CRC

CRC (Cyclic Redundancy Check, циклический избыточный код) - один из видов контрольной суммы или хэша. Основная идея состоит в том, чтобы представить файл, как одну огромную строку бит, и поделить ее на некоторое число; оставшийся в результате остаток и есть CRC. Но в алгоритме "деления"все сложения и вычитания заменяются на операцию "исключающего или"(xor).

Вычисление CRC

Для вычисления CRC нам необходимо выбрать делитель, который с этого момента мы будет называть полиномом. Степень полинома (W – Width) – это номер позиции его старшего бита, следовательно, полином 1001 будет иметь степень '3', а не '4'. Обратите внимание, что старший бит всегда должен быть равен 1, следовательно, после того, как вы выбрали степень полинома, вам необходимо подобрать лишь значения младших его битов.

Если вы хотите вычислять CRC для последовательности бит, вам следует убедиться, что обработаны все биты. Поэтому в конце последовательности необходимо добавить W нулевых бит:

```
Полином                                = 10011, степень W=4
Последовательность бит + W нулей = 110101101 + 0000

10011/1101011010000\110000101 (о частном мы не заботимся)
      10011|||||||| -
      -----||||||||
       10011|||||||
       10011|||||||   -
       -----|||||||
        00001||||||
        00000||||||    -
        -----||||||
         00010|||||
         00000|||||     -
         -----|||||
          00101||||
          00000||||      -
          -----||||
           01010|||
           00000|||       -
           -----|||
            10100||
            10011||        -
            -----||
             01110|
             00000|         -
             -----|
              11100
              10011          -
              -----
               1111 -> остаток -> то есть CRC!
```

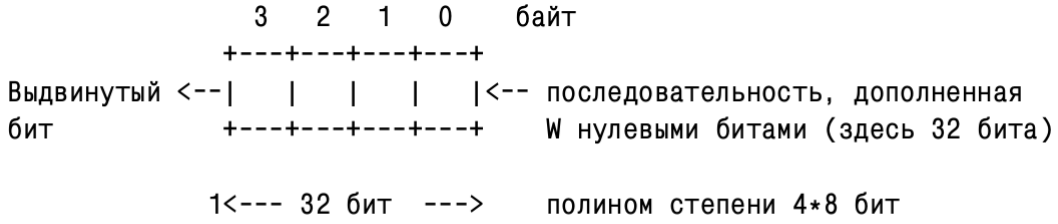
Здесь необходимо упомянуть 2 важных момента:

1) Операция XOR выполняется только в том случае, когда старший бит последовательности равен 1, в противном случае мы просто сдвигаем ее на один бит влево.

2) Операция XOR выполняется только с младшими W битами, так как старший бит всегда в результате дает 0.

Обзор табличного алгоритма

Понятно, что алгоритм, основанный на битовых вычислениях очень медленен и неэффективен. Было бы намного лучше проводить вычисления с целыми байтами. Но в этом случае мы сможем иметь дело лишь с полиномами, имеющими степень, кратную 8 (то есть величине байта). Давайте представим себе такие вычисления на основе полинома 32 степени (W=32).



(Схема 1)

Здесь показан регистр, который используется для сохранения промежуточного результата вычисления CRC, и который я будут называть регистром CRC или просто регистром. Когда бит, выдвинутый из левой части регистра, равен 1, то выполняется операция "Исключающее ИЛИ"(XOR) содержимого регистра с младшими W битами полинома (их в данном случае 32). Фактически мы выполняем описанную выше операцию деления.

А что, если (как я уже предложил) сдвигать одновременно целые группы бит. Рассмотрим пример вычисления CRC длиной 8 бит со сдвигом 4 бит одновременно. До сдвига регистр равен: 10110100

Затем 4 бита выдвигаются с левой стороны, тогда как справа вставляются новые 4 бита. В данном примере выдвигается группа 1011, а вдвигается – 1101.

Таким образом мы имеем:

текущее содержимое 8 битного регистра (CRC): 01001101

4 старших бита, только что выдвинутые слева: 1011

используемый полином (степень W=8): 101011100

Теперь вычислим новое значение регистра:

```
Старш. Регистр
биты
---- --------
1011 01001101          Старшие биты и регистр
1010 11100    ⊕ (*1)   Полином складывается по XOR, начиная с 3 бита
                       старшей группы (естественно, он равен 1)
-------------
0001 10101101          Результат операции XOR
```

В старшей группе в позиции 0 у нас остался еще один бит, равный 1

```
0001 10101101          Предыдущий результат
   1 01011100 ⊕ (*2)   Полином складывается по XOR, начиная с 0 бита
                       старшей группы (естественно, он тоже равен 1)
-------------
0000 11110001          Результат второй операции XOR
^^^^
```

Теперь все 4 бита старшей группы содержат нули, поэтому нам больше нет необходимости выполнять в ней операцию XOR.

То же самое значение регистра могло быть получено, если операнд (*1) сложить по XOR с операндом (*2). Это возможно в результате ассоциативного свойства этой операции – $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

```
1010 11100             Полином, прикладываемый к 3 биту старшей группы
   1 01011100 ⊕ (*2)   Полином, прикладываемый 0 биту старшей группы
-------------
1011 10111100   (*3)   Результат операции XOR
```

Если теперь применить к исходному содержимому регистра операцию XOR со значением (*3), то получим:

```
1011 10111100
1011 01001101 ⊕        Старшие биты и регистр
-------------
0000 11110001
```

Видите? Тот же самый результат! Значение (*3) для нас достаточно важно, так как в случае, когда старшая группа бит равна 1011, младшие W=8 бит всегда будут равны 10111100 (естественно, для данного примера). Это означает, что мы можем заранее рассчитать величины XOR-слагаемого для каждой комбинации старшей группы бит. Обратите внимание, что эта старшая группа в результате всегда превратится в 0.

Вернемся снова к схеме 1. Для каждой комбинации битов старшей группы (8 битов), выдвигаемых из регистра, мы можем рассчитать слагаемое. Получим таблицу из 256 (или 2^8) двойных слов (32 бита).

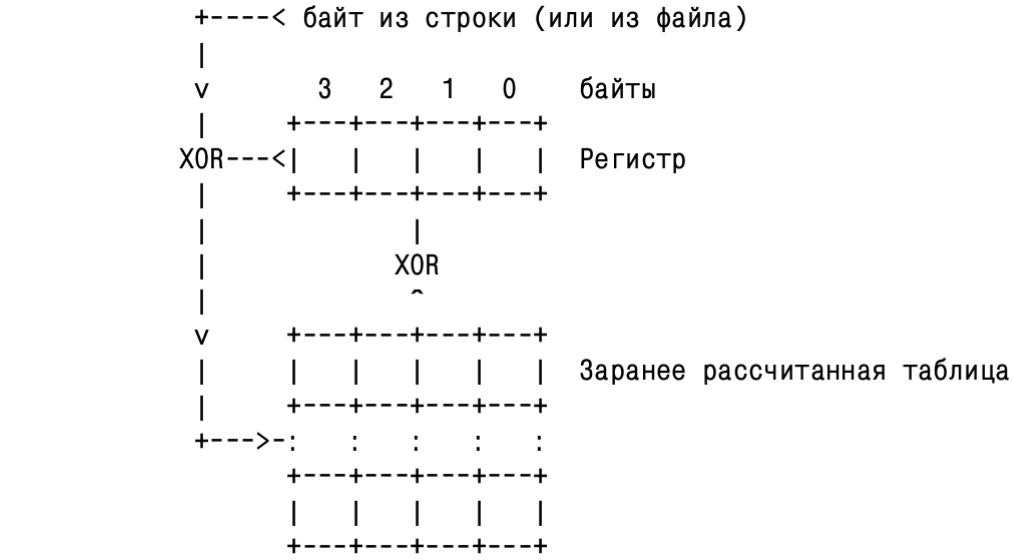
На мета-языке этот алгоритм можно записать следующим образом:

```
While (в строке имеются необработанные биты)
  Begin
    Top = top_byte of register ;
    Register = (сдвиг Register на 8 бит влево) OR (новые биты из строки);
    Register = Register XOR табличное_значение[Top];
  End
```

Прямой табличный алгоритм

Описанный выше алгоритм можно оптимизировать. Нет никакой необходимости постоянно "проталкивать"байты строки через регистр. Мы можем непосредственно применять операцию XOR байтов, выдвинутых из регистра, с байтами обрабатываемой строки. Результат будет указывать на позицию в таблице, значение из которой и будет в следующем шаге складываться в регистром.

Давайте взглянем на схему алгоритма (схема 2):



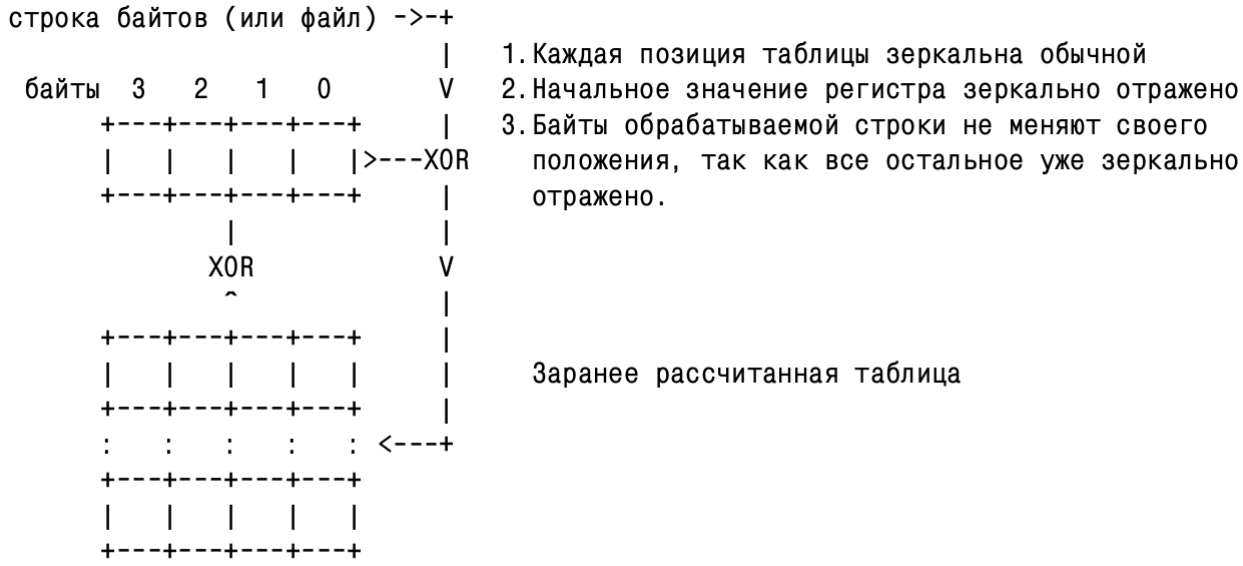
'Зеркальный' табличный алгоритм

Чтобы жизнь не казалась медом, была создана еще и так называемая 'зеркальная' версия этого алгоритма. В 'зеркальном' регистре все биты отражены относительно центра. Например, '0111011001' есть 'отражение' значения '1001101110'.

Эта версия обязана своим возникновением существованию UART (микросхема последовательного ввода/вывода), которая посылает биты, начиная с наименее значимого (бит 0) и заканчивая самым старшим (бит 7), то есть в обратном порядке.

Вместо того, чтобы менять местами биты перед их обработкой, можно зеркально отразить все остальные значения, участвующие в вычислениях, что в результате дает очень компактную реализацию программы. Так, при расчетах биты сдвигаются

вправо, полином зеркально отражается относительно его центра, и, естественно, используется зеркальная таблица предвычисленных значений.



(Схема 3)

В этом случае наш алгоритм станет следующим:

1) Сдвигаем регистр на 1 байт вправо.

2) Выполняем операцию XOR только что выдвинутого вправо из регистра байта с байтом обрабатываемой строки для получения номера позиции в таблице ([0,255]).

3) Табличное значение ксорим с содержимым регистра.

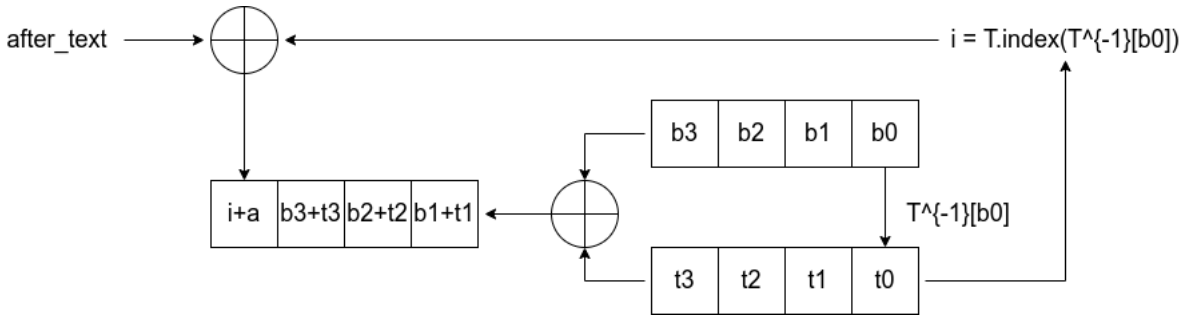
4) Если байты еще остались, то повторяем шаг 1.

Подбор прообраза для CRC

Если мы контролируем некоторое количество подряд идущих байт (равное длине многочлена без старшего разряда в байтах) в документе, то мы можем подобрать для него любое значение CRC.

Пусть у нас будет часть до контролируемого текста и часть после. Нам важно знать, какое CRC имеет часть до контролируемого текста - вычисляем его. Также мы можем, используя байты 'части после' обратить значение CRC, которое нам дано. То есть у нас будет состояние регистра до наших байт и состояние регистра после наших байт, и нам нужно подобрать такие байты, которое превратили бы состояние 'до' в состояние 'после'.

Возьмем прямой алгоритм, который используется в задании. Рассмотрим блок-схему получения состояния 'после' (которое будет содержаться в переменной reg_val):



Если все действия обратить, то мы увидим алгоритм расчета CRC. Нарисуем блок-схему подбора контролируемых байт:

То есть мы сначала от reg_val последовательно получаем все значения из таблицы T^{-1} (в скрипте она называется table_reverse), а затем начинаем в прямом порядке получать байты контролируемого текста (то есть начинаем от crc(pre_text) и в конечном итоге получаем подобранный текст result).

Источники

1) "CRC and how to reverse it". Anarchriz/DREAD

2) https://github.com/221294583/crc32

3) https://raw.githubusercontent.com/beched/forge-crc64/master/forge_crc64.py