

# Optimalizace softwarového zpracování CRC

Jiří Matějka

Vysoké učení technické v Brně, Fakulta informačních technologií  
Božetěchova 1/2 612 66 Brno  
[xmatej52@stud.fit.vutbr.cz](mailto:xmatej52@stud.fit.vutbr.cz)



25. února 2019

- Jeden z nejčastěji používaných algoritmů pro detekci chyb přenosu dat
- Hardwarové i softwarové implementace
- Slabá pro odhalení záměrné změny dat (není schopen odhalit všechny chyby)
- Při náhodné změně vstupní posloupnosti roste pravděpodobnost odhalení chyby spolu s šírkou klíče (dělitele)
- Používá se ve spoustě komunikačních protokolů (Ethernet, asynchronous transfer mode, ...)

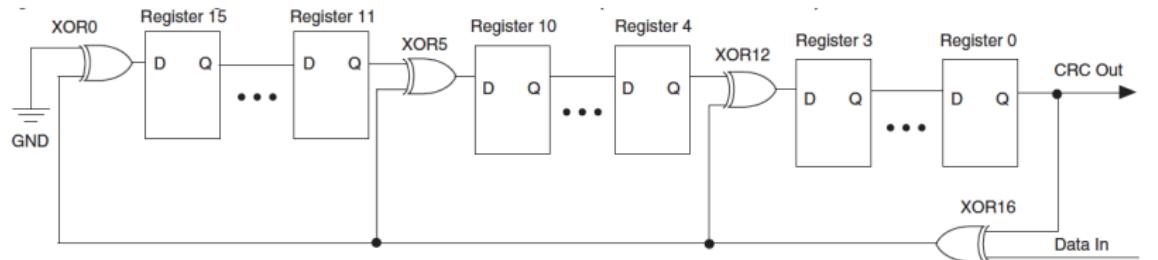
Nechť  $M$  je zpráva, kterou je třeba odeslat.  $C$  je dělitel o délce  $c$  bitů, kde první a poslední bit má hodnotu 1. Potom algoritmus výpočtu odesílané zprávy bude následující:

- 1 Na konec zprávy  $M$  přidej  $c - 1$  nul.
- 2 Výsledek vyděl dělitelem  $C$  ve zbytkové třídě 2
- 3 Zbytkem po dělení označme  $R$ , které je  $c - 1$  dlouhé
- 4 Na konec zprávy  $M$  přidej  $R$  (namísto přidaných nul) a výsledek odešli příjemci

Nechť  $M'$  je zpráva, kterou obdrží příjemce.

Pokud  $Y \bmod C = 0$ , potom příjemce předpokládá, že přijatá zpráva neobsahuje chyby (ne vždy to ale musí být pravda).

- Založena na posuvných registrech a XOR operacích
- Polynom je reprezentován sekvencí binárních hodnot vregistrech



Obrázek: 16 bitová CRC s využitím LFSR (Posuvný registr s lineární zpětnou vazbou)

Obrázek: Zdroj: [https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an049\\_01.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an049_01.pdf)

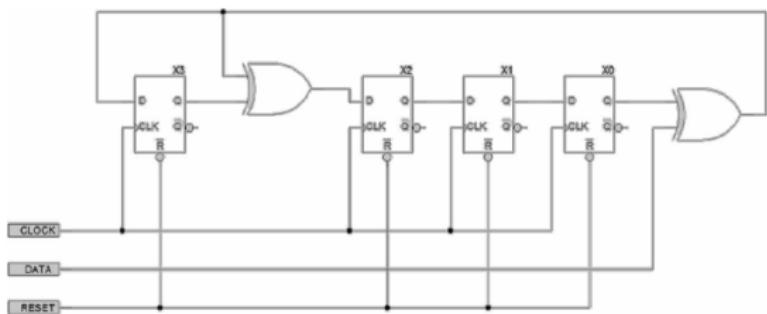
Algoritmus založený na principu posuvného registru:

- Jednoduché řešení
- Opakuje cyklus pro každý bit dat
- Pomalé

Algoritmus s předvypočítanou tabulkou (Table-driven CRC Calculation):

- O něco složitější řešení
- Před prvním výpočtem je třeba inicializovat tabulku
- Opakuje cyklus pro každý byte dat
- Mnohem rychlejší, ale stále pomalejší než HW řešení

Pro jednoduchost budeme pracovat s crc4 a polynomem  $C = x^4 + x + 1$ .



Zdroj: <https://ieeexplore.ieee.org/document/6987839>

Algoritmus implementující následující CRC4 s využitím tabulky bude vypadat následovně:

```
// https://ieeexplore.ieee.org/document/6987839
int CRC4_procByte( int acc, int data, int table[256] )
{
    return acc>>8 ^ table[ (data^acc) & 0xFF ];
}
```

Rozdělme si zprávu na jednotlivé bity. Každý bit označme jako  $d_n$  kde  $n$  je pořadí bitu. Hodnotu v jednotlivých registrech označme jako  $x_{n,t}$ , kde  $n$  je pořadí registru a  $t$  je pořadí prováděného cyklu. Výsledek každého cyklu může být potom reprezentován jako:

$$x_{0,t+1} = x_{1,t}$$

$$x_{1,t+1} = x_{2,t}$$

$$x_{2,t+1} = x_{3,t} \oplus x_{0,t} \oplus d_t$$

$$x_{3,t+1} = x_{0,t} \oplus d_t$$

Clock 0	$x_{0,1} = x_{1,0}$ $x_{1,1} = x_{2,0}$ $x_{2,1} = x_{3,0} \oplus x_{0,0} \oplus d_0$ $x_{3,1} = x_{0,0} \oplus d_0$
Clock 1	$x_{0,2} = x_{2,0}$ $x_{1,2} = x_{3,0} \oplus x_{0,0} \oplus d_0$ $x_{2,2} = x_{0,0} \oplus d_0 \oplus x_{1,0} \oplus d_1$ $x_{3,2} = x_{1,0} \oplus d_1$
Clock 2	$x_{0,3} = x_{3,0} \oplus x_{0,0} \oplus d_0$ $x_{1,3} = x_{0,0} \oplus d_0 \oplus x_{1,0} \oplus d_1$ $x_{2,3} = x_{1,0} \oplus d_1 \oplus x_{2,0} \oplus d_2$ $x_{3,3} = x_{2,0} \oplus d_2$
Clock 3	$x_{0,4} = x_{0,0} \oplus d_0 \oplus x_{1,0} \oplus d_1$ $x_{1,4} = x_{1,0} \oplus d_1 \oplus x_{2,0} \oplus d_2$ $x_{2,4} = x_{3,0} \oplus d_3 \oplus x_{2,0} \oplus d_2 \oplus x_{0,0} \oplus d_0$ $x_{3,4} = x_{3,0} \oplus d_3 \oplus x_{0,0} \oplus d_0$

Zdroj: <https://ieeexplore.ieee.org/document/6987839>

Poslední řádek tabulky, Clock 3, reprezentuje výsledek. Nechť  
 $y_n = x_{n,0} \oplus d_n$ , potom nám po dosazení do Clock 3 vyjde:

$$x_{0,4} = y_0 \oplus y_1$$

$$x_{1,4} = y_1 \oplus y_2$$

$$x_{2,4} = y_3 \oplus y_2 \oplus y_0$$

$$x_{3,4} = y_3 \oplus y_0$$

V softwarové implementaci CRC známe všechny bity dat ( $d_{0..3}$ ) a známe všechny bity akumulátoru ( $d_{0..3,0}$ ). Můžeme tedy položit před samotným výpočtem programu  $x = x \oplus d$ . Vznikne následující implementace algoritmu:

```
// https://ieeexplore.ieee.org/document/6987839
int CRC4_procByte( int acc, int data, int table[256] )
{
    acc = acc ^ data;
    acc = acc >> 8 ^ table[acc & 0xFF];
    acc = acc >> 8 ^ table[acc & 0xFF];
    acc = acc >> 8 ^ table[acc & 0xFF];
    return acc >> 8 ^ table[acc & 0xFF];
}
```

Čeho bylo touto modifikací dosaženo?

- Optimalizace algoritmu využívající tabulku
- Na 32 bitovém procesoru zpracovávo najednou 32 bitů dat
- Maximální délka dělícího polynomu je 32 bitů
- Ušetření 3 instrukcí za každých 32 bitů dat

Rovnici přepíšeme do tabulky:

$$\begin{array}{cccc} y_3 & y_3 & y_1 & y_0 \\ y_0 & y_2 & y_2 & y_1 \\ & y_0 \end{array}$$

Ve sloupcích popřeházíme některé řádky:

$$\begin{array}{cccc} y_3 & y_2 & y_1 & y_0 \\ y_0 & & & \\ y_3 & y_2 & y_1 & \\ y_0 & & & \end{array}$$

Nechť  $z_n = y_n \oplus y_{n-3} = y_n \oplus (y_n \ll 3)$ :

$$\begin{array}{cccc} z_3 & z_2 & z_1 & z_0 \\ z_3 & z_2 & z_1 & \end{array}$$

Výsledkem potom bude  $z = z \oplus (z \gg 1)$

Výše uvedený algoritmus je tak jednoduchý, že ho lze přepsat na pár řádků v assembleru.

```
// https://ieeexplore.ieee.org/document/6987839
eor r0, r1          // y = x ^ d
eor r0, r0, lsl #3  // z = y ^ ( y << 3 )
and r0, #0xF         // z = z & 0xF
eor r0, r0, lsr #1  // crc = z ^ ( z >> 1 )
```

- Algoritmus pro CRC4 byl výrazně zoptimalizován
- Vzorce pro delší šířku klíčů jsou generovány programy v Lispu
  - Model úspěšně aplikován na Ethernet CRC-32
  - V některých dalších případech chybí buď výpočetní výkon nebo metoda není efektivnější než algoritmus s využitím tabulky

Uvedené hodnoty byly naměřeny na stejném stroji a reprezentují počet nanosekund potřebných na zpracování jednoho bytu. Algoritmy byly implementovány v C++ a spuštěny na Samsung Chromebook s 1.7 GHz ARM Cortex-A15 CPU s OS Chrubuntu 12.04 Linux. Každá hodnota byla měřena 100× a byl vybrán vždy ten nejkratší čas. Pro výpočet CRC byla použita náhodně generovaná zpráva o délce 65536 bytů.

	Serial	Table1	Table2	Parallel
CRC16	22.35	5.29	4.85	2.06
CRC32	21.55	5.29	4.85	3.16

- J. R. Engdahl and D. Chung, Fast parallel CRC implementation in software (online), Soul: Automation and Systems (ICCAS 2014). 2014, Dostupné z <https://ieeexplore.ieee.org/document/6987839>
- Altera Corporation, Implementing CRCCs in Altera Devices (online), San Francisko: Altera Corporation. 2005, Dostupné z: [https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an049\\_01.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an049_01.pdf)
- Norman Matloff, Cyclic Redundancy Checking (online), University of California: University of Californi – department of Computer Science, dostupné z:  
<http://heather.cs.ucdavis.edu/~matloff/Networks/CRC/Old/ErrChkCorr.NEW.tex>

```
uint16_t crc16_software( char *message ) {
    uint64_t crc = 0xFFFF;
    while ( *message ) {
        crc ^= *message++ << 8;
        for ( int i = 0; i < 8; i++ ) {
            if ( crc & 0x8000 ) {
                crc = ( crc << 1 ) ^ 0x1021;
            }
            else {
                crc <<= 1;
            }
        }
    }
    return crc;
}
```

```
uint16_t crc_table16[256] = {0, };
void init() {
    uint16_t remainder16 = 0;
    for ( int i = 0; i < 256; i++ ) {
        remainder16 = i << 8;
        for ( int i = 0; i < 8; i++ ) {
            if ( remainder16 & 0x8000 )
                remainder16 = ( remainder16 << 1 ) ^ 0x1021;
            else
                remainder16 <= 1;
        }
        crc_table16[ i ] = remainder16;
    }
}
uint16_t crc16_software_table( char *message ) {
    uint16_t crc = 0xFFFF;
    while ( *message )
        crc = (crc << 8) ^ crc_table16[((crc >> 8) ^ *message++)];
    return crc;
}
```