Combining and splitting CRCs

Nicolai Stange

Version 0.1, November 13, 2015

Contents

1 Overview

Let us assume, that we are given some bitstream which is logically split into two chunks as depicted in figure 1. Furthermore, assume that we have already computed the individual chunks' cyclic

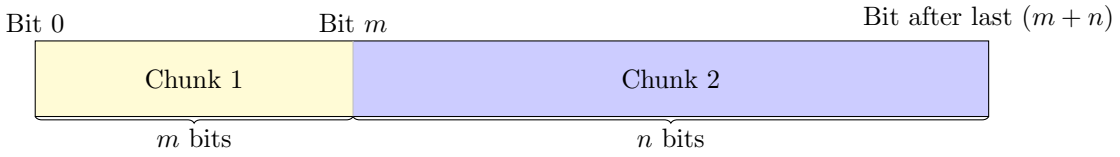

Figure 1: Data stream made up of two logical chunks of size m and n

redundancy checksums (CRC) as if they had been isolated streams of data on their own. Is it possible to efficiently combine the two CRCs into the the complete data's CRC, i.e. that of the two chunks concatenated? As we will see, the converse question is also of interest: given the CRC of the whole data, as well as that of only one of its two chunks, is it possible to "split off" the other chunk's CRC?

The answer to the first question seems to be well known, although I only managed to find it implemented in some source code. In section 2, I will present it in a (hopefully) more convenient form with references to the basic ideas involved. For the second question, that of "splitting" CRCs, I haven't found anything on the web. Obtaining a sensible solution does not involve any black magic though and I will handle it in section 3. A great part of section 2 about CRC combination deals with the problems of optimizing polynomial multiplications at the machine code level as well as optimizing at the algorithmic level what I term "hybrid exponentiation", a mixture of simple linear and binary exponentiation. Fortunately, the results can be used verbatim for checksum splitting also – basically, section 3 only sets up the problem of CRC splitting such that the previous results about combination can be reused.

I do recognize the fact that a great fraction of this document's intended audience might not be extraordinary proficient in the language of algebra. However, I see little value in rederiving basic facts already known to the mathematicians for quite a while, especially because the very same reasoning carried out at the polynomial or even algorithmic level tends to take a far more tedious form. In order to not scare you off, I've put most of the mathematical stuff into appendices of their own and tried to keep the main document free of anything going beyond polynomials. Furthermore, I kept the mathematical appendix B quite verbose so that anybody should be able to grasp the basic ideas, if wanted.

Finally, a particularly good guide on CRC computation algorithms is Williams 1993. However, being targeted at a mathematically less inclined audience, this guide quickly leaves the language of polynomials and focuses more on the algorithms' implementations rather than a strict mathematical formulation. In appendix C, I attempt to "recover" that formulation in order to make drawing connections between the two worlds less painful. After all, when shifting chunks of data back and forth, it is important to know what the actual meaning of your summing register's content is after any iteration of the standard CRC algorithms.

1.1 Preliminaries and notation

The set of all polynomials with coefficients in $\{0, 1\}$ will be denoted by $\mathbb{F}_2[X]$. This is consistent with the notation $\mathbb{R}[X]$, the polynomials with coefficients in $\mathbb{R}$ you probably know from school: $\mathbb{F}_2$ is the set $\{0, 1\}$ furnished with an addition as well as a multiplication. For these two laws of composition, the rules known from the integers hold except that $1 + 1 = 0$. By definition of -1, $1 + (-1) = 0$ and thus, one often writes $-1 = 1$. Note that addition as well as subtraction are both equivalent to the exclusive or operation (`xor`). Any polynomial in $\mathbb{F}_2[X]$ will be denoted by lowercase latin letters such as p, q and g. The latter, g, will be exclusively used to denote a CRC algorithm's generator polynomial. Note that I do *not* use the notation $p(X)$ for the polynomials: they are not functions but mathematical objects and indeed they won't get evaluated ever.

I assume that you are familiar with the fact that any stream of bits may be interpreted as a polynomial $p \in \mathbb{F}_2[X]$ and that computing that bitstream's CRC is nothing but calculating the remainder of the division of p by the CRC type's generator polynomial g, in short: $p\%g$. Some facts about the remainder or "modulo" operation are derived in appendix B.2, the most important one being that for any $p, q \in \mathbb{F}_2[X]$, we have

$$(p \cdot q)\%g = ((p\%g) \cdot q)\%g$$

(c.f. equation (30)).

Finally, the logarithm to the base two will be written as "lg" whereas "ln" is the one to the base e.

2 Combining checksums

In this section, we will assume that all CRC calculations are seeded with zero as well as that the involved data bitstreams aren't augmented. Thus, nothing is either prepended or appended to the inputs before throwing them at the algorithms of appendix C. You will learn in the next section 3 how to remove those artifacts before processing, if needed.

Suppose, we have got two bitstreams, $p_1, p_2 \in \mathbb{F}_2[X]$ and that we have already computed their individual checksums $p_1\%g$ and $p_2\%g$ for some fixed generator polynomial g. Question: are we able derive the checksum of the two bitstreams' concatenation more efficiently than resumming the whole chunk of data? That is, can we somehow calculate $(p_1 \cdot X^{\deg p_2+1} + p_2)\%g$ from the values of $p_1\%g$ and $p_2\%g$? Using properties of the modulo operation alone, we derive

$$\begin{aligned} & (p_1 \cdot X^{\deg p_2+1} + p_2)\%g \\ = \; & p_1 \cdot X^{\deg p_2+1}\%g + p_2\%g \end{aligned}$$

This suggests the algorithm depicted in figure 2: Take $p_1\%g$ as the seed for the `simple` or equivalent algorithm of Williams 1993 and feed $\deg p_2 + 1$ zero bits into it. Add $p_2\%g$ to obtain the final checksum of the complete, concatenated bitstream. This algorithm reduces the runtime of $O(\deg p_1 + \deg p_2 - \deg g + 1)$ necessary for naive resumming to $O(\deg p_2 - \deg g + 1)$ plus it avoids any costly memory accesses to the bitstream.

Repeated squaring However, it turns out, that we can do better in calculating $p_1 \cdot X^n\%g$: the algorithm found in Spelvin 2014 achieves a runtime logarithmic in $\deg p_2 + 1$ by using a technique for evaluating powers commonly known as *repeated squaring* or *binary method for exponentiation*,

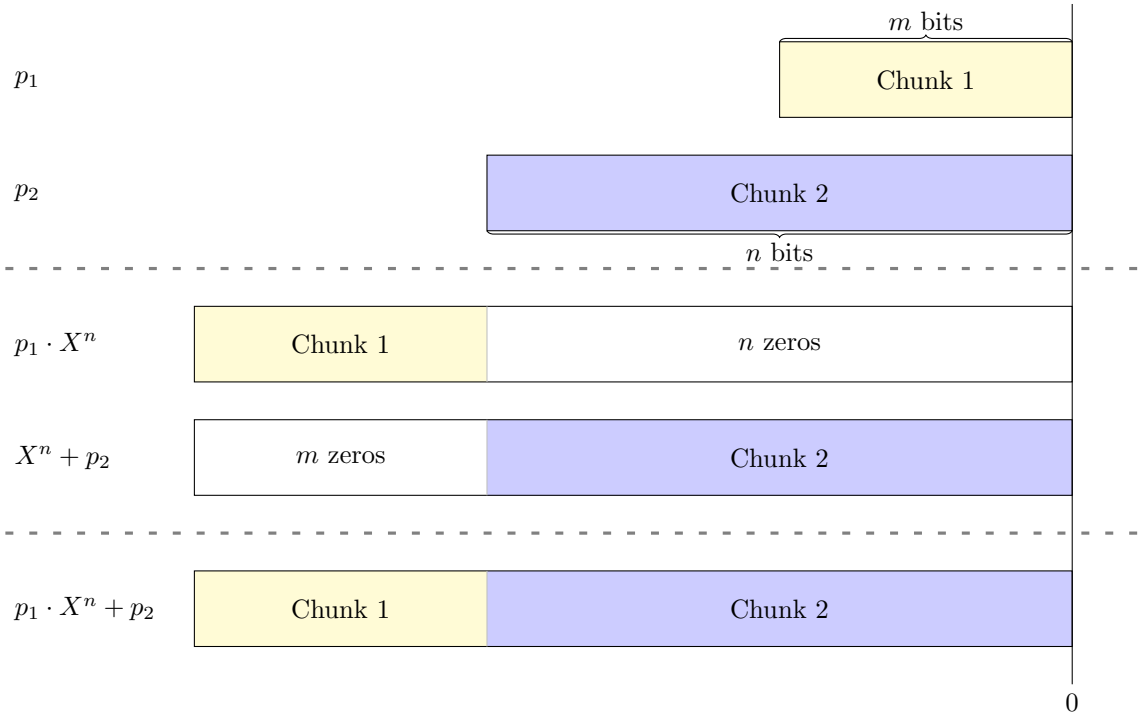

Figure 2: Merging of two data chunk's checksums

c.f. Knuth 1997, section 4.6.3. The underlying idea is to write the exponent n in base two representation

$$n = a_{\lfloor \lg n \rfloor} 2^{\lfloor \lg n \rfloor} + a_{\lfloor \lg n \rfloor - 1} 2^{\lfloor \lg n \rfloor - 1} + \ldots + a_1 2^1 + a_0 2^0 \tag{1}$$

with some $a_i \in \{0, 1\} \forall 0 \leq i \leq \lfloor \lg n \rfloor$ and exploit the basic rules of exponentiation, namely $\forall k, l \in \mathbb{N}$ we have

$$x^{k+l} = x^k \cdot x^l \tag{2}$$

$$x^{k \cdot l} = \left(x^k\right)^l \tag{3}$$

In particular, $X^{2^l} = X^{2l} = X^l \cdot X^l$. Thus, working our way from right to left in equation (1), all we have to do is to repeatedly square some auxiliary register to obtain X squared i times, $0 \leq i \leq \lfloor \lg n \rfloor$ and, depending on the a_i, multiply those intermediate values against the result register at each step.

It should be clear by now that we need $\lfloor \lg n \rfloor$ multiplications for the squarings and $\nu(n)$, the number of bits set in n's binary representation, multiplications of the auxiliary against the result register.

2.1 Tweaking the polynomial multiplications

What does one single multiplication actually consist of? First of all, it is a multiplication of polynomials. Second, every multiplication is to be followed by a modulo g operation. The naive

implementation of multiplying two general polynomials h_1 and h_2 term by term has got a runtime of $O((\deg h_1 + 1)(\deg h_2 + 1))$. There do exist schemes to improve this general case significantly, c.f. Knuth 1997, section 4.3.3. However, the case at hand is in no way general: we are working with coefficients in the "binary" field $\mathbb{F}_2$ and since products are reduced modulo g after each multiplication, the factor polynomials' degrees are bounded by $\deg g - 1$. Usually, the generator polynomial g is chosen such that $\deg g$ is equal or less than the machine's word size which in turn allows us to store the factor polynomials in one hardware register each. All this taken together allows us to exploit parallelism at the circuitry level and do the multiplication in $O(\deg g - 1)$: multiply (shift) the second polynom by each term of the first one and add (`xor`) the results together. If the modulo g were to be carried out after the whole multiplication, we needed two double-word registers: one for the shifts of the second factor and one for the result to `xor` into. A careful analysis reveals that it is possible to fuse the multiplication loop with the modulo reduction loop though. Besides eliminating the overhead of one loop, this fusion reduces the storage requirements for the calculation from two double-word registers plus one single-word register to three single-word registers only. The helper function `gf2_multiply` in Spelvin 2014 follows the scheme just described.

In order to analyze runtimes beyond the big-O notation, let us examine the `gf2_multiply` function from the Linux Kernel in more detail. It handles the case of $\deg g = 32$ and reads as

```
1  u32 gf2_multiply(u32 x, u32 y, u32 modulus)
2  {
3    u32 product = x & 1 ? y : 0;
4    int i;
5
6    for (i = 0; i < 31; i++) {
7      product = (product >> 1) ^ (product & 1 ? modulus : 0);
8      x >>= 1;
9      product ^= x & 1 ? y : 0;
10   }
11
12   return product;
13 }
```

Listing 1: `gf2_multiply` from the Linux Kernel

Note that here, the polynomials are stored in reversed order, that is the least significant bits correspond to a polynomial's most significant coefficient. Furthermore, the degree of `modulus` is assumed to be exactly equal to 32 and its highest order coefficient is implicit and not stored anywhere (otherwise 33 bits of storage would have been needed). Take a look at listing 2 to see how GCC translates `gf2_multiply` to `x86-64` machine code. As you might want to look up in *Intel 64 and IA-32 Architectures Optimization Reference Manual* 2015 on your own, every instruction used within the assembler listing above has got a latency of one cycle, except for the `cmovne` instruction. Up to the advent of the Broadwell microarchitecture, `cmovne` had got a latency of two cycles. Now, superscalar processors are able to execute several instructions per cycle and in order to get a theoretical lower bound on one `gf2_multiply` invocation's runtime, let us analyze how much potential for parallelism there actually is in the code from above. Consider the data dependency graph as drawn in figure 3. I denoted "reverse" data dependencies by dashed arrows where an arrow's target instruction scrambles a register used as input at the source instruction. However, these reverse dependencies are usually broken up by superscalar architectures through a technique called *register renaming* and may thus be safely ignored. Furthermore, the renaming stage is almost always capable of eliminating pure register `mov`'s completely. Taking these facts into consideration, the longest path in figure 3 is made up of two cycles for the preamble and either four or five cycles for the 31 loop iterations each, depending on whether `cmovne` takes one or two cycles. These

```
gf2_multiply:
        /*
         * Register assignments
         * x       - %edi
         * y       - %esi
         * modulus - %edx
         * i       - %r8d
         * product - %eax
         */
        test   $0x1, %dil          /* x&1 ? */
        mov    $0x0, %eax          /* product=0 */
        mov    $0x1f, %r8d         /* i=31 */
        cmovne %esi, %eax          /* if (x&1) product=y */
.Lloop1:
        mov    %eax, %ecx          /* %ecx=product */
        shr    %ecx               /* %ecx=product>>1 */
        and    $0x1, %eax          /* %eax=product&1 */
        cmovne %edx, %eax          /* if(product&1) %eax=modulus */
        shr    %edi               /* x>>=1 */
        xor    %ecx, %eax          /* product=(product>>1)^
                                      (product&1 ? modulus : 0) */
        mov    %edi, %ecx          /* %ecx=x */
        and    $0x1, %ecx          /* %ecx=x&1 */
        cmovne %esi, %ecx          /* if(x&1) %ecx=y*/
        xor    %ecx, %eax          /* product^=x&1 ? y : 0 */
        sub    $0x1, %r8d          /* --i */
        jne    .Lloop1             /* if(i!=0) goto loop1 */
        repz retq                  /* return product */
```

Listing 2: `gf2_multiply` translated by GCC 5.1.1 to `x86-64` with `-O2` (AT&T syntax)

values sum up to a total of either 126 or 157 cycles. Compare this to the numbers obtained from some lax benchmarks carried out on an Intel Core i7-4800MQ processor, which is from the Haswell generation, i.e. pre-Broadwell: one execution of `gf2_multiply` took ≈ 164 cycles.

It should be worth noting that the interchange of the two `xor` operations shortens the critical dependency chain by one instruction. To see how this works, refer to figure 4. Note that, in order to `xor` in the result from the right half of the figure, `(x=>>1)&1 ? y : 0`, first, the register `%ecx` has to be renamed explicitly, to `%ebx` for example. Remember that pure register `mov`'s are usually eliminated by the renamer and thus, if you happen to be on an architecture where `cmovne` takes only one cycle, the blue path in figure 4 might also become critical. The longest dependency chain(s) will now have a length of either $2 + 3 \cdot 31 = 95$ or $2 + 4 \cdot 31 = 126$ cycles , depending on whether `cmovne` has got a latency of one or two cycles. A benchmark carried out on the same pre-Broadwell processor as above, gives $\approx$ 132 cycles per invocation of `gf2_multiply` modified in the way just suggested. Another minor optimization possible is to get rid of the two `and` tests for the lowest bits being set in `product` and `x`. This is possible because the right shifts on these two variables make this information available through the carry flag already. Benchmarks on my Intel Core i7-4800MQ indicate a gain of another four cycles, that is, one invocation of `gf2_multiply` takes a total of approximately 128 cycles. The critical dependency chains are not shortened though. Now, that we came quite close to my theoretical pre-Broadwell minimum of 126 cycles, the big question is whether it is possible to get the above code anywhere near to the minimum of 95 cycles on Broadwell and later. Unfortunately, I haven't got access to such a shiny new processor and can't answer that question reliably.

Finally, if you are in the very specific situation that

- you are reducing modulo the *Castagnoli* polynomial `0x1edc6f41`

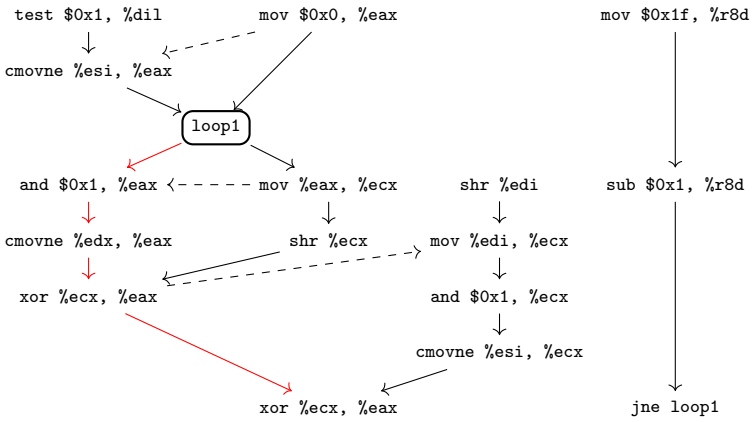

Figure 3: Data dependency graph of `gf2_multiply`. The critical dependency chain is colored in red.

- and that your processor is equipped with both, the `pclmulqdq` instruction from the CMUL instruction set as well as the `crc32` instruction from the SSE 4.2 instruction set,

then you can get rid of the inner loop all together and do the polynomial multiplication modulo $g =$ `0x1edc6f41` as implemented in listing 3. The fact that our polys are stored in reversed order makes it a bit cumbersome to see why the polynomial multiplication works that way. Denote the input polynoms by p and q. Let p_j and q_j be their coefficients for $0 \leq j \leq 31$. For any j beyond that range, define $p_j, q_j \coloneqq 0$. Define the reflected coefficients by $p'_j \coloneqq p_{k-j}$ and $q'_j \coloneqq q_{k-j}$ with $k = 31$. Denote the i'th bit position in `%xmm0` after the execution of the `pclmulqdq` instruction by h'_i with $0 \leq i \leq 63$. According to the *Intel 64 and IA-32 Architectures Software Developer's Manual* 2015, we have

$$h'_i = \sum_{j=0}^{i} p'_j q'_{i-j}$$

Inserting definitions and doing the substitution $j' \coloneqq k - j$, we get

$$\begin{aligned} h'_i &= \sum_{j=0}^{i} p_{k-j} q_{k-(i-j)} \\ &= \sum_{j'=k-i}^{k} p_{j'} q_{2k-i-j'} \end{aligned}$$

Now, there are two cases

1. $i < k$: For $j' \in [0, k-i)$, $k < 2k - i - j'$ and thus, $q_{2k-i-j'} = 0$ holds. Similiary, for $j' \in (k, 2k-i]$, we have $j' > k$ trivially and it follows that $p_{j'} = 0$.
2. $i > k$: We have $k - i < 0$ and thus, $p_{j'} = 0$ for $j' \in [k-i, 0)$. For $j' \in (2k-i, k]$, $2k - i - j' < 0$ holds and it follows that $q_{2k-i-j} = 0$.

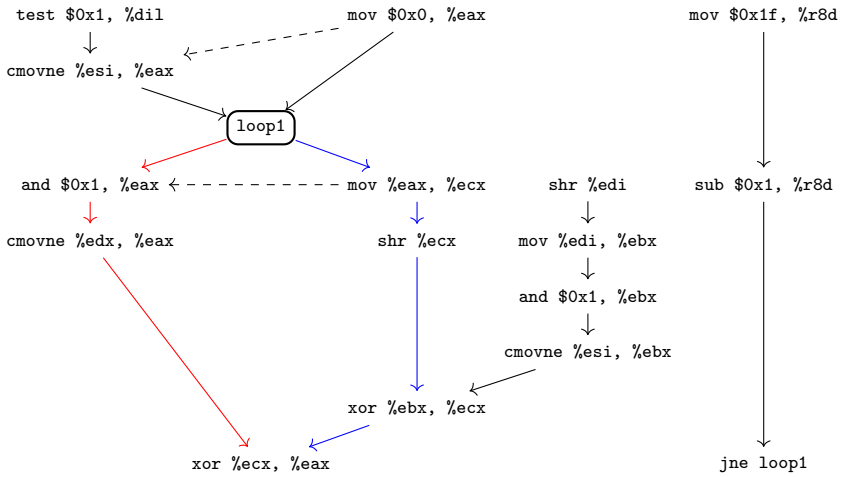

Figure 4: Data dependency graph of `gf2_multiply` with its longest dependency chain optimized. The two critical dependency chains are colored in red and blue respectively.

These considerations taken together allow us to extend (or shrink) the bounds of the summation for h'_i to

$$h'_i = \sum_{j'=0}^{2k-i} p_{j'} q_{2k-i-j'}$$

One readily recognizes that

$$h'_i = (p \cdot q)_{2k-i}$$

This is nothing but the result of $p \cdot q$ stored in reflected order and shifted to the right by one bit. To see this, note that for 64 bits, going from normal to reflected order is equivalent to the replacement $i \to 63 - i = 2k + 1 - i = 2k - (i - 1)$. To compensate for that right shift, the result is shifted left by one position in the above listing. On my Intel Core i7-4800MQ processor, the time needed for one polynomial multiplication has dropped to formidable ≈ 15 cycles now. Unfortunately, the use of the `crc32` instruction restricts the above implementation's applicability to the Castagnolian case. For general g, it might be tempting to still do the polynomial multiplication by means of `pclmulqdq`, followed by a "manual" reduction loop. However, this brings no gain in terms of speed – from the discussion on the general `gf2_multiply` implementation, it should be clear that the critical dependency chain is induced by those reduction steps. Benchmarks confirm that fact: one run of the polynomial multiplication implemented that way took ≈ 134 cycles which is worse than the ≈ 128 cycles already achieved above for the general case.

Returning to the `gf2_multiply` implementation for general g, one might ask oneself what to expect in terms of performance from non-x86 architectures, both superscalar and non-superscalar. It is no surprise, that a register width of ≥ 32 is a big plus when computing 32 bit checksums. Non-x86 architectures with register widths of at least 32 bits are usually RISC ones. In general, an instruction's destination register is specified completely independent from its source registers, obviating the need for register-register moves. Furthermore, conditional moves do exist in some

```
gf2_multiply_castagnoli_clmul_sse42:
        /*
         * Register assignments
         * x        - %edi
         * y        - %esi
        */

        /*
         * Do the polynomial multiplication using pclmuldqdq.
         * The final shift is necessary because the polynomials
         * are stored in reversed order.
         */
        movd      %edi, %xmm0
        movd      %esi, %xmm1
        pclmulqdq $0,%xmm1, %xmm0
        psllq     $1, %xmm0

        /*
         * Use the crc32 insn to reduce the upper half,
         * that is coefficients indexed 32 to 62,
         * modulo the Castagnoli polynomial 0x1edc6f41
         * and xor the lower half in.
         */
        xor       %eax, %eax
        movd      %xmm0, %edi
        crc32     %edi, %eax
        pextrd    $1,%xmm0, %edi
        xor       %edi, %eax
        retq
```

Listing 3: CMUL/SSE 4.2 implementation of `gf2_multiply` for the special case of reducing modulo Castagnoli polynomial (AT&T syntax)

form on any such architecture I know of. In light of this, the initial listing targeted at `x86-64` seems to be quite RISCy already and can be translated 1:1 while getting rid of the now unnecessary register-register moves. Beyond a trivial 1:1 translation, some optimizations might be possible by making use of the particular instructions' capabilities and semantics specific to the architecture at hand:

- Sometimes, logical shift instructions set the status register's carry flag to the value of the last bit shifted out. This potentially allows for the replacement of the two `and` instructions testing the least significant bits of `product` and `x` respectively by `mov`s of a constant zero to a register each. For example, AVR32 and ARM set the carry flag while SPARC v9 and POWER7 don't. On the latter, we could achieve nearly the same for the `x&1` case by using our freedom to force `x` to be negative after the first iteration and do a *arithmetic* right shift instead of a logical one[1].

- The `xor` operation, typically named "`eor`" in RISC, has often got richer capabilities than one is used to on `x86-64` in that it allows for an optional shift to be applied to the second source operand. This allows for the elimination of the right shift for calculating the intermediate value `product>>1`. This feature is available on AVR32 and ARM but not on SPARC v9 and

[1] On POWER7, the conditional move comes in the unique form of the `isel` instruction which has got the semantics of the ternary operator known from C. While saving us from a `mov` of constant zero to a register (or "`li`") within the loop body, we will have to move the carry flag from the exception register `XER` over to the condition register by means of an additional `mcrxr` instruction.

POWER7, for example. On ARM, even the carry flag is set to the bit shifted out last from the second source operand.

- Another capability partly available for `eor` is to conditionally mask its execution as possible on AVR32, ARM and SPARC v9 for example. Conditional masking of `eor` allows for the elimination of a conditional move based on the least significant bit tests on `x` or `product` as well as its possibly associated load of constant zero into a register.

These optimizations are not completely independent: due to data dependencies, it will only be possible to choose either of conditional masking or a right shift of the second source operand at the `eor` instruction involving the value of `product>>1`[2]. Which option to choose depends on the particular architecture at hand and can be answered through benchmarking only. For example, take the ARM instruction set. Choosing the conditional masking option, a possible implementation would look like the one in listing 4. The "`cs`" in the `eorcs` mnemonic is a shorthand for "*c*arry *s*et", that is the

```
1  gf2_multiply:
2          /*
3           * Register assignments
4           * x       - r0
5           * y       - r1
6           * modulus - r2
7           * product - r3
8          */
9          push {r4, lr}
10         lsrs   r0, r0, #1          /* x>>=1 */
11         movcs r3, r1               /* if(old_x&1) product=y */
12         movcc r3, #0               /* if(!(old_x&1)) product=0 */
13         mov   r4, #31              /* i=31 */
14 .Lloop1:
15         lsrs  r3, r3, #1           /* product >>= 1 */
16         eorcs r3, r2, r3           /* product^=old_product&1 ? modulus : 0 */
17         lsrs  r0, r0, #1           /* x>>=1 */
18         eorcs r3, r1, r3           /* product^=old_x&1 ? y : 0 */
19         subs  r4, r4, #1           /* --i */
20         bne   .Lloop1              /* if(i!=0) goto loop1 */
21
22         mov r0, r3
23         pop {r4, pc}
```

Listing 4: Hand-crafted implementation of `gf2_multiply` on ARMv7 with the choice to conditionally mask both `eor` operations

instruction is executed only if the carry flag has been set by the preceding logical right shift `lsrs`. On the other hand, the decision for shifting the second source operand at the first `eor` instruction rather than masking it would manifest itself in an implementation equivalent to listing 5. Since it has got one instruction less in its loop body, listing 4 might look superior to 5 at a first glance. However, at least on the ARM Cortex-A7 processor built into my Raspberry Pi 2 Model B this is not true. Both implementations take ≈ 161 cycles for one single run. More importantly, unrolling the loop six times in implementation 5 lowers the runtime to ≈ 134 cycles only while unrolling does not change anything for listing 4. Due to the lack of publicly available documentation on instruction latencies and such, the explanation to follow required some guesswork and must thus be taken with special care! First of all, according to the *ARM Cortex-A Series Programmer's Guide* 2014, the Cortex-A7 is a partial dual issue processor executing instructions in-order. Unfortunately, it is up to the

[2]On AVR32, this restriction is imposed by the instruction set already.

```
gf2_multiply:
        /*
         * Register assignments
         * x       - r0
         * y       - r1
         * modulus - r2
         * product - r3
        */
        push {r4, lr}
        lsrs   r0, r0, #1         /* x>>=1 */
        movcs r3, r1              /* if(old_x&1) product=y */
        movcc r3, #0              /* if(!(old_x&1)) product=0 */
        mov   r4, #31             /* i=31 */
.Lloop1:
        lsrs  r0, r0, #1          /* x>>=1 */
        movcc ip, #0              /* if(!(old_x&1)) ip=0 */
        movcs ip, r1              /* if(old_x&1) ip=y */
        eors  r3, ip, r3, lsr #1  /* product=(product>>1)^(old_x&1 ? y : 0) */
        eorcs r3, r2, r3          /* product^=old_product&1 ? modulus : 0 */
        subs  r4, r4, #1          /* --i */
        bne   .Lloop1             /* if(i!=0) goto loop1 */

        mov r0, r3
        pop {r4, pc}
```

Listing 5: Hand-crafted implementation of `gf2_multiply` on ARMv7 with the choice to shift the second source operand at the first `eor` operation

reader's interpretation what exactly "partial dual issue" might mean. My personal interpretation is that only certain combinations of instructions can execute in parallel, probably because the second execution unit is not a fully fledged one. The meaning of the attribute "in-order" is clear: a later instruction's execution start never passes an earlier one's. Some hand-crafted research indicates that every instruction involving some shift of any kind has got a latency of two cycles regarding the availability of the final result for later instructions. The updated carry flag seems to be available right after the first cycle though. The source operand involved in shifting must be available at the instruction's first cycle. Summarizing, it looks like the shifting instruction's first cycle is used to do the actual shift while the second cycle is in charge of some further processing like writing the result back or `eor`ing it with another source operand and writing it back afterwards. This is consistent with my observation that an `eor` instruction with a shift count on the second source operand needs its first source operand only at its second cycle. A possible scheduling obeying these rules is given for the first loop iteration and the beginning of the following one for the case of the "shifting `eor`" implementation 5 in table 1. Observe how the first `lsrs` instruction of the loop's second iteration overlaps nicely with the first one's final `eorcs` instruction in cycle 5. Furthermore, the state we have left after cycle 2 starts to repeat after cycle 6 and thus, one iteration of the unrolled loop takes us 4 cycles effectively instead of 5. To finish the discussion on possible implementations of `gf2_multiply` on ARM, let me remark a few points.

- Translating the original C implementation 1 of `gf2_multiply` with GCC gives a runtime of $\approx$ 227 cycles per invocation on my Cortex-A7. Compare this to the 161 cycles needed by the manually optimized implementation 5 without loop unrolling or to the 134 cycles needed by the version with an unrolled loop.

- Loop unrolling comes at a price of additional code size. Whether to pay that price depends on your environment as well as the criticality of a well performing `gf2_multiply`. Of course,

Cycle	1st unit	2nd unit
1	`lsrs r0, r0, #1`	
2		`movcc ip, #0`
3	`movcs ip, r1`	`eors r3, ip, r3, lsr #1`
4		
5	`lsrs r0, r0, #1`	`eorcs r3, r2, r3`
6		`movcc ip, #0`

Table 1: Scheduling of the unrolled loop's instructions among the execution units for the case of the "shifting `eor`" implementation. The instructions belonging to the second iteration are highlighted in yellow.

you are not bound to the unroll count of 6 chosen above. For example, you might want to "move" the preamble `product=x&1 ? y : 0` into the loop by adding another loop iteration – the new iteration count of 32 would allow you to choose any power of two. An unroll count of four seems to be a convenient choice: it trades off between a measured runtime of ≈ 140 cycles and an additional code size of 52 bytes.

- The fact that my Cortex-A7 has got only "partial dual issue" capability obviously did not impose any constraints on the cases at hands. Out of order issuing for $\geq$Cortex-A8 might allow the second `lsrs` instruction to shift up one cycle. In the long run, this might save a few cycles only if the flags register is somehow shadowed. However, I don't have access to such a processor and thus, I can't tell.

- On the non-superscalar Cortex-A5, the implementation 4 conditionally masking both `eors` would certainly be superior to the one incorporating a shift at the first of the two: it needs one cycle less per loop iteration, summing to a total of 253 cycles per invocation of `gf2_multiply`. Note that this situation is very similar to the AVR32 case, although the `lsr` instruction has got a latency of one cycle instead of two there (c.f. *AVR32UC Technical Reference Manual* 2010).

2.2 Comparison of the binary and "linear" exponentiation methods

In this section's beginning, we recognized the fact that $\lfloor \lg n \rfloor + \nu(n)$ polynomial multiplications are needed in order to compute $p \cdot X^n \% g$ by means of the binary exponentiation method. Since we have got an idea concerning the cycles taken by one such polynomial multiplication now, it is straightforward to estimate a total required runtime for each n. We want to compare the performance of the binary exponentiation method with that of simple linear exponentiation, that is repeated shifting and reduction modulo g.

For this purpose, consider the code for `n` iterations of the `SIMPLE` algorithm from Williams 1993 fed with zero bits.

```
1 for(size_t i = 0; i < n; ++i)
2     crc = (crc >> 1) ^ (crc & 1 ? modulus : 0);
```

Listing 6: `SIMPLE` shifting of a CRC by n places to the left

Again, polynomials are stored in reversed order. You can find GCC's output on this for `x86-64` in listing 7. Again, all used instructions with the exception of `cmovne` are tabulated with a latency of

```
        /*
         * Register assignments
         * crc         - %eax
         * modulus     - %esi
         * n           - %rdx
         */
.Lloop1:
        mov     %eax,%ecx  /* %ecx=crc */
        shr     %ecx       /* %ecx=crc>>1 */
        and     $0x1,%eax  /* %eax=crc&1 */
        cmovne %esi,%eax   /* (if %eax=crc&1) %eax=modulus */
        xor     %ecx,%eax  /* %eax=(crc>>1)^
                              (crc&1 ? modulus : 1) */
        sub     $0x1,%rdx
        jne .Lloop1
```

Listing 7: SIMPLE CRC left shifting translated by GCC 5.1.1 to x86-64 with -O2 (AT&T syntax)

one cycle in *Intel 64 and IA-32 Architectures Optimization Reference Manual* 2015. Remember that cmovne needs two cycles on pre-Broadwell processors. Drawing a dependency graph reveals that the longest path is 4 cycles per iteration on Haswell and earlier and benchmarks confirm that: one iteration needs ≈ 4.15 cycles. This result can be pushed down very close to the theoretical limit of 4 by replacing the and-cmovne sequence by a move of constant zero to %eax and conditionally masking the move on the value of the carry flag set by the earlier shr instruction.

Again, if you happen to be in the very specific situation to reduce modulo the Castagnoli polynomial on a SSE 4.2 machine, then you can employ the crc32 instruction to multiply by X^{32} modulo that polynomial very effectively. According to the*Intel 64 and IA-32 Architectures Optimization Reference Manual* 2015, the crc32 instruction has got a latency of 3 cycles and benchmarks show that this instruction embedded in a loop comes very close to that value per iteration. However, since you can only advance by multiples of 32 in n-space by means of crc32, you will have to do the remaining $n\%32$ steps with the simple shift algorithm from above.

Returning from the Castagnolian to the general case, let's have a look on ARM. A hand optimized version of linear shifting can be found in listing 8. Note how the and test on the least

```
        /*
         * Register assignments
         * crc     - r0
         * modulus - r1
         * n       - r3
         */
.Lloop1:
        lsrs r0, #1
        eorcs r0, r0, r1
        subs r3, r3, #1
        bne .Lloop1
```

Listing 8: Hand-crafted simple CRC left shifting for ARMv7

significant bit has been eliminated by using the carry flag set by the lsr instruction. Also, the conditional masking capability of the eor instruction saves a conditional move as well as some associated move of constant zero. Benchmarks on a Cortex-A7 show that one iteration of the above simple shift implementation needs ≈ 3 cycles per iteration. In contrast, the implementation in C from listing 6 translated by GCC 4.9.1 takes ≈ 5 cycles per iteration. Finally, one quickly calculates that

on the non-superscalar Cortex-A5, the above, hand optimized implementation would need 5 cycles per iteration.

By now, we have everything in place to compare the runtime analysises of the binary exponentiation method with the simple linear exponentiation as done exemplarily for a Haswell processor in figure 5. Corresponding plots for the different architectures look qualitatively very similar. What

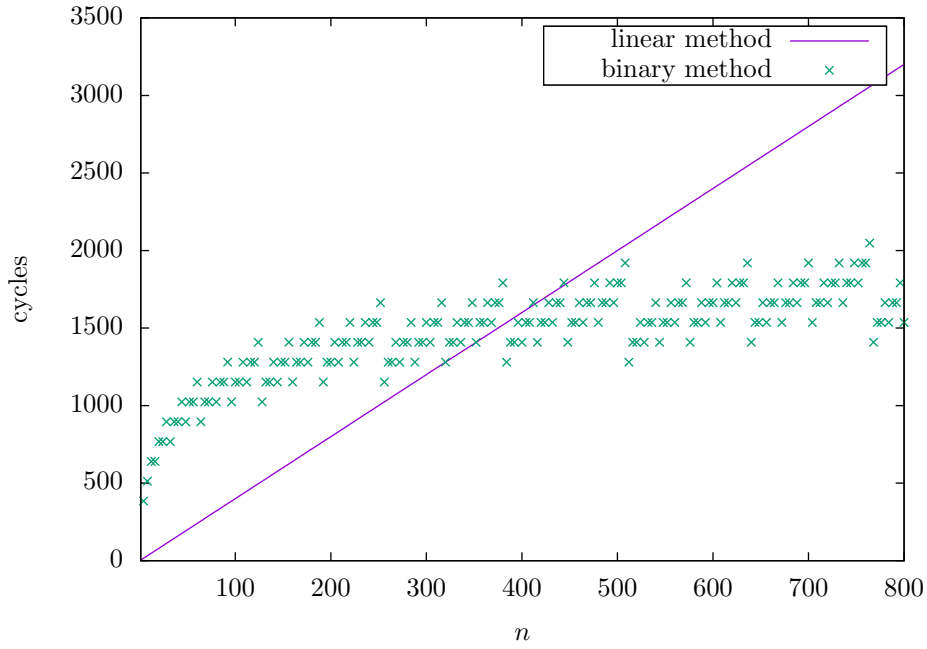

Figure 5: Runtime comparison of binary method versus simple linear exponentiation on an Intel Core i7-4800MQ (Haswell)

immediately catches one's eye is that the binary method performs significantly worse for small n than the linear exponentiation does. The reason becomes apparent soon: in the beginning, the binary method "consumes" only small steps away from the total n, that is small powers of two. Each such step needs a relatively costly polynomial multiplication. This is underlined by the steep growths in the binary method cases for small n in figure 5. The solution out of this dilema is to use a hybrid algorithm: start with the linear exponentiation and switch over to the binary method as soon as we reach a point where its exponential advancements in n-space start to pay off the costly polynomial exponentiation. The function `crc32_generic_shift` from the Linux Kernel (c.f. Spelvin 2014) uses this approach: as of writing, it shifts up to three bytes by means of simple left shift and processes any remaining bytes with the binary exponentiation method.

2.3 Optimizing hybrid exponentiation

Due to the nature of the binary exponentiation, the switch-over point from simple left shifting to binary exponentiation should be a power of two, let's call it 2^ρ for some $\rho \in \mathbb{N}$. In order to

start repeated squaring, we will need the value of $X^{2^\rho}\%g$. Let us assume that g is a compile-time constant. In this case we are able to precompute $X^{2^\rho}\%g$. Turning to the question of a sensible choice for the switch-over point 2^ρ, denote by σ the number of cycles needed for one iteration of the simple left shifting and by μ the ones needed by one polynomial multiplication. Clearly, calculating $p \cdot X^{2^\rho}$ by one polynomial multiplication should not be slower than computing it by simple left shifting. This immediately yields the condition

$$\rho \geq \lg \frac{\mu}{\sigma} \tag{4}$$

However, the choice $\rho \approx \lg \frac{\mu}{\sigma}$ is not necessarily the optimal one. To see this, consider the runtimes of the hybrid exponentiation algorithm drawn in figure 6 for three different choices of ρ each. As

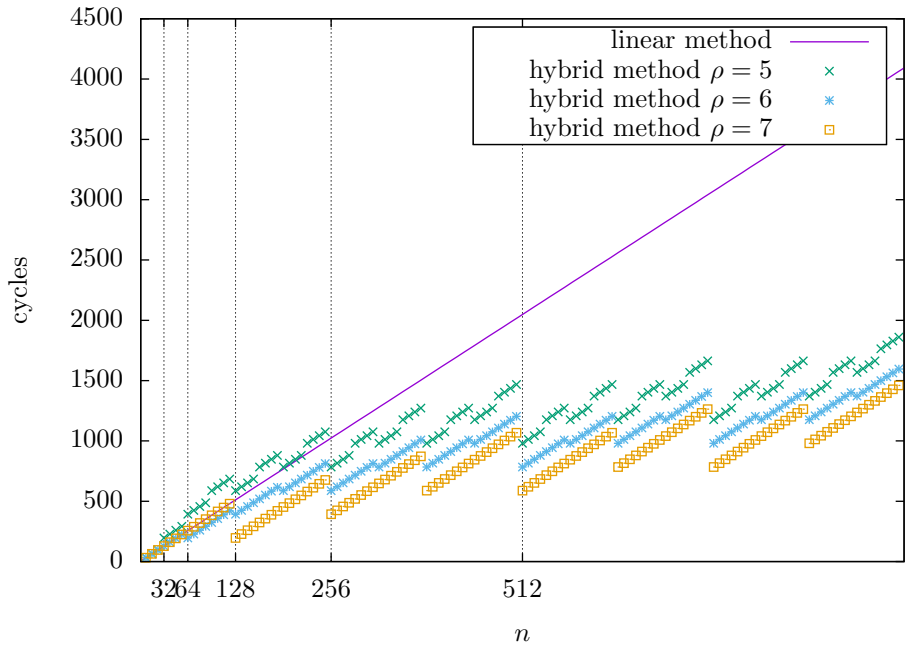

Figure 6: Runtime of hybrid exponentiation in the case of $\sigma = 4$ and $\mu = 196$ for three different choices of ρ. Only every eighth point is drawn.

you can see there, the choice of $\rho = 5 < \lg(\frac{196}{4}) \approx 5.6$ is too small: the hybrid exponentiation's runtime exceeds that of the linear method for small n. On the other hand, setting $\rho = 6$ ensures that hybrid exponentiation is always better than the linear method. This choice is still not the optimal one though: albeit hybrid exponentiation with $\rho = 7$ is a little bit slower than the one with $\rho = 6$ in the range of $n \in [64, 127]$, it outperforms the latter significantly for $n \geq 128$.

In order to answer the question of an optimal ρ systematically, the average runtime requirements have to be calculated. Assume that our n is bounded by $n < 2^{\Lambda+1}$ for some $\Lambda \in \mathbb{N}$. For the sake of generality, we only consider those n for which $n\%2^\kappa = 0$ holds for some fixed $\kappa \in \mathbb{N}, 0 \leq \kappa \leq \Lambda$. For example, if we always shifted in units of bytes, we would set $\kappa = 8$. In appendix A, it is shown

that the average number of cycles required by such a hybrid exponentiation is equal to

$$\bar{t}_\kappa(\rho,\Lambda) = \frac{1}{2}\sigma(2^\rho - 2^\kappa) + \frac{1}{2}\mu\left(3(\Lambda-\rho)-1\right) + \mu 2^{\rho-\Lambda} \tag{5}$$

To find its minimum, differentiate with respect to ρ:

$$\frac{\partial \bar{t}}{\partial \rho} = \frac{\sigma \ln 2}{2} 2^\rho - \frac{3}{2}\mu + \mu \ln(2) 2^{\rho-\Lambda} \tag{6}$$

Equating this first derivative to zero yields

$$\rho_0 = \lg\left(\frac{3}{\ln 2\left(\frac{\sigma}{\mu} + 2^{-\Lambda+1}\right)}\right) \tag{7}$$

Of course, since you can not begin binary exponentiation at a non-integral power of two, you have to choose either of $\lfloor\rho_0\rfloor$ and $\lceil\rho_0\rceil$ by comparison of what $\bar{t}$ evaluates to at these points. For example, setting $\sigma = 4$, $\mu = 196$ and $\Lambda = \lg(1024) - 1 = 9$ as done in figure 6 gives $\rho_0 \approx 7.48$. The integral neighbors $\rho = 7$ and $\rho = 8$ inserted into the average runtime (5) give values of 793 and 804 cycles respectively and thus, one would choose $\rho = 7$.

Since the original equation (5) is valid only for $\rho \leq \Lambda + 1$, its minimum ρ_0 as given by (7) is meaningful only if $\rho_0 \leq \Lambda + 1$. One readily gets that equation (7) is valid for all

$$\Lambda \geq \lg\left(\left(\frac{3}{2\ln 2} - 2\right)\frac{\mu}{\sigma}\right) \approx -2.6 + \lg\frac{\mu}{\sigma} \tag{8}$$

only. For any Λ below that bound you would choose not to do any binary exponentiation at all: simple left shifting outperforms it in all cases of interest. The same holds for the case that $\rho_0 = \Lambda+1$: by definition of our region of interest, we always have $n < 2^{\Lambda+1}$ and if we are told to begin binary exponentiation at $n \geq 2^{\Lambda+1}$, we effectively shall not start it at all. On the other hand, if $\rho_0 \leq \Lambda$ holds, then $\lceil\rho_0\rceil \leq \Lambda$ also and binary exponentiation is beneficial at least for the $n \geq 2^\Lambda$. Inserting this condition into equation (7), we conclude that if

$$\Lambda \geq \lg\left(\left(\frac{3}{\ln 2} - 2\right)\frac{\mu}{\sigma}\right) \approx 1.2 + \lg\frac{\mu}{\sigma} \tag{9}$$

holds, then we will definitely end up entering the binary exponentiation stage for at least the $n \geq 2^\Lambda$. If you happen to encounter the corner case $\Lambda < \rho_0 < \Lambda + 1$, then you have to compare $\bar{t}_\kappa(\lfloor\rho_0\rfloor = \Lambda, \Lambda)$ to $\bar{t}_\kappa(\lceil\rho_0\rceil = \Lambda+1, \Lambda)$ in order to decide whether to do some binary exponentiation stage for the $n \geq 2^\Lambda$. One quickly calculates that

$$\bar{t}_\kappa(\Lambda,\Lambda) < \bar{t}_\kappa(\Lambda+1,\Lambda) \Leftrightarrow \Lambda > \lg\frac{\mu}{\sigma} \tag{10}$$

in conformity with (4). To summarize: hybrid exponentiation is superior to simple left shifts if and only if $\Lambda > \lg\frac{\mu}{\sigma}$.

It is quite interesting to have a look at how the minima ρ_0 vary with the upper averaging bound $\Lambda+1$. To this end, the dependence of $\bar{t}_0$ on ρ is visualized for several choices of a fixed Λ in figure 7. Note that in accordance to (8), none of $\bar{t}_0$'s minima is located within the "forbidden region" for

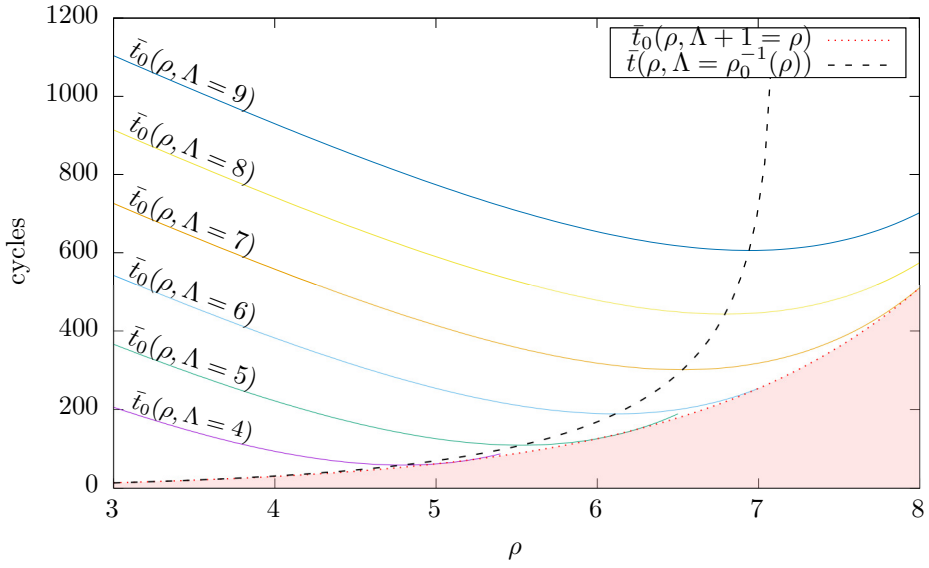

Figure 7: Minima of $\bar{t}$ with respect to ρ. The case of $\sigma = 4$ cycles and $\mu = 128$ cycles is drawn for different choices of Λ. The black, dashed line connects all points $(\rho, \bar{t}(\rho, \Lambda = \rho_0^{-1}(\rho)))$, that is all the minima of $\bar{t}$ for the different Λ. Within the region above the red, dashed line $\rho \leq \Lambda + 1$ holds, while this assumption is violated below that boundary.

any of the chosen Λs: $-2.6 + \lg \frac{\mu}{\sigma} = -2.6 + 5 = 2.4 \leq \Lambda$ holds for all choices of Λ made. For $\Lambda = 4$ and $\Lambda = 5$, we do not have $\Lambda > 5 = \lg \frac{\mu}{\sigma}$ and thus, the condition (10) is not satisfied and one would not do any binary exponentiation at all. There is an important observation to make in figure 7: for smaller values of Λ, the minima ρ_0 grow rapidly until they eventually stabilize at the limit

$$\lim_{\Lambda \to \infty} \lg \left(\frac{3}{\ln 2 \left(\frac{\sigma}{\mu} + 2^{-\Lambda+1} \right)} \right) = \lg \left(\frac{3}{\ln 2} \frac{\mu}{\sigma} \right) \approx 7.1$$

of equation (7).

Let us have a deeper look at the dependency of (7) on Λ. Obviously, a larger Λ implies a larger optimal ρ – there is some sort of tension between starting the binary exponentiation as early as possible for small n and starting it later for larger n. This might look a little bit puzzling at first: after all, next to some additional processing, large n have to go through the very same procedure as the smaller ones on the average and thus, optimizing the case of small n also brings some benefit for the larger ones. Unfortunately, this is only half of the truth as the "additional processing" required for the larger n grows with their distance from the smaller ones. To make this concrete, consider some n with $\lfloor \lg n \rfloor < \Lambda + 1$. In making the transition $\Lambda \to \Lambda + 1$, we will encounter the original n as well as a new n' of the form

$$n' = 2^{\Lambda+1} + n$$

From equation (16), it follows that the runtime needed for that n' is equal to

$$t(n') = t(n) + \mu(\Lambda + 1 - \max(\lfloor \lg n \rfloor, \rho)) + \mu \tag{11}$$

In addition to the processing required for the original n, we have to do $\Lambda + 1 - \max(\lfloor \lg n \rfloor, \rho)$ repeated squarings in order to obtain $X^{2^{\Lambda+1}}$ from $X^{2^{\max(\lfloor \lg n \rfloor, \rho)}}$. Furthermore, one extra polynomial multiplication is required for the additional high order bit set at position $\Lambda + 1$ in n'. Summing the second term in equation (11) over all n with $\lfloor \lg n \rfloor < \Lambda + 1$ and $n \% 2^\kappa = 0$ yields

$$(1 - \frac{1}{2} 2^{\rho - \Lambda}) 2^{\Delta + 2 - \kappa}$$

To obtain the increment of $\bar{t}$ due to the second term in equation (11) during the transition $\Lambda \to \Lambda+1$, we have to divide by $2^{\Lambda+2-\kappa}$, the total number of $n < 2^{\Lambda+2}$ with $n \% 2^\kappa = 0$. We finally get

$$\frac{1}{2^{\Lambda+2-\kappa}} \sum_{\substack{n=0 \\ n\%2^\kappa=0}}^{2^{\Lambda+1}-1} \mu(\Lambda + 1 - \max(\lfloor \lg n \rfloor, \rho)) = \mu(1 - \frac{1}{2} 2^{\rho - \Lambda}) \tag{12}$$

The increment of $\bar{t}$ due to the third term in equation (11) during the transition $\Lambda \to \Lambda + 1$ is simply $\frac{1}{2}\mu$ since the high order bit corresponding to $2^{\Lambda+1}$ is set half of the times. To summarize: due to the repeated squarings, increments in Λ lead to increments in $\bar{t}$ which in turn increase with the distance of Λ from ρ. It is this non-linearity in $\Lambda - \rho$ which makes the minima ρ_0 as given by (7) depend on Λ. However, this non-linearity quickly converges to almost-linearity, i.e. to almost constant increments of $\bar{t}$ with Λ, for medium values of $\Lambda - \rho$ already. Note that this convergence is carried forward to a convergence of the minima given by equation (7).

3 Splitting checksums

Checksum splitting is the reverse process of combining checksums: you have got a whole data chunk's checksum as well as the one of either its leading or trailing subchunk and you want to calculate the other part's checksum from the two known ones. Depending on which subchunk's checksum you have got already, there are two cases which are depicted in figures 8 and 9. For the second case, note that

$$p_2 \% g = (p_1 \cdot X^n + p_2 - p_1 \cdot X^n) \% g = (p_1 \cdot X^n + p_2) \% g - p_1 \cdot X^n \% g \tag{13}$$

Since you already know from the previous section 2 how to compute $p_1 \cdot X^n \% g$ from a known value of $p_1 \% g$, the problem has already been solved. Note that this method can be used to remove seeds from checksums.

The first case is harder, since up to now, we can only calculate the left shifted checksum

$$p_1 \cdot X^n \% g = (p_1 \cdot X^n + p_2 - p_2) \% g = (p_1 \cdot X^n + p_2) \% g - p_2 \% g \tag{14}$$

The question is whether it is possible to recover $p_1 \% g$ from a known value of $p_1 \cdot X^n \% g$. Algebra tells us that this is indeed possible if the generator polynomial is not divisible by X. For a derivation, please refer to appendix B.3. To be more specific, there exists some (unique) polynomial X_g^{-1} with $\deg X_g^{-1} \leq \deg g$ such that

$$(p_1 \cdot X \% g) \cdot X_g^{-1} \% g = p_1 \% g$$

for all polynomials p_1. To find X_g^{-1}, one may write

$$X_g^{-1} \cdot X + \gamma \cdot g = 1 \tag{15}$$

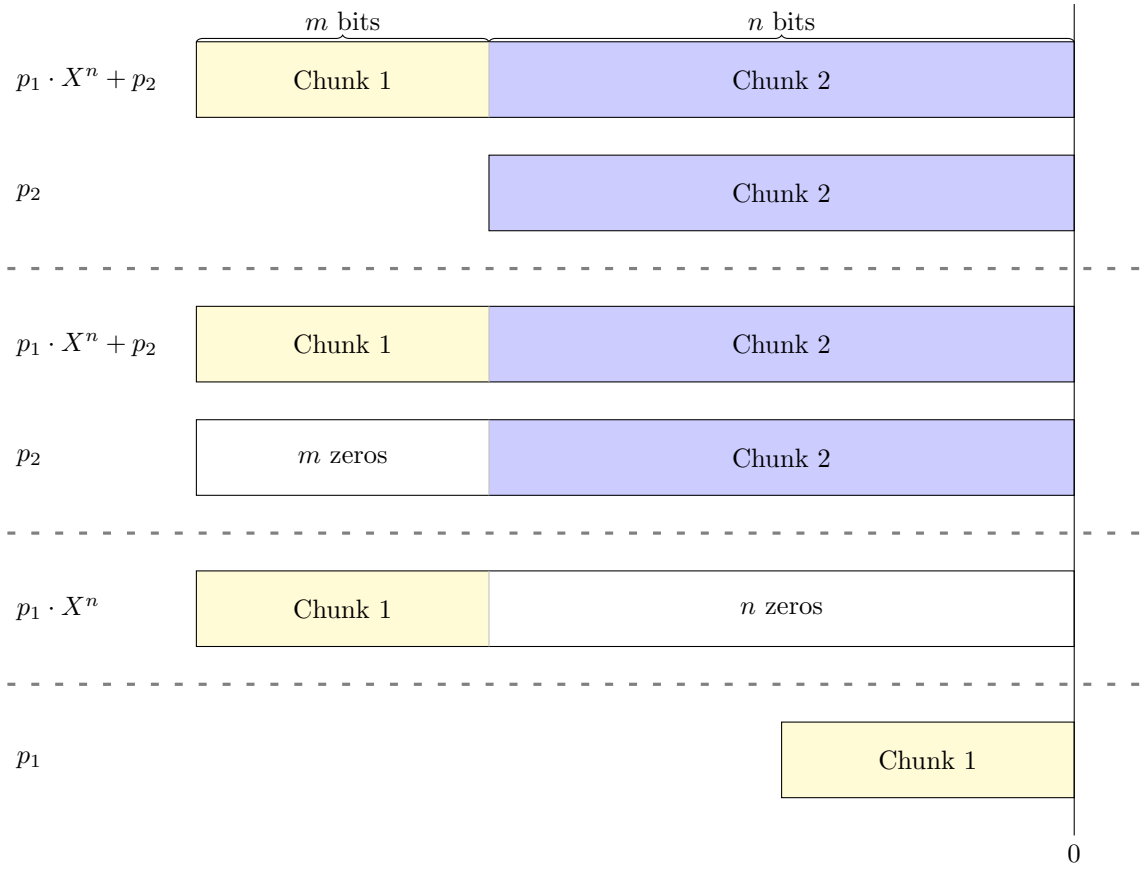

Figure 8: Splitting of two data chunk's checksums: first case

and use the Extended Euclid's algorithm to determine X_g^{-1} (as well as the γ not of further interest here). On the Extended Euclid's algorithm, please refer to Knuth 1997, section 4.5.2. We are now in the position to calculate the first chunk's checksum

$$p_1 \% g = (p_1 \cdot X^n \% g) \cdot (X_g^{-1})^n \% g$$

by means of multiplying the value obtained through (14) n times by X_g^{-1}. However, from section 2.2 we know that polynomial multiplications are relatively expensive compared to a few simple left shifts. If there existed something such as a simple *right* shift, then we could adapt the hybrid exponentiation method from section 2.3 to our case.

Simple right shifting Consider the value of $p \cdot X \% g$, that is the CRC of p shifted left by one position. The generator polynomial g has got its zeroth order coefficient set to one because it is not divisible by X. Now, since we always shift in zeros for simple left shifting, we can tell whether g would have been subtracted (`xor`ed in) during simple left shifting of $p \% g$, i.e. in calculating $p \cdot X \% g$ from $p \% g$, simply by looking at the zeroth coefficient of $p \cdot X \% g$:

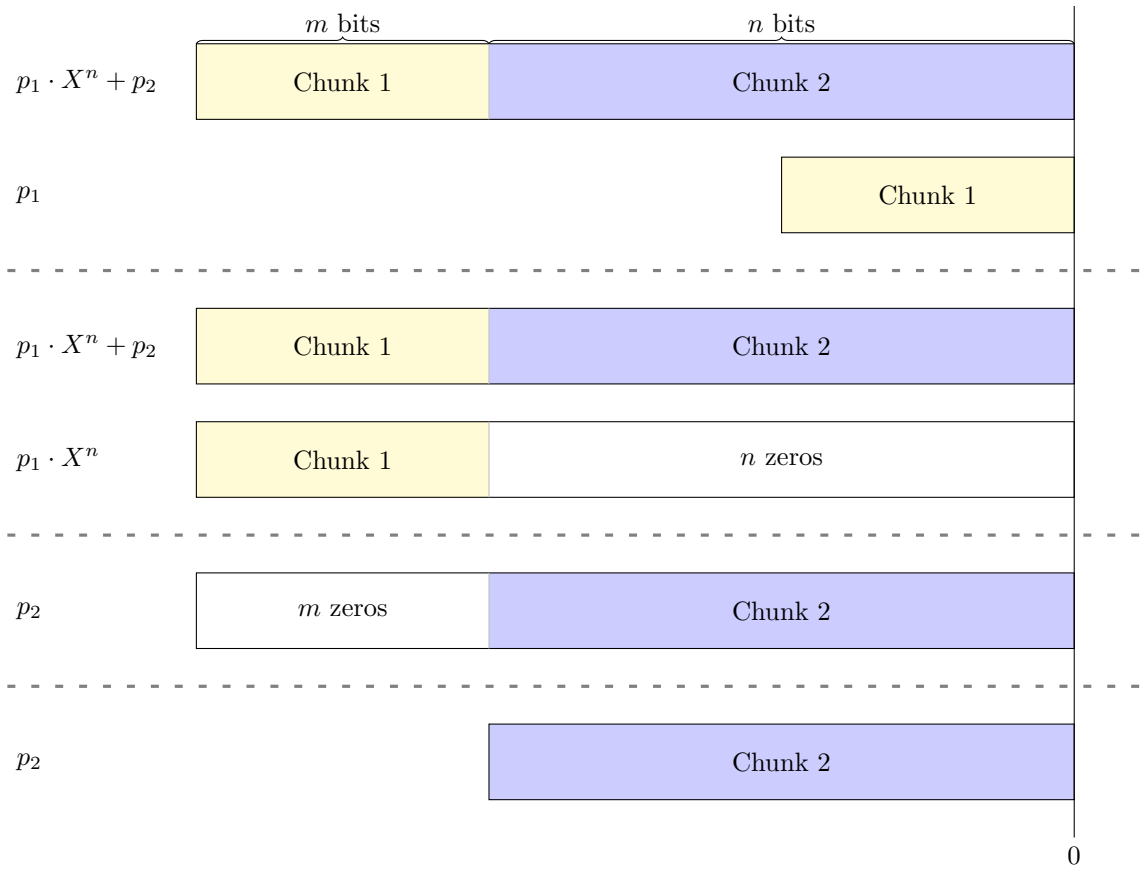

Figure 9: Splitting of two data chunk's checksums: second case

- $(p \cdot X\%g)_0 = 0$: g would not have been subtracted in calculating $p \cdot X\%g$ from $p\%g$. We have $p \cdot X\%g = (p\%g) \cdot X$ and thus,
$$p\%g = (p \cdot X\%g)/X$$
- $(p \cdot X\%g)_0 = 1$: g would have been subtracted in calculating $p \cdot X\%g$ from $p\%g$. We have $p \cdot X\%g = (p\%g) \cdot X - g$ and thus,
$$p\%g = ((p \cdot X\%g) + g)/X$$

In analogy to listing 6 for simple left shifting, these considerations suggest the following implementation of simple right shifting for the case of 32 bit CRCs:

```
1 for(size_t i = 0; i < n; ++i)
2     crc = (crc << 1) ^ (crc & 0x80000000) ? (modulus << 1 | 0x1) : 0);
```

Listing 9: SIMPLE shifting of a CRC by n places to the right

Again, polynomials are stored in reversed order. Remember that the bit corresponding to the coefficient of X^{32} in g is usually not stored in `modulus` since it is always set to one. For our purposes, we need to restore that missing bit. On the other hand, we don't need the bit corresponding to X^0 in g and may shift it out safely. Finally, since polynomial multiplications commute, observe that we may use the simple right shifts of listing 9 in order to compute X_g^{-1} or any power $(X_g^{-1})^n$ thereof: simply initialize `crc` with a "reversed" one, `0x80000000` (corresponding to an initial value of $p \cdot X^n \% g = 1$). Depending on the circumstances, this might be easier than using the Extended Euclid's algorithm.

Application of CRC splitting: verification of an encapsulated data's checksum One of the CRC splitting's applications is the verification of a network packet's checksum which has been encapsulated within the packet of another protocol. For what follows, it is assumed, that both, the encapsulating as well as the encapsulated protocol use the same CRC type. An example for such an encapsulation are Infiniband packets encapsulated within Ethernet packets, also known as "RoCE" (c.f. *Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1, Annex A16: RDMA over Converged Ethernet (RoCE)* 2010 and *Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1, Annex A17: RoCEv2* 2014). Both, the encapsulated Infiniband packet's invariant checksum (ICRC) field as well as the Ethernet frame's checksum (FCS, "frame check sequence") are computed using the same type of CRC, but over different ranges of the whole Ethernet packet. The point is, that upon receipt of a packet, the Ethernet checksum is almost always verified by the network adapter, i.e. very efficiently in hardware. Furthermore, many network adapter allow access to an received packet's FCS from software. Thus, we may assume that we have got the whole data chunk's checksum. Usually, the extra data covered by the FCS, that is the Ethernet headers at the packet's head as well as the Infiniband's ICRC at its end, are much smaller in size than the payload covered by the ICRC. It is likely more efficient to strip (split) off the extra chunks from the FCS by means of the methods discussed in this section than to recompute the ICRC over the whole Infiniband packet. There are two possible approaches:

- We could either explicitly recover the ICRC from the FCS and compare it to the one stored within the Infinband packet
- or we could combine the transmitted ICRC with the manually computed CRC of the Ethernet headers preceding the encapsulated Infiniband packet. The outcome is then used as the seed for a final CRC calculation over the fields following the Infiniband packet[3]. The results from appendix B.3 tell us that if and only if the result of this final CRC calculation is equal to the FCS, then the ICRC has been successfully validated.

The latter strategy saves us from doing the right shifting after removing the fields following the encapsulated Infiniband packet. Please note that I glossed over some details like "invariant fields" in the computation of the ICRC as well as CRC seeds. Incorporating these details is straight-forward though.

A The hybrid exponentiation method's average runtime

Again, denote by σ the cycles needed for a simple left shift by one position and by μ the runtime needed by one polynomial multiplication modulo g. Let $\kappa, \rho, \Lambda \in \mathbb{N}$ with $\kappa \leq \rho < \Lambda + 1$ be fixed.

[3] only the ICRC is stored there

We want to compute $p \cdot X^n \% g$ for some given $n \in \mathbb{N}$ with $n < 2^{\Lambda+1}$ and $n\%\kappa = 0$. The input polynomial p is not of any importance here. Up to, but not including 2^ρ, we will do simple left shifts and from 2^ρ on we will use the binary exponentiation method to proceed up to the final n. More specifically, write

$$n = \left\lfloor \frac{n}{2^\rho} \right\rfloor 2^\rho + n\%2^\rho$$

The second term will be handled by simple left shifts, the first term by binary exponentiation. For the binary exponentiation, we will need

$$\left\lfloor \lg \left\lfloor \frac{n}{2^\rho} \right\rfloor \right\rfloor = \lfloor \lg n \rfloor - \rho$$

polynomial multiplications to obtain the powers $X^{2^{\rho+1}}, X^{2^{\rho+2}}, \ldots, X^{2^{\left\lfloor \lg \left\lfloor \frac{n}{2^\rho} \right\rfloor \right\rfloor}}$ of X by means of repeated squaring. In addition, we need to multiply some of these powers into the final product as many times as there are ones set in the binary representation of $\left\lfloor \frac{n}{2^\rho} \right\rfloor$, i.e. $\nu\left(\left\lfloor \frac{n}{2^\rho} \right\rfloor\right)$ times. The total runtime required for that n in question will thus be equal to

$$t(n) = \begin{cases} \sigma n & n < 2^\rho \\ \sigma(n\%2^\rho) + \mu\left(\lfloor \lg n \rfloor - \rho + \nu\left(\left\lfloor \frac{n}{2^\rho} \right\rfloor\right)\right) & n \geq 2^\rho \end{cases} \tag{16}$$

To obtain the mean value $\bar{t}$ of the runtime averaged over all n, observe that there are $2^{\Lambda+1-\kappa}$ such n meeting our requirements. We thus have

$$\begin{aligned} 2^{\Lambda+1-\kappa}\bar{t} &= \sum_{\substack{n=0 \\ n\%2^\kappa=0}}^{2^{\Lambda+1}-1} t(n) \\ &= \sum_{\substack{n=0 \\ n\%2^\kappa=0}}^{2^\rho-1} t(n) + \sum_{\substack{n=2^\rho \\ n\%2^\kappa=0}}^{2^{\Lambda+1}-1} t(n) \\ &= \sum_{\substack{n=0 \\ n\%2^\kappa=0}}^{2^\rho-1} t(n) + \sum_{\lambda=\rho}^{\Lambda} \sum_{\substack{\lfloor \lg n \rfloor=\lambda \\ n\%2^\kappa=0}} t(n) \end{aligned} \tag{17}$$

The first part is easy: since all $n \in \mathbb{N}$ with $n\%2^\kappa = 0$ are of the form $j2^\kappa$ for some $j \in \mathbb{N}$, we have

$$\sum_{\substack{n=0 \\ n\%2^\kappa=0}}^{2^\rho-1} t(n) = \sigma \sum_{j=0}^{2^{\rho-\kappa}-1} j2^\kappa = \frac{1}{2}\sigma 2^\rho (2^{\rho-\kappa} - 1) \tag{18}$$

In order to obtain the second part in equation (17), let us first calculate the total sum of the runtimes over all allowed n with $\lfloor \lg n \rfloor = \lambda$ for some fixed $\lambda \in \mathbb{N}$ with $\rho \leq \lambda$. Combinatorics yields

$$\sum_{\substack{\lfloor \lg n \rfloor=\lambda \\ n\%2^\kappa=0}} t(n) = 2^{\lambda-\rho}\sigma \sum_{j=0}^{2^{\rho-\kappa}-1} j2^\kappa + 2^{\lambda-\kappa}\mu(\lambda-\rho) + 2^{\rho-\kappa}\mu \sum_{\omega=0}^{\lambda-\rho} \binom{\lambda-\rho}{\omega}(\omega+1)$$

To see this, note that

- The number of n with $\lfloor \lg n \rfloor = \lambda$ and n modulo 2^ρ equal to some given value is $2^{\lambda-\rho}$.
- The conditions $\lfloor \lg n \rfloor = \lambda$ and $n\%2^\kappa = 0$ allow for a total of $2^{\lambda-\kappa}$ combinations.
- For $\lfloor \lg n \rfloor = \lambda$ fixed and $n\%2^\kappa = 0$, there are exactly $2^{\rho-\kappa}$ combinations such that $\left\lfloor \frac{n}{2^\rho} \right\rfloor$ is equal to some given value. Furthermore, $\binom{\lambda-\rho}{\omega}$ such values of $\left\lfloor \frac{n}{2^\rho} \right\rfloor$ have got exactly $\omega + 1$ ones set in their binary representations.

By using the well-known summation formulas and the fact that $\sum_{\omega=0}^{\Omega} \binom{\Omega}{\omega}\omega = \Omega 2^{\Omega-1}$, the sum can be simplified to

$$\sum_{\substack{\lfloor \lg n \rfloor = \lambda \\ n\%2^\rho = 0}} t(n) = \frac{1}{2}2^\lambda \sigma(2^{\rho-\kappa} - 1) + \frac{1}{2}2^{\lambda-\kappa}\mu(3(\lambda - \rho) + 2) \tag{19}$$

Summing (19) over λ from ρ to Λ and substituting the result together with (18) in equation (17) yields

$$\bar{t}_\kappa(\rho, \Lambda) = \frac{1}{2}\sigma(2^\rho - 2^\kappa) + \frac{1}{2}\mu\left(3(\Lambda - \rho) - 1\right) + \mu 2^{\rho-\Lambda} \tag{5}$$

after solving for $\bar{t}$ and simplifying further.

By a lucky chance, the formulas used to sum (19) over λ, namely the ones to calculate $\sum_{\lambda=\rho}^{\Lambda} 2^\lambda$ and $\sum_{\lambda=\rho}^{\Lambda} \lambda 2^\lambda$, also hold for the case that $\rho = \Lambda + 1$ for which these sums are equal to zero by definition. This allows us to extend the validity of the final result in equation (5) from $\kappa \le \rho < \Lambda + 1$ to $\kappa \le \rho \le \Lambda + 1$, i.e. the possibility of doing no binary exponentiation step at all is also covered.

B Inverting CRC left shifts

This section deals with the question whether the operation of CRC left shifting

$$p \cdot X\%g$$

has got an inverse, i.e. whether there exists an $\tilde{X}$ such that

$$(p \cdot X\%g) \cdot \tilde{X}\%g = p\%g \tag{20}$$

for any polynomial p. It turns out that this is true for all real world choices of the generator polynomial g and we will denote $\tilde{X}$ by X_g^{-1}.

B.1 (Very) brief intro to the language of algebra

For those readers not accustomed to the algebraic speak employed within this section, I summarized the basic terms and definitions in a not so strict manner. For details, please refer to some algebra textbook, Lang 2002 for example.

B.1.1 Algebraic spaces and homomorphisms

Algebraic spaces The definitions of the algebraic spaces of interest here obey some kind of hierarchy in the sense that they equip some set with a law of composition, each definition in turn requesting more "structure" from it. We will review these definitions in the order induced by this hierarchy, starting with the most basic one.

A *monoid* $(M, \star)$ is a set M together with a law of composition

$$\star : M \times M \to M$$

that is associative and for which there exists a neutral element $e \in M$. Neutral means, that it leaves any element $m \in M$ unchanged under the action of the law of composition: $\star(m, e) = \star(e, m) = m$. For $a, b \in M$, one also writes $a \star b$ for the evaluation $\star(a, b)$ of the map $\star$ at $(a, b) \in M \times M$. Depending on the context, one often writes the law of composition either additively or multiplicatively, i.e. instead of the "$\star$"-symbol, either a "$+$" or a "$\cdot$" is used. In the former case, a "0" denotes the unit element, in the latter case a "1" is written. Example for a monoid: the non-negative integers $\mathbb{N}$ with their law of addition.

Written multiplicatively, a *group* $(G, \cdot)$ is a monoid for which there exist inverses, that is for every $g \in G$ there exists a $g' \in G$ such that $g \cdot g' = 1$. The notation "g^{-1}" is almost always used in place of g' for multiplicatively written groups. If the law of composition is commutative, the group is called *abelian*, usually written additively and g's inverse is denoted by "$-g$". Example for a group: the integers $\mathbb{Z}$.

A *ring* $(A, +, \cdot)$ is a set A with two laws of composition, such that $(A, +)$ is an abelian group and $(A, \cdot)$ is a group, the latter not necessarily being abelian. The two laws of composition shall "interact" in such a way, that the well known law of distributivity is satisfied:

$$\begin{aligned} a \cdot (b + c) &= a \cdot b + a \cdot c \\ (b + c) \cdot a &= b \cdot a + c \cdot a \end{aligned}$$

If the law of multiplication is also commutative, the ring is called *abelian*. Example for a ring: the integers $\mathbb{Z}$ together with the well known law of multiplication – for positive $n \in \mathbb{Z}$, $n \cdot a$ is defined to be equal to a added n times to 0. Another example of particular interest here are the rings of polynomials. Let A be an arbitrary ring. By $A[X]$, we denote the ring consisting of polynomials with coefficients in A.

Finally, a commutative ring $(K, +, \cdot)$ is called a *field* if $(K \setminus \{0\}, \cdot)$ is a group, that is, there do not only exist additive inverses, but also multiplicative ones. Example for a field: the rational numbers $\mathbb{Q}$.

Homomorphisms Loosely speaking, homomorphisms are maps between two algebraic spaces that are compatible with their algebraic structure in that they "commute" with the law of composition. More specifically, let $(M, \star_M)$ and $(N, \star_N)$ be monoids. A map $f : M \to N$ is a called *homomorphism of monoids*, if

$$\forall m, m' \in M : f(m \star_M m') = f(m) \star_N f(m')$$

Observe that the law of composition on the left hand side is that in M while that on the right hand side is that in N. Furthermore, it is required that the neutral element gets mapped to the target monoid's neutral element. A similar definition applies for the *homomorphisms of groups*.

For rings $(A, +_A, \cdot_A)$ and $(B, +_B, \cdot_B)$, a map $f : A \to B$ is called a *homomorphism of rings*, if f is a homomorphism of monoids for both, A and B taken as additive monoids $(A, +_A)$ and $(B, +_B)$ as well as multiplicative monoids $(A, \cdot_A)$ and $(B, \cdot_B)$. Finally, if the rings are actually fields, a homomorphism of rings is sometimes called a *homomorphism of fields*.

B.1.2 Factor groups and residue class rings

Subgroups and cosets Let G be a multiplicatively written group and $H \subset G$ be a subset containing the unit. If H is closed[4] under multiplication as well as taking the inverse, it is called a *subgroup* of G.

Let $x \in H$. By $xH \subset G$ we denote the subset of G obtained by multiplying any element in H from the left with x. Subsets of the form $xH \subset G$ for a subgroup $H \subset G$ are termed *left cosets* of H. Similiarly for right cosets Hx of H.

Observe, that $x = x \cdot 1 \in xH$. Since H is a subgroup and thus closed under taking inverses, if we have $y \in xH$, then $yH = xH$. It follows, that any two cosets xH and zH are either disjoint or equal: If $y \in xH \cap zH$, then $xH = yH = zH$. Putting it together, any element of G is contained in exactly one left coset of H, that is to say, the left cosets of H make up the classes of a partition of G. Remember that any partition of a set into disjoint classes induces an *equivalence relation* on that set and vice versa.

Finally, x is called a *representative* of the coset xH. From what has just been said, it should be clear, that any member of xH would do as its representative.

Factor groups and the canonical homomorphism Again, let $H \subset G$ be a subgroup of G. It is called a *normal* subgroup if $xH = Hx$ holds[5] for all $x \in G$. Because of normality, the set of cosets is closed under multiplication: for any $x, y \in G$, we have

$$(xH)(yH) = x(Hy)H = x(yH)H = xy(HH) = xyH$$

Furthermore, $(x^{-1}H)(xH) = H$, i.e. inverses exist. Thus, the multiplication in G induces a law of multiplication on the set of cosets of a normal group H, turning that set of cosets into a group again. This group is called a *factor group* of G by H and denoted by G/H.

For any homomorphism f taking G to some other group G', it can be shown that the set $f^{-1}(1)$, made up of those elements of G taken to the neutral element in G' by f, is a normal subgroup of G. It is called the *kernel* of f and denoted by $\ker f$. Consider the map

$$\begin{aligned} \phi : G &\to G/H \\ x &\mapsto xH \end{aligned} \tag{21}$$

which assigns each element of G to the coset of H it is a member of. It can be shown that this is a homomorphism of groups. It is called the *canonical* homomorphism. The important fact is that every f whose kernel contains H can always be written as a sequence of maps

$$\begin{array}{ccc} G & \xrightarrow{\phi} & G/H \\ & \searrow^{f} & \downarrow{\scriptstyle f_*} \\ & & G' \end{array} \tag{22}$$

[4] A set is closed under a certain operation, if the result of the operation applied to any of the set's members is a member of that set again.

[5] Observe how this requirement is always satisfied if G is commutative.

i.e. $f = f_* \circ \phi$ for some unique homomorphism $f_* : G/H \to G'$. f is said to *factor* through the canonical homomorphism ϕ.

It should be clear by now that if H is normal, the corresponding cosets carry much more structure than the mere equivalence relation discussed in the previous paragraph. For this reason, we call two members $x, y \in G$ of the same coset not only equivalent, but *congruent modulo* H and write

$$x \equiv y \mod H$$

Ideals and residue class rings Consider a ring A. Let $\mathfrak{a}$ be a subgroup of A viewed as an additive group. Since A viewed as an additive group is abelian, $\mathfrak{a}$ is normal. Instead of writing the additive cosets of $\mathfrak{a}$ multiplicatively as done in the preceding paragraph, we now write them additively. That is for $x \in A$, we write $x + \mathfrak{a}$ in order to denote the additive coset of $\mathfrak{a}$ containing x. Now, if, besides $\mathfrak{a}$ being a subgroup of the additive group of A,

$$A\mathfrak{a} \subset \mathfrak{a}$$

holds, $\mathfrak{a}$ is called an ideal. Furthermore, if $\mathfrak{a}$ is an ideal, instead of calling the $x+\mathfrak{a}$ "additive cosets", we favor the term *residue class* from now on.

Ideals play the role for rings that normal subgroups play for groups: namely the set of residue classes carries a natural ring structure. That set of residue classes together with the ring structure is called a *residue class ring* and denoted by $A/\mathfrak{g}$.

For the case of rings, the kernel $\ker f$ of a homomorphism $f : A \to A'$ is defined to be the set of those elements in A which are taken to 0 by f. It can be shown that $\ker f$ is an ideal in A. As it is the case for factor groups, there is a canonical homomorphism

$$\begin{aligned} \phi : A &\to A/\mathfrak{a} \\ x &\mapsto x + \mathfrak{a} \end{aligned} \tag{23}$$

through which every homomorphism f with $\mathfrak{a} \subset \ker f$ factors. That is, there exists a unique $f_* : A/\mathfrak{a} \to A'$ such that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{\phi} & A/\mathfrak{a} \\ & \searrow^{f} & \downarrow{\scriptstyle f_*} \\ & & A' \end{array} \tag{24}$$

Finally note that there is a class of ideals of particular importance in this text: if $g \in A$, then Ag is clearly an ideal. Ideals of this form are called *principal* and Ag is said to be *generated by* g and is often denoted by (g).

B.2 CRC computation in the language of algebra

From now on, let $g \in \mathbb{F}_2[X]$ be some fixed element in the polynomial ring over the field $\mathbb{F}_2$ containing just two elements, a 0 and a 1. We denote the prinicipal ideal generated by g in $\mathbb{F}_2[X]$, i.e. $\mathbb{F}_2[X]g$ by $\mathfrak{g}$.

I assume that you are familiar with the fact, that any data bitstream of length $l \in \mathbb{N}, l > 0$ may be interpreted as a polynomial $p \in \mathbb{F}_2[X]$ of degree $l - 1$ by reinterpreting the individual bits as coefficients in $\mathbb{F}_2$ to powers of X.

The basic idea behind CRC is to add some redundancy to the transmitted data by sending along with p a canonical representative $p' \in \mathbb{F}_2[X]$ of the residue class modulo g the p resides in: $p \in p' + \mathfrak{g}$. The receiver verifies whether the received data bitstream $\tilde{p} \in \mathbb{F}_2[X]$ is a member of the residue class modulo $\mathfrak{g}$ identified by the received representative $\tilde{p}'$. The probability of false negatives, i.e. transmission errors going undetected, as well as criteria for sensible choices of the generator polynomial g are not the topic covered here – there is plenty of material about this in literature as well as on the web. Here, we take g for granted and want to play around with CRC computations.

For the question which representative of p's residue class modulo g to transmit, the answer is clear: from a mathematical standpoint, any of its members, even p itself, would do. In order to preserve bandwidth, the one with the shortest bitstream representation, i.e. that of smallest degree is chosen. The Euclidean division algorithm (c.f. Lang 2002, IV, §1), that is repeatedly subtracting suitable polynomial multiples of g from p, tells us, that it is always possible to choose a representative p' of $p + \mathfrak{g}$ with $\deg p' < g$. It is easy to see that this choice is unique: if $p'_1 + \mathfrak{g} = p'_2 + \mathfrak{g}$, then $p'_1 - p'_2 \in \mathfrak{g} = \mathbb{F}_2[X]g \Leftrightarrow p'_1 - p'_2 = qg$ for some $q \in \mathbb{F}_2[X]$. Thus, either $q = 0$ or $\deg(p'_1 - p'_2) \geq \deg g$. The latter possibility is in contradiction to $\deg p'_1, \deg p'_2 < \deg g$. Now, consider the canonical homomorphism

$$\begin{aligned} \phi : \mathbb{F}_2[X] &\to \mathbb{F}_2[X]/\mathfrak{g} \\ p &\mapsto p + \mathfrak{g} \end{aligned} \tag{25}$$

together with the homomorphism of abelian groups (not rings!) mapping each residue class modulo $\mathfrak{g}$ to its unique representative of minimum degree:

$$\begin{aligned} \%_* : \mathbb{F}_2[X]/\mathfrak{g} &\to \mathbb{F}_2[X] \\ p + \mathfrak{g} &\mapsto p' \in p + \mathfrak{g} \quad \text{with minimum degree} \end{aligned} \tag{26}$$

First note that a residue class's representative is chosen within that class itself, that is, we have $\forall p \in \mathbb{F}_2[X] : \%_*(p + \mathfrak{g}) \in p + \mathfrak{g}$. It immediately follows that

$$\phi \circ \%_* = \mathrm{id}_{\mathbb{F}_2[X]/\mathfrak{g}} \tag{27}$$

In particular, $\%_*$ is injective. The reason why $\%_*$ is *not* a homomorphism of rings is that in general, for $p + \mathfrak{g}, q + \mathfrak{g} \in \mathbb{F}_2[X]/\mathfrak{g}$ arbitrary, $\%_*(p + \mathfrak{g})\%_*(q + \mathfrak{g})$ is not the representative of $p \cdot q + \mathfrak{g}$ of *minimum degree*.

In view of the algorithms to discuss, what we really want to have is the homomorphism of abelian groups defined by $\% \coloneqq \%_* \circ \phi$.

$$\begin{array}{ccc} \mathbb{F}_2[X] & \xrightarrow{\phi} & \mathbb{F}_2[X]/\mathfrak{g} \\ & \searrow^{\%} & \downarrow{\scriptstyle \%_*} \\ & & \mathbb{F}_2[X] \end{array} \tag{28}$$

Due to (27), it immediately follows from the definitions that

$$\phi \circ \% = \phi \tag{29}$$

From 29 and the fact that ϕ is a homomorphism of rings, we get

$$\begin{aligned}\%(\%(p)\cdot q) &= \%_* \circ \phi(\%(p)\cdot q)\\ &= \%_*(\phi(\%(p)\cdot q))\\ &= \%_*(\phi(\%(p))\cdot\phi(q))\\ &= \%_*(\phi(p)\cdot\phi(q))\\ &= \%_*(\phi(p\cdot q))\\ &= \%(p\cdot q)\end{aligned} \tag{30}$$

As a special case, setting $q = 1$ yields idempotency of %: $\% \circ \% = \%$.

Finally let us introduce an operator like notation for %, analogous to the one found in most programming languages:

$$\forall p \in \mathbb{F}_2[X] : p\%g \coloneqq \%(p)$$

B.3 The units in $\mathbb{F}_2[X]/\mathfrak{g}$

For which $q \in \mathbb{F}_2[X]$ is the "multiplication homomorphism" of abelian groups

$$\begin{aligned}\mathbb{F}_2[X]/\mathfrak{g} &\to \mathbb{F}_2[X]/\mathfrak{g}\\ p+\mathfrak{g} &\mapsto (p+\mathfrak{g})\cdot(q+\mathfrak{g})\end{aligned} \tag{31}$$

injective? Or, by definition of injectivity and $p \equiv 0 \mod g \Leftrightarrow p \in \mathfrak{g}$, for which $q \in \mathbb{F}_2[X]$ does

$$\forall p \in \mathbb{F}_2[X] : p\cdot q \in \mathfrak{g} \Rightarrow p \in \mathfrak{g}$$

hold?

Since $\mathbb{F}_2$ is a field, the ring $\mathbb{F}_2[X]$ is *factorial* (c.f. Lang 2002, IV, §1), which just means that each element has got a unique representation as a product of *irreducibles*, also called *primes* in that context. Now, since $\forall u \in \mathbb{F}_2[X]$

$$u \in \mathfrak{g} \Leftrightarrow \exists h \in \mathbb{F}_2[X] : u = h\cdot g$$

by definition of $\mathfrak{g} = \mathbb{F}_2[X]g$, we conclude that the factorization of q into primes must not have any factors in common with g's factorization in order for the multiplication homomorphism defined above to be injective: if $p\cdot q$ is a multiple of g, it must contain all factors of g. These factors do not divide q, hence they have to divide p. If on the other hand, q does contain some factors of g, we can always find $p \in \mathbb{F}_2[X], p \notin \mathfrak{g}$ such that $p\cdot q \in \mathfrak{g}$ by setting it equal to the product of factors of g not dividing q. Thus, the multiplication homomorphism (31) is injective if and only if the greatest common divisor (g.c.d.) of q and g is 1.

Having answered the introductionary question, we can go further. Again, since $\mathbb{F}_2$ is a field, the polynomial ring $\mathbb{F}_2[X]$ is even *principal*[6]. *Principal* means that every ideal in the ring can be written as $\mathbb{F}_2[X]u$ for some $u \in \mathbb{F}_2[X]$. Now, consider the ideal $(q,g) \coloneqq \mathbb{F}_2[X]q + \mathbb{F}_2[X]g$ generated by q and g in $\mathbb{F}_2[X]$. Since $\mathbb{F}_2[X]$ is principal, this ideal (q,g) must be equal to $\mathbb{F}_2[X]u$ for some $u \in \mathbb{F}_2[X]$. The key point is, that u is equal to the g.c.d. of q and g by Lang 2002, II, §5, Proposition 5.1. Now, if that g.c.d. is 1, we have

$$(q,g) \coloneqq \mathbb{F}_2[X]q + \mathbb{F}_2[X]g = \mathbb{F}_2[X]1 = \mathbb{F}_2[X]$$

[6]Actually, the ring's factoriality follows from its principality, c.f.Lang 2002, II, §5.

Especially, since $1 \in \mathbb{F}_2[X]$, we can write the unit element as a linear combination of q and g with factors in $\mathbb{F}_2[X]$:

$$\exists \rho, \gamma \in \mathbb{F}_2[X] : \rho \cdot q + \gamma \cdot g = 1 \tag{32}$$

The significance of this is that $\rho + \mathfrak{g}$ is an inverse of $q + \mathfrak{g}$ in $\mathbb{F}_2[X]/\mathfrak{g}$:

$$(\rho + \mathfrak{g}) \cdot (q + \mathfrak{g}) = \rho \cdot q + \mathfrak{g} = 1 - \gamma \cdot g + \mathfrak{g} = 1 + \mathfrak{g}$$

An element of a ring having an inverse is also called an *unit*. Hence, putting it all together, the units of $\mathbb{F}_2[X]/\mathfrak{g}$ are exactly the elements $q + \mathfrak{g}$ for which the g.c.d. of q and g is 1.

In the context of CRCs, a popular choice for the generator polynomial g is to make it irreducible[7]. This means, that for any $q \in \mathbb{F}_2[X]$, the g.c.d. of q and g is either g or 1. From what has been said above, this means that all elements of $\mathbb{F}_2[X]/\mathfrak{g}$ but $0 + \mathfrak{g}$ are units, i.e. that $\mathbb{F}_2[X]/\mathfrak{g}$ is a field. However, be warned that g is not always chosen to be irreducible for every CRC. The generator polynomial g is always chosen such that it is not divisible by X though and thus, the special case of $q = X^n + \mathfrak{g}$ for any $n \in \mathbb{N}$ is a unit in $\mathbb{F}_2[X]/\mathfrak{g}$ for all real world CRCs.

C The standard algorithms reformulated

C.1 SIMPLE algorithm

Let $p \in \mathbb{F}_2[X]$. p corresponds to our possibly zero augmented data bitstream of length $\deg p + 1$. Our goal is to compute the bitstream's CRC $p \% g$ for some arbitrary but fixed generator polynomial $g \in \mathbb{F}_2[X]$ with $W \coloneqq \deg g$.

Set $p_0 \coloneqq p$. Let $N_0 \in \mathbb{N}$ with $\deg p \leq N_0$. Write

$$p_0 = p_0' \cdot X^{N_0 - W + 1} + p_0''$$

for some $p_0', p_0'' \in \mathbb{F}_2[X]$ with $\deg p_0' < W$ and $\deg p_0'' < N_0 - W + 1$. Take $i \in \mathbb{N}, i \leq N_0 - W$ and assume that $p_i \in \mathbb{F}_2[X]$ has already been defined such that

$$p_i \equiv p_0 \mod g \tag{33}$$

$$\deg p_i \leq N_i \coloneqq N_0 - i \tag{34}$$

Caution: the first relation above is a "congruence modulo g" relation, not an equality relation: $p \equiv h \mod g \Leftrightarrow \exists q \in \mathbb{F}_2[X] : p - h = qg$. Write

$$p_i = p_i' \cdot X^{N_i - W + 1} + p_i''$$

for some $p_i', p_i'' \in \mathbb{F}_2[X]$ with

$$\begin{aligned} \deg p_i' &< W \\ \deg p_i'' &< N_i - W + 1 \end{aligned}$$

Observe that we can rewrite p_i into

$$p_i = (p_i' \cdot X + p_{i,N_i - W}'') \cdot X^{N_i - W} + (-p_{i,N_i - W}'' X^{N_i - W} + p_i'') \tag{35}$$

[7] $\mathbb{F}_2[X]$ being factorial, this is equivalent (c.f. Lang 2002, II, §5) to saying that $\mathfrak{g}$ is *prime* in $\mathbb{F}_2[X]$, another algebraic term you might hear in the context of CRCs.

with $p''_{i,N_i-W} \in \mathbb{F}_2$ being the coefficient of X^{N_i-W} in p''_i. Inductively define

$$\begin{aligned} p'_{i+1} &\coloneqq (p'_i \cdot X + p''_{i,N_i-W})\%g && (36)\\ p''_{i+1} &\coloneqq -p''_{i,N_i-W}X^{N_i-W} + p''_i && (37)\\ p_{i+1} &\coloneqq p'_{i+1} \cdot X^{N_{i+1}-W+1} + p''_{i+1} && (38) \end{aligned}$$

with $N_{i+1} \coloneqq N_i - 1 = N_0 - (i+1)$. Adopting the convention that $\deg 0 = -1$, the verification of the induction assumptions for $i+1$ is a straight forward task.

The connections between p'_i and the "summing register" in Williams 1993 on the one hand and p''_i and the remaining tail of possibly augmented data on the other hand, are drawn easily. Observe that the rewriting (35) of p_i corresponds to the shift step in the SIMPLE algorithm fromWilliams 1993 while the modulo operation on the parenthesized polynomial of degree $\leq W$ in (36) corresponds to the conditional XOR operation. Finally, for $i = N_0 - W + 1$, we have $p_{N_0-W+1} = p'_{N_0-W+1}$, i.e we are done and have got the final checksum stored in the "summing register".

C.2 Table driven implementation

Instead of handling one term (bit) at a time, we could as well attempt to handle $K \in \mathbb{N}$ terms at a time. Usually, $K = 8$ is chosen. The rewriting or shift step (35) as done in the SIMPLE algorithm now becomes

$$\begin{aligned} p_i = (p'_i \cdot X^K + \sum_{j=0}^{K-1} p''_{i,N_i-W-j}X^{K-1-j}) \cdot X^{N_i-W-K+1}\\ +(-\sum_{j=0}^{K-1} p''_{i,N_i-W-j}X^{K-1-j} \cdot X^{N_i-W-K+1} + p''_i) \end{aligned} \quad (39)$$

The first parenthesized term has got degree $< W + K$, i.e. at most $W + K - 1$. In calculating its modulus with respect to g analogous to (36), observe that the modulo operation simply subtracts a suitable multiple of g in $\mathbb{F}_2[X]$ from the original. What exactly this "suitable multiple" is, depends solely on the original's terms of degree $\geq W$, i.e. the multiple is chosen such that, after subtraction, all those terms of degree $\geq W$ vanish. Including the terms with coefficient zero, there are exactly K of them. Determination of the suitable multiple can now be done in either of two ways:

1. Calculating it at runtime, possibly by utilizing your CPU's quantum coprocessor.
2. By looking up the combination of the K leading terms in a precomputed table.

Due to the lack of a widespread quantum coprocessor deployment, the table driven implementation uses the second approach.

Finally, for completeness, the new inductive definition analogous to (36)-(38) reads as

$$\begin{aligned} p'_{i+1} &\coloneqq (p'_i \cdot X^K + \sum_{j=0}^{K-1} p''_{i,N_i-W-j}X^{K-1-j})\%g && (40)\\ p''_{i+1} &\coloneqq -\sum_{j=0}^{K-1} p''_{i,N_i-W-j}X^{K-1-j} \cdot X^{N_i-W-K+1} + p''_i && (41)\\ p_{i+1} &\coloneqq p'_{i+1} \cdot X^{N_{i+1}-W+1} + p''_{i+1} && (42) \end{aligned}$$

with $N_{i+1} \coloneqq N_0 - (i+1)K$ while the inductive assumptions (33) and (34) remain unchanged.

C.3 Direct table algorithm

Let $p \in \mathbb{F}_2[X]$ denote the polynomial corresponding to the original, non-augmented bitstring. As described in Williams 1993, the "table driven implementation" has got an implementation artifact in that it actually calculates the remainder of

$$\left(\sum_{j=0}^{W-1} 0X^{W-j} \cdot X^{\deg p}\right) + p$$

modulo g, i.e. it prepends p with W terms of zero coefficients. This is made kind of explicit by initializing the summing register with a zero. Furthermore, the augmentation of the original message p with W zero bits requires some special handling. To begin with, we set

$$p_0 \coloneqq p \cdot X^W$$

and define

$$\begin{aligned} p'_0 &\coloneqq 0 \\ p''_0 &\coloneqq p_0 \end{aligned}$$

Again, choose $N_0 \in \mathbb{N}$ such that $\deg p \leq N_0$ holds. Note that in what follows, the N_i do not include the additional W zero terms from the multiplication of p by X^W! The step formerly known as the "shift" step now reads as

$$\begin{aligned} p_i = (p'_i \cdot X^K + \sum_{j=0}^{K-1} p''_{i,N_i+W-j} X^{W+K-1-j}) \cdot X^{N_i-K+1} \\ +(-\sum_{j=0}^{K-1} p''_{i,N_i+W-j} X^{W+K-1-j} \cdot X^{N_i-K+1} + p''_i) \end{aligned} \tag{43}$$

Observe how the K leading polynomial terms of the two summands in the first parentheses overlap now. This stems from the fact, that the K leading bits from the data tail are not shifted into the summing register from the right, but XORed into its high order bits prior to lookup. The inductive definitions now become

$$p'_{i+1} \coloneqq (p'_i \cdot X^K + \sum_{j=0}^{K-1} p''_{i,N_i+W-j} X^{W+K-1-j}) \% g \tag{44}$$

$$p''_{i+1} \coloneqq (-\sum_{j=0}^{K-1} p''_{i,N_i+W-j} X^{W+K-1-j} \cdot X^{N_i-K+1} + p''_i) \tag{45}$$

$$p_{i+1} \coloneqq p'_{i+1} X^{N_i-K+1} + p''_{i+1} \tag{46}$$

with $N_{i+1} \coloneqq N_0 - (i+1)K$. Note that we always have $\deg p''_i \leq N_i + W$ and that the W low order terms of p''_i are always zero.

To draw the connections to the "direct table algorithm" as implemented in Williams 1993, the summing register still corresponds to p'_i while the yet unhandled data tail times X^W corresponds to p''_i. However, the interpretation of the summing register's contents has changed: at any point, it contains the checksum of the data handled so far *as if it had been augmented by W bits*. This explains the statement

> Note: The initial register value for this algorithm must be the initial value of the register for the previous algorithm fed through the table four times.

in Williams 1993: in order to glue any seed intended for use with the "direct algorithm" or equivalently, its "table driven implementation" to the "direct table algorithm" at hand, augment it by W zero bits, i.e. multiply it by X^W.

References

ARM Cortex-A Series Programmer's Guide (2014). Version 4.0. ARM DEN0013D. ARM Limited. URL: http://infocenter.arm.com/help/topic/com.arm.doc.den0013d/index.html.

AVR32UC Technical Reference Manual (2010). Version F. 32002F-03/2010. Atmel Corporation. URL: http://www.atmel.com/Images/doc32002.pdf.

Intel 64 and IA-32 Architectures Optimization Reference Manual (2015). Order Number: 248966-031. Intel Corporation. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

Intel 64 and IA-32 Architectures Software Developer's Manual (2015). *Volume 2: Instruction Set Reference A-Z*. Order Number: 325383-056US. Intel Corporation. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

Knuth, Donald E. (1997). *Seminumerical Algorithms*. In: *The Art of Computer Programming*. Third Edition. Vol. 2. Boston: Pearson Education. ISBN: 0-201-89684-2.

Lang, Serge (2002). *Algebra*. Revised third edition. New York: Springer Science+Business Media Inc. ISBN: 0-387-95385-x.

Spelvin, George (2014). *[PATCH 1/3] lib: crc32: Greatly shrink CRC combining code*. Post to the Linux Kernel Mailinglist. URL: https://lkml.kernel.org/r/20140530053505.18155.qmail@ns.horizon.com.

Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1, Annex A16: RDMA over Converged Ethernet (RoCE) (2010). InfiniBand Trade Association. URL: http://www.infinibandta.org/content/pages.php?pg=technology_download.

Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1, Annex A17: RoCEv2 (2014). InfiniBand Trade Association. URL: http://www.infinibandta.org/content/pages.php?pg=technology_download.

Williams, Ross N. (1993). *A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS*. Version 3. URL: http://www.ross.net/crc/download/crc_v3.txt.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.