Fast CRC32 in Software

Richard Black

18th February 1994

1 Introduction

This document describes how to implement a 32-bit CRC algorithm efficiently. By efficiently I mean about twice as fast as any previous algorithm that I have come across. I'm not necessarily implying that this is a world first.

This document explains a little of the maths, but although I understand it[1], I don't understand it well enough to explain it properly. This is a document for C hackers, not mathematicians. Flames from mathematicians will be forwarded to `/dev/null`.

2 Maths

The k bits of data that are the message to be checked by the CRC can be represented by a polynomial $F(x)$ of degree $k-1$. As an example the one octet message 01010010_2 is represented by the polynomial x^6+x^4+x. In general, leading zero bits do not affect $F(x)$ whereas trailing zero bits each add a factor of x. The generating polynomial $G(x)$ is chosen by very clever mathematicians and is essentially fixed. The polynomial $L(x)$ is used to represent 32 successive ones.

The CRC is defined as the ones complement of a remainder $R(x)$ obtained from the modulo two division of $x^{32}F(x) + x^k L(x)$ by the generating polynomial $G(x)$. I.e. the CRC is $R(x) + L(x)$.

The multiplication by x^{32} corresponds to leaving space at the end of the message for the generated CRC. The addition of $x^k L(x)$ (which is equivalent to inverting the first 32 bits of the message) precludes erroneous addition or deletion of leading zeros. The inversion of the CRC before sending it precludes erroneous addition of trailing zeros.

2.1 Transmission

As described:

$$[x^{32}F(x) + x^k L(x)]/G(x) = Q(x) + R(x)/G(x) \tag{1}$$

for some quotient $Q(x)$ and remainder $R(x)$. The CRC is the complement.

$$CRC = L(x) + R(x) \tag{2}$$

The transmitted message denoted $M(x)$ is $F(x)$ followed by the CRC.

$$M(x) = x^{32}F(x) + CRC \tag{3}$$

[1] I would like to thank Joshua Ross of the Computer Laboratory for explaining it to me.

2.2 Reception

At the receiver the operation is again repeated. This time however the value used is $M(x)$ rather than $F(x)$ also the message is 32 bits longer so k is replaced by $k + 32$. We consider:

$$[x^{32}M(x) + x^{k+32}L(x)]/G(x) \tag{4}$$

Factoring out x^{32} we get:

$$x^{32}[M(x) + x^k L(x)]/G(x) \tag{5}$$

Substituting (3) and thence (2) gives:

$$x^{32}[x^{32}F(x) + L(x) + R(x) + x^k L(x)]/G(x) \tag{6}$$

Taking the divisor inside we can now substitute (1).

$$x^{32}[Q(x) + R(x)/G(x) + R(x)/G(x) + L(x)/G(x)] \tag{7}$$

From which the $R(x)$ terms cancel in our modulo two arithmetic. We are, or course, only interested in the remainder of the division which is therefore.

$$x^{32}L(x)/G(x) \tag{8}$$

Which is a known constant value (called the *residual* or *residue*) $C(x)$ for all messages $F(x)$.

3 What polynomial

The usual polynomial, which is used for AAL5, FDDI, and probably Ethernet too is:

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

The residual for this polynomial is:

$$x^{31} + x^{30} + x^{26} + x^{25} + x^{24} + x^{18} + x^{15} + x^{14} + x^{12} + x^{11} + x^{10} + x^8 + x^6 + x^5 + x^4 + x^3 + x + 1$$

In hexadecimal these are $104C11DB7_{16}$ and $C704DD7B_{16}$.

4 Some Examples

For the purposes of the AAL5 checksum, individual bytes are presumed to be transmitted with the most significant bit first.

For a forty eight byte ATM cell which forms a single AAL5 block. If the forty data bytes are all zeros then the CRC value in the last four bytes is 0x86, 0x4D, 0x7F, 0x99.

If the forty bytes of data are all `0xff`s then the CRC value in the last four bytes is 0xC5, 0x5E, 0x45, 0x7A.

If the forty bytes of data are sequential numbers beginning `0x01` and ending `0x28` then the CRC value in the last four bytes is 0xBF, 0x67, 0x1E, 0xD0.

5 Hardware Serial Implementation

Figure 1 shows a simple shift register implementation of the CRC algorithm. The shift register is initially pre-set to all ones. The data is shifted in bit by bit. As each data bit arrives in the system

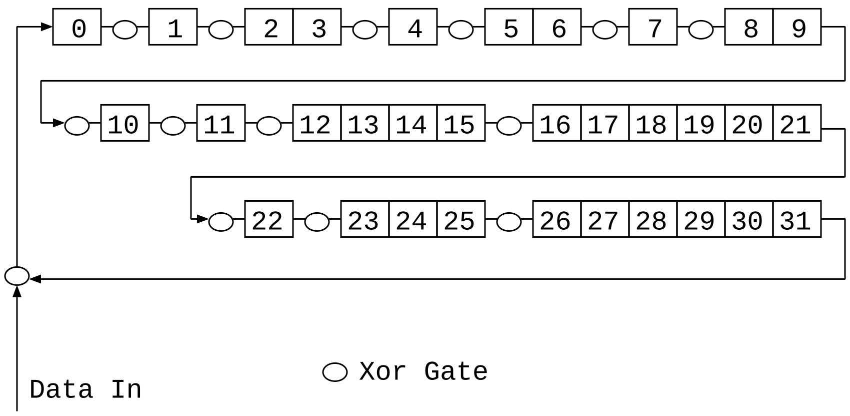

Figure 1: CRC32 shift register implementation

it is individually multiplied by x^{32} and added (added is the same as exclusive-or in modulo two arithmetic) to the current remainder. If the remainder then has a term in x^{32} then the divisor $Q(x)$ is subtracted from the current remainder. This proceeds until all the data bits have been presented.

If the calculation is being performed in the transmitter then the contents of the shift register are shifted out without further feedback and each bit is added inverted to the tail of the message.

If the calculation is being performed in the receiver then the shift register, will at this point, in the case of an error free transmission, contain the residue $C(x)$.

6 Software Implementations

In all cases, the macro `QUOTIENT` has the value `0x04c11db7`. Although I am detailing these in terms of software, they have direct parallel implementations in the hardware domain. I am merely describing them from the point of view of a programmer implementing them in software.

6.1 Algorithm One

The first and most obvious algorithm is to simply implement the hardware algorithm in software. We use a 32 bit register to store the current remainder. For each byte in the message we take each bit in turn and if a new term in x^{32} is generated we XOR in the polynomial. The code is as follows:

```
unsigned int wombat(unsigned char *data, int len)
{
    unsigned int        result;
    int                 i,j;
    unsigned char       octet;

    result = -1;

    for (i=0; i<len; i++)
    {
        octet = *(data++);
        for (j=0; j<8; j++)
        {
```

```
            if ((octet >> 7) ^ (result >> 31))
            {
                result = (result << 1) ^ QUOTIENT;
            }
            else
            {
                result = (result << 1);
            }
            octet <<= 1;
        }
    }

    return ~result;             /* The complement of the remainder */
}
```

The user must be careful to write the four octets of the result into the data part of the cell in the correct order (that is bits 24-31 first and bits 0-7 last) irrespective of the endian of the machine.

6.2 Algorithm two

The next algorithm makes the observation that it is possible to do the calculation by multiplying by x^{32} at the end by feeding in an additional 32 zeros, rather than by doing it at the beginning of the division. We initialise the register to the first thirty two bits of $F(x) + x^{k-32}L(x)$.

The code is written assuming that the transmitting end calls with the length field higher by an additional four bytes (which are where the CRC will be put and which are initially zero). The code is as follows:

```
unsigned int wombat(unsigned char *data, int len)
{
    unsigned int        result;
    int                 i,j;
    unsigned char       octet;

    if (len < 4) abort();

    result = *data++ << 24;
    result |= *data++ << 16;
    result |= *data++ << 8;
    result |= *data++;
    result = ~ result;
    len -=4;

    for (i=0; i<len; i++)
    {
        octet = *(data++);
        for (j=0; j<8; j++)
        {
            if (result & 0x80000000)
            {
                result = (result << 1) ^ QUOTIENT ^ (octet >> 7);
            }
            else
            {
```

```
                result = (result << 1) ^ (octet >> 7);
            }
            octet <<= 1;
        }
    }

    return ~result;             /* The complement of the remainder */
}
```

This code contains an additional benefit which is that it is possible to perform the computation at the recipient without multiplying by the additional x^{32} term. Thus the remainder in equation (8) is simply $L(x)$ and the calculation function returns 0.

6.3 Algorithm three

Algorithm three is the algorithm which I have seen most usually in other literature and implementations. It is a substantial optimisation based on the observation that in algorithm two, (apart from the first thirty two bits) the data to be transmitted is not needed for the control path until some thirty two bits later. This permits the division by subtraction to be done more than one bit at a time using a pre-generated table.

This table is normally indexed by eight bits because in that case the table is 1K in size. Attempting to do more bits than this at once (e.g. 16) is usually *slower* because at 256K the table has horrific cache performance.

The code to generate the table is:

```
unsigned int crctab[256];

void crc32_init(void)
{
    int i, j;

    unsigned int crc;

    for (i = 0; i < 256; i++)
    {
        crc = i << 24;
        for (j = 0; j < 8; j++)
        {
            if (crc & 0x80000000)
                crc = (crc << 1) ^ QUOTIENT;
            else
                crc = crc << 1;
        }
        crctab[i] = crc;
    }
}
```

The implementation then proceeds byte wise feeding the eight bit subtraction term in at each stage along with the next eight bits of data. The code is:

```
unsigned int wombat(unsigned char *data, int len)
{
    unsigned int        result;
```

```
    int                 i;
    unsigned char       octet;

    if (len < 4) abort();

    result = *data++ << 24;
    result |= *data++ << 16;
    result |= *data++ << 8;
    result |= *data++;
    result = ~ result;
    len -=4;

    for (i=0; i<len; i++)
    {
        result = (result << 8 | *data++) ^ crctab[result >> 24];
    }

    return ~result;
}
```

6.4 Algorithm four

This algorithm takes the observation used to produce algorithm three one stage further. Whereas it still performs the division eight bits at a time for cache performance reasons, it is designed in such a way that the data can be fed into the remainder in thirty two bit units (which are more efficient units on most computers). This necessitates re-ordering the bits of the polynomial in a non-monotonic fashion depending on the endian of the computer on which the algorithm is running. The polynomials in the lookup table likewise have the same non-linear transform applied to them as they are generated.

The code for initialising the table becomes:

```
unsigned int crctab[256];

void crc32_init(void)
{
    int i,j;

    unsigned int crc;

    for (i = 0; i < 256; i++)
    {
        crc = i << 24;
        for (j = 0; j < 8; j++)
        {
            if (crc & 0x80000000)
                crc = (crc << 1) ^ QUOTIENT;
            else
                crc = crc << 1;
        }
        crctab[i] = htonl(crc);
    }
```

```
}
```

Note the use of htonl to rearrange the terms in the partial quotients. The calculation code then also becomes dependent on the endian of the machine:

```
unsigned int wombat(unsigned char *data, int len)
{
    unsigned int        result;
    unsigned int        *p = (unsigned int *)data;
    unsigned int        *e = (unsigned int *)(data + len);

    if (len < 4) abort();

    result = ~*p++;

    while( p<e )
    {
#if defined(LITTLE_ENDIAN)
        result = crctab[result & 0xff] ^ result >> 8;
        result = crctab[result & 0xff] ^ result >> 8;
        result = crctab[result & 0xff] ^ result >> 8;
        result = crctab[result & 0xff] ^ result >> 8;
        result ^= *p++;
#else
        result = crctab[result >> 24] ^ result << 8;
        result = crctab[result >> 24] ^ result << 8;
        result = crctab[result >> 24] ^ result << 8;
        result = crctab[result >> 24] ^ result << 8;
        result ^= *p++;
#endif
    }

    return ~result;
}
```

Of course this now only works for word aligned data and assumes that the data is an exact number of words. I do not regard these as significant limitations. This code is approximately *twice* as fast as algorithm three. It should also be noticed that since the data is not broken up into bytes, this code has an even larger benefit when used as an integral part of a data copy routine. The result is also in the local form and should be written directly as a word to the data and not as a sequence of bytes.

The tinkerer will observe that the C can be made slightly more beautiful by rearranging the insertion of the data to the start of the loop, using a pre-condition of -1, and stopping early. However, apart from the issue at the receiver, such code will be considerably slower, because the data is required synchronously. In the code as I have written it, the compiler[2] will perform the load on `p` early in the loop body, and so the load delay will have passed by the time the data is required.

This loop compiles to 16 instructions on the Arm, 29 on the Mips, 30 on the Alpha, and 19 on the HP-PA. The Arm's instruction count is so low because of the ability to perform shifts for free on all operations, but it looses out on its blocking loads to about the same number of cycle as the Mips. The Alpha gains over the Mips with its `s4addq` instruction, but looses this win because 32bit loads sign extend and require zeroed. The HP-PA assembler is too weird to comment further.

[2]Given a decent compiler such as `gcc`.

On all architectures, assuming the 1K lookup table is in the cache, the algorithm proceeds at an average of about 1 bit per clock cycle. This represents 25 Mbit/sec on the FPC3 or Maxine, and 150 Mbit/sec on the Sandpiper.

This implementation is particularly suitable for hardware at a point where the data path is thirty two bits wide. The generating register can be implemented as a 32 bit register with an 8 bit barrel roll operation. Between each data word being exclusive-ored in, four rolls with exclusive or of the quotient can be performed. (or two of sixteen). This makes the xor circuitry much simpler and reduces the percentage of cycles which must involve the actual data.