AN02007: Calculating Cyclic Redundancy Checks (CRC) on XCORE

Publication Date: 2024/10/16
Document Number: XM-015153-AN v1.0.0

IN THIS DOCUMENT

A Cyclic Redundancy Check (CRC) is a type of error-detecting code used to detect accidental changes to raw data in digital networks and storage devices. It's a simple but powerful technique to ensure data integrity. When data is transmitted or stored, CRCs help verify that the information has not been altered or corrupted. Typically, checksums, calculated by a CRC, are attached to their corresponding data to allow a consumer of that data to verify it's integrity. The computation involved is based around polynomial division.

1 Nomenclature

The following standard terms are used to define a CRC algorithm instance:

- Bit Width: The width of the CRC value, n bits. The width of the polynomial is then limited to n bits ((n+1) if including the implicit bit).
- Polynomial: Used to generate the CRC, stored as a bit string with each bit representing an increasing or decreasing order of x.
- Input reflected: If reflected then the input bytes are fed into the CRC in reverse order, i.e. bit 0 is first and bit 7 is last.
- Result reflected: The output CRC value is reflected, i.e. bit-reversed, before being returned. It is done over the whole n bits, not byte wise.
- Initial Value: The value used to initialize the CRC value.
- Final XOR value: This is a value, the same width as the CRC, that is XORed in at the end of the data ingestion.

C functions that illustrate the behaviour of an XMOS ISA assembly instruction will have the suffix `_asm`.

2 Standard form of a CRC

Upon searching for a CRC implementation one is likely to come across the following implementation, or one very similar:

```
uint32_t crc32(uint32_t len, uint8_t *data, uint32_t init,
               uint32_t final_xor) {
    uint32_t val, crc = init;
    while( len-- ) {
        val = (crc ^ *data++) & 0xff;
        for(int i=0; i < 8; i++)
            val = val & 1 ? (val>>1)^POLY : val>>1;
        crc = val ^ crc >> 8;
    }
    return crc ^ final_xor;
}
```

In the implementation above the data is consumed byte-wise but with a trivial transformation it could be adjusted to consume any number of bits. The code below assumes that the data is stored as an array of `len` bits:

```
uint32_t crc32(uint32_t len, uint8_t *data, uint32_t init,
               uint32_t final_xor) {
    uint32_t val, crc = init;
    while( len-- ) {
        val = (crc ^ *data++) & 0x1;
        if (val)
            crc = POLY ^ crc >> 1;
        else
            crc = crc >> 1;
    }
    return crc ^ final_xor;
}
```

Of note here is that the `data` is inserted at the left most end of the shift register, `crc`., this is contrary to the naive implementation, which is on the right(in little endian format).

In the byte-wise version you may also find the:

```
for(int i=0; i < 8; i++)
    val = val & 1 ? (val>>1)^POLY : val>>1;
```

replaced with:

```
val = precomputed_look_up_table_256(val);
```

This is a further optimised version, as the reader might observe that the resulting `val` depends solely on the 8 bit initial `val` and thus can be precomputed.

3 The XMOS CRC instructions

Across the XCORE ISAs there are four instructions that assist in calculating CRCs.

- `crc32`
- `crc8`
- `crcn`
- `crc32_inc`

The implementation can be generalised by the following C code:

```
uint32_t crc_xcore(uint32_t crc, uint32_t data, uint32_t poly, uint32_t count){
    for(int i=0; i<count; i++){
      if(crc&1)
        crc = ( ((data&1)<<31) | (crc>>1) )^poly;
      else
        crc =   ((data&1)<<31) | (crc>>1)       ;
      data >>= 1;
    }
    return crc;
}
```

where `data`, `crc` and `poly` from the C implementation match to `d`, `x`, and `p` from the assembly mnemonics.

Instruction	count	Mnemonic and Arguments	Other
`crc32`	32	`crc32 x, d, p`	
`crcn`	n	`crcn x, d, p, n`	
`crc8`	8	`crc8 x, e, d, p`	`e = d << 8`
`crc32_inc`	32	`crc32_inc x, d, p, a, b`	`a = a + b`

The key difference between the standard form and the XMOS instructions is the point of insertion into the shift register. In the XMOS implementations the data is inserted at the right most end of the shift register with the bit exiting on the left determining if the polynomial xor should be applied. In the standard form, otherwise known as the DIRECT TABLE ALGORITHM, the data is inserted at the left most end. This is shown in Fig. 1.

This difference manifests as the standard form XORing every bit of inserted data into the crc as it is ingested and the XMOS implementation running “32 bits behind”.

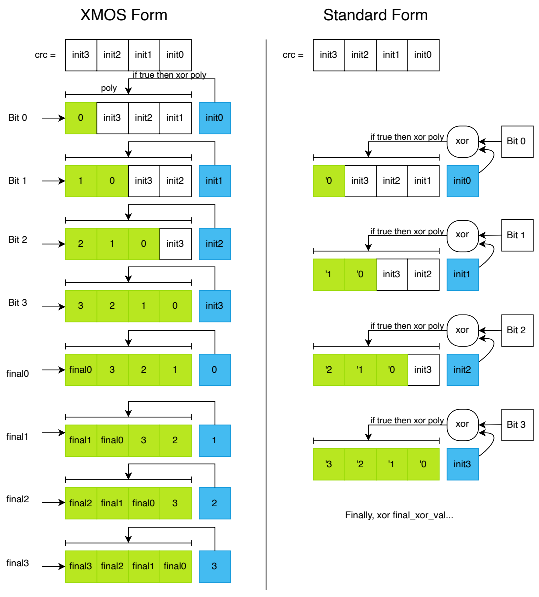

Fig. 1: CRC creation

3.1 Data bit ordering

In order to compute CRC efficiently the XMOS CRC instructions can ingest many bits at a time. Upon executing a XMOS CRC instruction the instruction will ingest the data from the right most bit, bit 0, until it has ingested the count required by the particular instruction.

Bits are ingested from the right, as this matches up with the way that ports serialise and deserialise data. Serialisation outputs the LSB first and shifts right. Deserialisation shifts right and insert the new data as the MSB.

If the bit ordering of data stored in memory or data coming from your input stream is different, then the BITREV and BYTEREV instructions can be used to alter the order. BITREV will swap all bits form left to right. BYTEREV swaps the byte in a word. A combination of the two can swap all bits in a byte.

3.2 Polynomial representation

The XMOS CRC instructions use a reversed representation of the polynomial. This means that the bit field representing the polynomial will have its highest order powers of x in the least significant bits, i.e. to the right. The left most bit of the bit field represents the x^0 term, the right most represents x^31 and x^32 is implicitly one. We discuss polynomials of less than 32 bits in *Computing the CRC with a polynomial that is shorter than 32 bits*.

4 Example - Ethernet

The Ethernet 801.22 standard specifies how the CRC, or Frame Check Sequence, should be calculated. In summary, the first 32 bits ingested following the preamble should be inverted, this is equivalent to the initial value of the CRC being set to 0xffffffff. The remaining data should be ingested as normal, finally, the final xor should be 0xffffffff.

Therefore, the CRC for an ethernet frame would be computed by the following:

```
uint8_t data[];
uint32_t ethernet_crc =
  crc32(sizeof(data), data, 0xffffffff, 0xffffffff);
```

The returned value *ethernet_crc* would be appended to the frame in the case of a ethernet transmitter and in the case of a receiver could be compared to the received FCS.

In order to implement this efficiently with the XMOS instruction set a few optimisations may be incorporated. Below is the pseudo assembly for implementing a ethernet transmit using the XMOS CRC instruction set. Data needs to be fetched from somewhere, i.e. input from a channel or loaded from memory, then ingested into the CRC. After all data has been transmitted the FCS is complete and need to be transmitted also.

```
set crc, 0

fetch data               // get the data
crc32 ~data, crc, poly   // ingest the first word inverted
out data                 // output the data

data_loop:
  fetch data             // get the data
  crc32 data, crc, poly  // ingest the data
  out data               // output the data
  bu loop                // loop

crc32 ~0, crc, poly      // finalise the crc
out crc                  // output the crc
```

Conveniently the finalisation of the crc is a single crc32 of `0xffffffff` which can be made with a MKMSK instruction.

As an optimisation we can remove the initialisation of the crc register to zero and the first crc32 instruction. As the first crc is always with zero, we can simply replace it with a NOT instruction:

```
fetch data               // get the data
not crc, data            // init to data inverted
out data                 // output the data

data_loop:
  fetch data             // get the data
  crc32 data, crc, poly  // ingest the data
  out data               // output the data
  bu loop                // loop

crc32 ~0, crc, poly      // finalise the crc
out crc                  // output the crc
```

Here is the optimised XMOS pseudo assembly for implementing a ethernet receiver using the XMOS CRC instruction set. In the receive case the crc is initialised to a magic number such that the resulting correct data sequence will produce a `0xffffffff` output crc. Such an output is trivial to branch against.

```
crc = magic             // initialise the crc to a magic number

data_loop:
  input data            // get the data
  crc32 data, crc, poly // ingest the data including the FCS

//check that crc is 0xffffffff
```

The magic number can be computed by running the crc in reverse with the data required to make the crc be as if it were initialised to the `initial_crc`. This is achieved by running:

```
uint32_t make_magic_number(uint32_t poly)
{
  uint32_t data[2] = {0, ~0};
  uint32_t crc = 0;
  for (int i = ARRAY_SIZE(data) - 1; i > 0; i--)
    crc = crc32_rev(data[i], crc, poly);
  return crc;
}
```

This allows the CRC to be computed such that at the end of the ingestion of all the data the crc, in correct, should be in the state of `final_xor_value`.

5 Computing the CRC with a polynomial that is shorter than 32 bits

Using the XMOS CRC instructions any length of polynomial up to 32 can be implemented. To achieve an M-bit CRC

- Set the polynomial register to the reversed representation of the desired polynomial. i.e. for the 7 bit poly x^7 + x^3 + 1 the forward representation is 0x9, the reversed representation is 0x48. Note that for an M-bit CRC bit M-1 must be '1' and the M most-significant bits are '0'.
- Apply the `initial_crc` value, M-bits, to the initial M-bits of the data by preforming an XOR between them.
- Ingest the data as normal using any of the CRC instructions.
- Ingest a final 32 zeros with the `final_xor` value XORed onto it to finish the CRC.

An illustration of this can be seen in the `test_n_bit_poly()` function in the source code.

```
//First XOR in the initial CRC into the data
for(int len=0;len < poly_len; len +=8){
  data[len/8] = (init ^ data[len/8]);
  init >>= 8;
}

//Then perform the data ingestion as normal
uint32_t xmos_crc = 0;
for(int i=0;i<sizeof(data);i++){
  //crc8_asm consumes one byte at a time
  xmos_crc = crc8_asm(xmos_crc, data[i], poly);
}

//Finally, ingest 32 zeros with the final_xor included.
xmos_crc = crc32_asm(xmos_crc, final_xor, poly);
```

6 Computing the CRC by ingesting fewer than 32 bits at a time

The XMOS instruction set makes this very easy, `crcn` ingests `n` bits at a time, up to 32. Alternatively, data can be ingested and shifted bytewise by using the `crc8` instruction.

7 Helpful links

- *CRC Calculator* https://crccalc.com
- *CRC pages on Wikipedia* http://en.wikipedia.org/wiki/Cyclic_redundancy_check