data);
// and so, till the end of the frame ...
// And the final crc:
crc = next_crc32_data122(crc_123);
```

Don’t forget to negate the final CRC before appending at the end of the frame.

Now I let you think... or **NOBUG Consulting** can think for you.

support@nobugconsulting.com