LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Gruppe 120 – Abgabe zu Aufgabe A401
Wintersemester 2018/19

Liudongnan Yang Alexandru Balotescu Mehmet Dereli

Inhaltsverzeichnis

1 Einleitung **2**

2 Problemstellung und Spezifikation **2**
2.1 Problemstellung . . . 2
2.2 Spezifikation . . . 3
2.2.1 Beispiel . . . 3
2.3 Sicherheit . . . 5
2.3.1 Beispiel . . . 5

3 Lösungsfindung **6**
3.1 Implementation . . . 6
3.1.1 CRC4 in C . . . 6
3.1.2 CRC32 in C . . . 6
3.1.3 Optimierter CRC32 . . . 6
3.1.4 CRC32 in Assembler . . . 7

4 Dokumentation der Implementierung **7**
4.1 Entwickler-Dokumentation in C . . . 7
4.1.1 Iterative Version . . . 7
4.1.2 Lookup Table Version . . . 8
4.2 Entwickler-Dokumentation in Assembler . . . 9
4.3 Benutzer-Dokumentation . . . 10

5 Ergebnisse **10**
5.1 Laufzeit . . . 10
5.2 Genauigkeit . . . 11

6 Zusammenfassung **11**

1 Einleitung

In unserem Informatik: Games Engineering Bachelor Studium ist es ein Teil des Studienziels, zu lernen wie die Entwicklung von Software, insbesondere Games, durch hardwarenahe Programmierung umgesetzt wird.
Dazu dient das Praktikum 'Aspekte der Systemnahen Programmierung', welches wir im Laufe des Wintersemesters 2018/2019 besuchen.
Während des Praktikums haben wir grundlegende Kenntnisse zur Programmierung mit Assembler kennengelernt und konnten diese mithilfe der TUM-eigenen Raspberry-Pis anwenden.
Zu Beginn haben wir die korrekte Umrechnung von unterschiedlichen Stellenwertsystemen gelernt und Einblicke in die C-Programmierung erhalten. Mithilfe dieser Kenntnisse ist uns die Abstufung zwischen Hochsprachen und den eigentlichen Operationen auf Hardwareebene verständlich geworden. Weitere Lernerfolge waren das Erlernen der Aufrufkonventionen und der Einteilung der Register.
Nach dem ersten Drittel, welches aus Grundlagen bestand, folgte eine ARM-Coding orientierte Praktikumseinheit. Bei dieser lag der Fokus auf komplexeren ARM-Structuren, wie beispielsweise dem Barrelshifter oder der Floating-Point-Unit.
Das letzte Drittel der Lehreinheiten, bevor uns ein Gruppenprojekt zur Bearbeitung zugewiesen wurde, bestand zum einen aus einer Erklärung der SIMD (Single-Instruction-Multiple-Data) Befehle und zum anderen aus einer Einführung in Tools zur Performance-Analyse.
Die Mathematik geht mit der Informatik einher. Mathematische Methoden helfen uns dabei Algorithmen zu implementieren und somit konkrete Probleme effizient zu lösen.
Diese Aufgabe erfordert den vielseitigen Einsatz von Fähigkeiten wie der Performanz Analyse, Programmierkenntnisse in C, Assembler sowie Teamwork.
Diese Dokumentation beschreibt das Problem, wie es gelöst wurde und wie die fertige Implementation genutzt werden kann. Zuletzt werden die Ergebnisse, also die Zusammenfassung sowie die Qualität und Performance des Programms präsentiert und eingeschätzt.

2 Problemstellung und Spezifikation

Ein wichtiger Teilbereich der Informatik ist die Informationstheorie. Dabei geht es allgemein um die korrekte Kodierug und Dekodierung von Nachrichten.
Im folgenden wird ein bestimmter Algorithmus aus diesem Teilbereich, und zwar der CRC32, näher erläutert.

2.1 Problemstellung

Algorithmen und das Internet sind nicht ohne Fehler. Diese werden durch verschiedenste Gründe ausgelöst und müssen daher erkannt und behoben werden. Der CRC32 ist ein fehlererkennendes Verfahren. Dieses Verfahren wird beispielsweise von der zwei-

ten Schicht (Sicherungsschicht) des ISO/OSI-Modells (International Organization for Standardization/Open Systems Interconnection Modell) verwendet um zu überprüfen, ob die übertragene Nachricht korrekt ist oder ob sie gegebenenfalls neu vom Sender beantragt werden muss, da es durchaus zu einer Fehlübertragung durch die erste Schicht (Physikalische Schicht) gekommen sein kann.

2.2 Spezifikation

Die Zyklische Redundanzprüfung erstellt, mithilfe der Polynomdivision, sogenannte Prüfsummen (engl. Checksums) von Dateien. Die zu übertragende Nachricht N mit der Länge l wird als Polynom aufgefasst:

$$N = x_1x_2x_3...x_n = \sum_{i=0}^{l-1} a_i \cdot x^i \text{ mit } a_i \in \{0,1\} = N(x) \tag{1}$$

Als Generatorpolynom $G(x)$ wird bei CRC32 das folgende Polynom des Grades 32 verwendet:

$$G(X) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+x^0 \tag{2}$$

Zu Beginn wird $N(x)$ mit dem Grad des Generatorpolynoms multipliziert, was dem Anhängen von 32 Nullen entspricht.
Um nun eine korrekte Prüfsumme für die Nachricht $N(x)$ zu erhalten, wird $N'(x)$ durch $G(x)$, mithilfe der Polynomdivision, dividiert. Der Rest der Polynomdivision wird an die ursprünglich Nachricht $N(x)$ angehängt.

Um nun die Korrekheit von $N''(x)$ zu überprüfen, wird die Nachricht mit dem Generatorpolynom polynomdividiert. Wenn der Rest der Division Null ist, ist mit hoher Wahrscheinlichkeit kein Übertragungsfehler aufgetreten. Andernfalls ist auf jeden Fall ein Übertragungsfehler aufgetreten.

2.2.1 Beispiel

Der Ablauf des CRC Verfahrens ist, unabhängig vom höchsten Exponenten des Generatorpolynoms, immer dasselbe. Um den Ablauf des Verfahrens zu veranschaulichen, stellen wir das Verfahren mit den folgenden Bitfolgen dar:

$$G(X) = x^5 + x^4 + x^2 + x^0 \tag{3}$$

$$N(X) = 10010011 \tag{4}$$

Zu Beginn werden Nullen in Höhe des höchsten Exponenten des Generatorpolynoms an $N(x)$ gehängt:

$$N'(x) = 1001001100000 \tag{5}$$

Um die Prüfsumme von $N(x)$ zu erhalten, wird $N'(x)$ durch das Generatorpolynom $G(x)$ polynom dividiert:

```
  1 0 0 1 0 0 1 1 0 0 0 0 0 : 1 1 0 1 0 1 = 1 1 1 1 0 0 1 1
⊕1 1 0 1 0 1
    1 0 0 0 1 1
  ⊕1 1 0 1 0 1
      1 0 1 1 0 1
    ⊕1 1 0 1 0 1
        1 1 0 0 0 0
      ⊕1 1 0 1 0 1
        0 0 0 1 0 1 0 0 0
            ⊕1 1 0 1 0 1
              0 1 1 1 0 1 0
              ⊕1 1 0 1 0 1
                0 0 1 1 1 1
```

Damit erhalten wir den folgenden $Rest = 01111$. Anstelle der Nullen, wird nun der berechnete Rest an die ursprüngliche Nachricht $N(x)$ gehängt. Damit erhalten wir die folgende zu übertragende Nachricht:

$$N''(x) = 1001001101111 \tag{6}$$

Um nun die zu übertragende Nachricht $N''(x)$ auf ihre Korrektheit hin zu überprüfen, wird eine weitere Polynomdivision von $N''(x)$ und $G(x)$ durchgeführt. Dieser Schritt geschieht üblicherweise beim Empfänger der Nachricht, da dieser wissen will, ob bei der Übertragung der Bitfolge über die Physische Schicht ein Fehler unterlaufen ist oder ob es sich dabei um die korrekte Bitfolge handelt. Sollte ein Fehler in der Nachricht vorliegen, wird eine neue Übertragung der Bitfolge vom Empfänger beantragt.

```
  1 0 0 1 0 0 1 1 0 1 1 1 1 : 1 1 0 1 0 1 = 1 1 1 1 0 0 1 1
⊕1 1 0 1 0 1
    1 0 0 0 1 1
  ⊕1 1 0 1 0 1
      1 0 1 1 0 1
    ⊕1 1 0 1 0 1
        1 1 0 0 0 0
      ⊕1 1 0 1 0 1
        0 0 0 1 0 1 1 1 1
            ⊕1 1 0 1 0 1
              0 1 1 0 1 0 1
              ⊕1 1 0 1 0 1
                0 0 0 0 0 0
```

Da der Rest der Division Null ist, ist mit hoher Wahrscheinlichkeit kein Fehler aufgetreten.

2.3 Sicherheit

Der Grund weshalb bei der Nachrichtenübertragung im Internet mehr als nur das CRC32 Verfahren verwendet wird, ist der, dass dieses Verfahren nicht alle Übertragungsfehler erkennt.
Wenn n die Länge der Prüfsumme ist, also $n = grad(G(x))$, werden die folgenden Fehler nicht zuverlässig oder gar nicht erkannt:

- Fehler, die länger sind als n
- Fehler, die aus mehreren Bursts bestehen
- Alle Fehler, die ein Vielfaches des Generatorpolynoms sind.

2.3.1 Beispiel

Alle Fehler die ein Vielfaches des Generatorpoylnoms sind, können nicht zuverlässig oder garnicht erkannt werden. Ein Beispiel dafür ist das folgende.
Für die folgende Nachricht $N(x)$

$$N(x) = 00101100 \tag{7}$$

und dem folgenden Generatorpolynom $G(x)$

$$G(x) = x^3 + x^1 + x^0 \tag{8}$$

ist das Ergebnis der Polynomdivision gleich Null.
Wenn nun jedoch ein Fehler $F(x)$, welches ein Vielfaches des Generatorpolynoms ist, auftritt,

$$\begin{aligned} N(x) &= 00101100 \\ \oplus F(x) &= 10110000 \\ \hline M(x) &= 10011100 \end{aligned} \tag{9}$$

ist der Rest der Polynomdivision von $M(x)$ durch $G(x)$ ebenfalls gleich Null.

```
  1 0 0 1 1 1 0 0 0 0 0 : 1 0 1 1 = 1 0 1 0 0 0 0 0
⊕ 1 0 1 1
  0 0 1 0 1 1
    ⊕ 1 0 1 1
      0 0 0 0 0 0 0 0
```

Das bedeutet, dass mehrere Nachrichten auf den selben Rest abbilden und es somit trotz Fehler bei der Übertragung, zu einem korrekten Ergebnis kommen kann.

3 Lösungsfindung

Um einen Vergleich zwischen Assembler und höheren Programmiersprachen ziehen zu können, war es Teil der Aufgabe den CRC32 sowohl in C als auch in Assembler zu programmieren.
Im Laufe der Lösungsfindung haben wir drei aufeinander aufbauende Stufen durchlaufen.

3.1 Implementation

3.1.1 CRC4 in C

Um den CRC32 besser zu verstehen, haben wir als ersten Schritt den CRC4 in C implementiert. Die Codestruktur und der Ablauf der beiden Verfahren sind sich sehr ähnlich, jedoch schien uns eine Implementierung des CRC4 zu Beginn simpler. Nach der Implementierung, haben wir diesen Code als Basis für unsere folgende Implementation des CRC32 verwendet.
Zum einen wurde durch die eigentliche Implementation des Ablaufs das Verfahrens weiter vertieft, zum anderen konnten wir Fehler in unseren Gedankengängen finden und haben somit einen ersten Meilenstein abgeschlossen.

3.1.2 CRC32 in C

Um einen sauberen CRC32 zu implementieren, haben wir den Code auf dem bereits vorhandenen CRC4 Code aufgebaut. Wir haben die Struktur verbessert und den Ablauf optimiert, indem wir einige Teile in Methoden ausgelagert und andere Teile umgeschrieben haben. Der nun implementierte und verbesserte Code hat das Verfahren wie gewünscht ausgeführt.

3.1.3 Optimierter CRC32

Um die Implementation weiter zu verbessern, sind wir auf eine neue Methode zur Berechnung der Polynomdivision gestoßen.
Da das Generatorpolynom im CRC32 immer das selbe ist, kann man die 265 Werte der Bitfolgen vor der Ausführung des CRC32 in ein Lookup Table schreiben. Wenn nun eine Bitfolge durch das Generatorpolynom geteilt wird, muss nur der entsprechende Wert aus dem Lookup Table genommen werden und es wird keine bitweise XOR-Operation benötigt.
Die Nachricht wird dabei in Stücken von 1 Byte bearbeitet. Das erste Byte wird im Lookup Table nachgeschaut. Der 4-Byte Wert aus dem Table wird dann mit den nächsten 4 Byte bitweise mit XOR-Operationen verrechnet. Daraufhin wird das erste Byte des Ergebnisses im Table nachgeschaut. Der Wert wird mit den restlichen 3-Byte des Ergebnisses plus dem nächten Byte der Nachricht mit einer bitweisen XOR-Operationen verrechnet.

Dieser Prozess wird so lange wiederholt, bis die Nachricht zu Ende ist und ein Rest übrig bleibt.

3.1.4 CRC32 in Assembler

Analog zu dem optmierten Alogirthmus in C, haben wir den CRC32 in Assembler geschrieben. Jedoch war es uns nicht möglich SIMD-Befehle mit einzubauen, da die einzelnen Schritte aufeinander aufbauen. Die nächsten 4 Byte folgen werden immer durch das aktuelle Byte verändert und können somit nicht parallel bearbeitet werden. Bei der Programausführung müssen Aufrufkonventionen eingehalten werden. Diese beschreiben wie eine Funktion aufgerufen wird und welche Regeln in Caller-Save bzw. Callee-Save Registern eingehalten werden müssen.
Wir konnten das Verletzen der Aufrufkonventionen, durch das Laden des Werts aus dem Register [x19] auf den Stack, umgehen.

4 Dokumentation der Implementierung

4.1 Entwickler-Dokumentation in C

Der vorhandene Code lässt sich in zwei unterschiedliche Lösungswege einteilen.
Zum einen den iterativen Weg und zum anderen den optimierten Weg, in welchem Bytes mithilfe eines Lookup Tables bearbeitet werden.

4.1.1 Iterative Version

Bei der iterativen Version wird die Funktion $unsigned\ int\ CRC32\_C\_helper2(char * data)$ aufgerufen. Diese Funktion ist das Rahmenprogramm für die Berechnung der Prüfsumme und gibt diese am Ende zurück.
Zuerst ermittelt die Funktion $int\ char\_counter(char *\ data)$ die Länge der Nachricht, indem über die Bytes iteriert wird, bis die Iteration die Zeichenkombination '\0' erreicht. Danach padded die Funktion $char*padding(int\ data\_length, char*\ data)$ die Nachricht:

```
1 char* padding (int data_length , char* data)
2 {
3     char* padded_data = (char*) malloc(data_length+4);
4     memset(padded_data,0,data_length);
5     memcpy(padded_data,data,data_length);
6     return padded_data;
7 }
```

In den ersten beiden Zeilen wird ein Bereich im Speicher, der die Länge der Nachricht plus die 4 Byte der anzuhängenden Nullen, mit Nullen gefüllt. Danach wird die Nachricht $data$ in diesen Speicher kopiert.

Nachdem das Ergebnis returned wird, wird in der Hauptschleife der Funktion die bitweise Polynomdivision durchgeführt.

```
unsigned int CRC32_C_helper2(char* data)
{
    char* padded_data = padding2(data);
    //initialize the Register & Byte-position index
    unsigned int reg = 0 ;
    unsigned int pos = 0 ;

    //main loop
    //stop the loop when there's no char in the stack
    while (pos != strlen(padded_data)+4){
        for(int cur_bit = 7; cur_bit >= 0 ; cur_bit--){

                int high = (reg >> 31) & 0x01;
                //lsl
                reg <<= 1;
                //load data from buffer
                //push up the next bit from the buffer into reg
                reg |= ((padded_data[pos]>>cur_bit)&0x00000001);
                if(high == 1){
                    reg = reg ^ POLY;
                }
        }
        pos++;
    }
    return reg;
}
```

Es wird in der Main loop solange iteriert, bis der Integer pos, welches das momentane Byte repräsentiert, so groß wie die Länge der gepaddeten Nachricht in Byte ist.
Der Integer reg repräsentiert die momentan bearbeitete 32Bit Folge.
In der for-Schleife werden die einzelnen Bits der Nachricht in die Variable reg geladen und sollte das höchste Bit eine Eins sein, wird eine Polynomdivision von reg mit dem Generatorpolynom durchgeführt.
Zum Schluss wird der Rest der Polynomdivision, welcher sich in der Variable reg befindet, zurückgegeben und von der Main-Funktion über einen Print ausgegeben.

4.1.2 Lookup Table Version

Bei der Lookup Table Version wird die Funktion $unsigned\ int\ CRC32\_C\_table(char * data)$ aufgerufen. Diese Funktion ist das Rahmenprogramm für die Berechnung der Prüfsumme und gibt diese am Ende zurück.
Bevor diese Funktion jedoch aufgerufen werden kann, muss in der Main-Funktion $generateCRC32\_Table()$ aufgerufen werden. Diese Funktion berechnet die 256 verschiedenen Ergebnisse, die eine Polynomdivision zwischen einer 8 Bit (mit 4 Byte padding) Folge und dem Generatorpolynom haben kann. Die Ergebnisse der Berechnung werden

in das folgende Array eingetragen $unsigned\ int\ Table\_CRC32[256]$.

```
1  unsigned int CRC32_C_table(char* data)
2  {
3     unsigned int reg = 0;
4     unsigned int pos = 0;
5     while (pos != strlen(data)+4){
6        unsigned int tablevalue = Table_CRC32[(reg>>24)&0xff];
7        reg = (reg<<8)|data[pos];
8        reg = reg ^ tablevalue ;
9        pos++;
10    }
11    return reg;
12 }
```

Die Variablen reg und pos sind für den momentan berechneten Wert und für die Position in $data$ zuständig.
Die Berechnung des Restes erfolgt solange, wie die Variable pos nicht das letzte Bit der Nachricht $data$ erreicht hat.
In Zeile 6 wird der polynomdividerte Wert, der Bitfolge aus reg, aus dem Lookup Table herausgenommen und der Variable $tablevalue$ zugewiesen. Die nächsten 4 Byte der Nachricht werden dann mit diesem Wert polynomdividiert und anschließend wieder in reg geschrieben. In der darauf folgenden Zeile wird pos inkrementiert, wodurch sich entweder die Schleife wiederholt, oder das Ende der Nachricht erreicht ist.

4.2 Entwickler-Dokumentation in Assembler

Die Assembler Version wird von der C-Mainfunktion aus aufgrufen. Dabei übergibt der Aufruf lediglich die gepaddete Datei an den Assemblercode weiter. Die Tabelle wird von der Assemblerfunktion aus aufgrufen und in x19 gespeichert. Die Nachricht wird dabei in zwei Teile unterteilt. Einmal die Nachricht und einmal den gepaddeten Anteil. Der einzige Unterschied ist der, dass es beim letzteren einen Zähler von 4 Byte gibt, durch welchen das genaue Ende bestimmt werden kann. Der erste Teil endet nach der Zeichenkombination '\0' erreicht wird.

```
1  .mainloop:
2      ldrb w2, [x0],1  // Post-increment: adds 1 to x0 after loading
3      cbz w2, .Lpad  // Branch to return if char is '\0'
4
5
6      mov x5, x4 ,lsr 24
7      and x5, x5 ,0xff
8      //load table value to w6
9      //convert byte to int ( need *4 )
10     ldr w6,[x1,x5,lsl 2]
11     //adding next byte data from the pointer
```

```
12      orr x4,x2,x4,lsl 8
13      eor x4,x4,x6
14
15      b .mainloop
```

Zu Beginn wird überprüft, ob das Ende des ersten Teils der Nachricht erreicht wurde. Wenn dies nicht der Fall ist, werden die 4 Byte nach dem aktuellen Byte geladen in x5 geladen. Danach wird der Tabellenwert des aktuellen Bytes in w6 geladen und dann mit dem aktuell berechneten Wert mit einer Bitweisen XOR-Operation verrechnet.
Das Ergebnis wird anschließend über das Register x0 zurückgegeben.

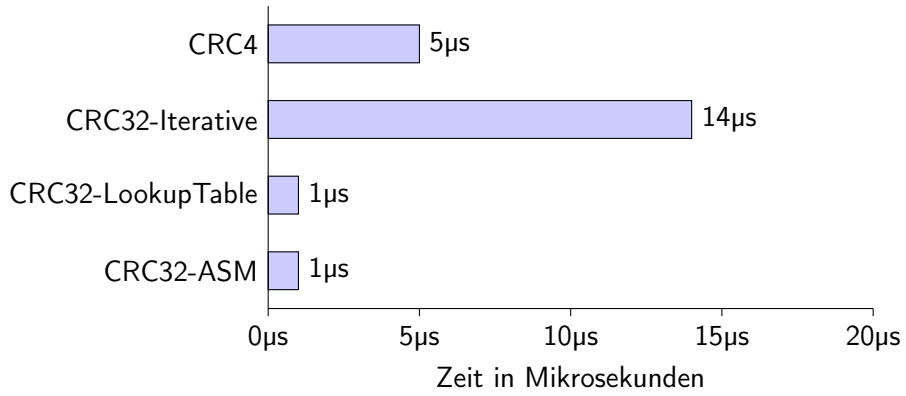
4.3 Benutzer-Dokumentation

Um das Programm als Benutzer zu verwenden, gibt es einige Dinge zu beachten.
Nach dem Aufruf des Programms mit $./CRC32 < inputstring >< runtimes >$, muss zuerst die zu übertragende Nachricht als Parameter und dann die Anzahl an Wiederholungen angegbeben werden. Diese Nachricht wird dann mit dem CRC4, CRC32(iterative Version), CRC32(Lookup Table Version) und dem CRC32 in Assambler verarbeitet. Die Ergebnisse werden zusammen mit der Laufzeit der Programme und den jeweiligen Ergebnissen zurückgeliefert.

5 Ergebnisse

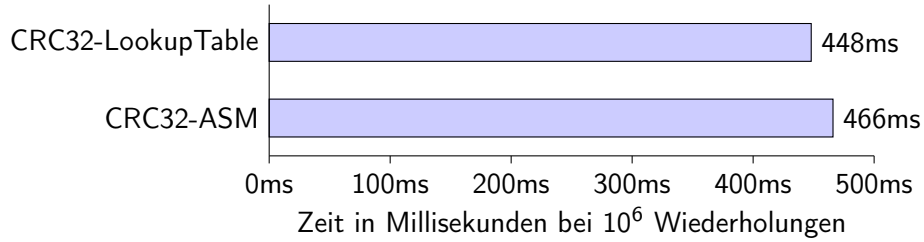
5.1 Laufzeit

Um die Effizienz der Implementationen zu vergleichen, wird die Prüfsumme einer Nachricht von den verschiedenen Implementationen berechnet.
Wir haben für den Vergleich die Nachricht „SayHellotoMyLittleFriend"gewählt. Ein char in diesem String besitzt 8 Bit und bei 24 char sind das 195 Bit die überprüft werden. Die Ergebnisse sind in dem folgenden Diagramm einsehbar:



Aus dem Diagramm lässt sich erkennen, dass die Iterative 32Bit Version definitiv am langsamsten ist. Dies ist auf die erhöhte Anzahl an Rechenoperationen zurückzuführen.

Bei der Lookup Table Version werden nur halb so viele XOR-Operationen ausgeführt, da die Ergebnisse bereits in der Tabelle vorhanden sind.
Die Iterative 4Bit Version ist 3 Mal so schnell wie die 32 Bit Version, da die Prüfsumme bei der 4 Bit Version kleiner ist.



Bei 10^6 Durchläufen entsteht ein erkennbarer Unterschied in der C und der Assemblerprogrammierung. Die Assembler Programmierung ist 12 ms langsamer als die C Programmierung. Da beim Austausch von Daten im Internet eine weitaus höhere Anzahl an Durchläufen geschieht, lässt sich daraus schließen, dass die C Programmierung deutlich besser ist.

5.2 Genauigkeit

Die Frage „Wann benutzt man CRC32 und wann CRC4" lässt sich leicht beantworten, da die Antwort bereits in den Namen vorhanden ist. Die mögliche Anzahl an Resten sollte größer/gleich die Länge der Nachricht sein, da sonst mehrere Reste auf die selbe Nachricht abbilden. Bei der 4 Bit Version gibt es 2^4 verschiedene Reste und bei der 32 Bit Version gibt es 2^{32} verschiedene Reste, was jeweils genau der Anzahl an möglichen Bitfolgen für Nachrichten der Länge 4 bzw. 32 Bit entspricht.
Die Version muss demnach der Nachricht angepasst werden, damit es einen möglichst effizienten Tradeoff zwischen Schnelligkeit und Genauigkeit gibt.

6 Zusammenfassung

Da höhere Programmiersprachen bereits auf einer effizienten niedrigeren Programmiersprache aufbauen, ist das Ergebnis unserer Arbeit die, dass es in unserem Fall keinen Sinn macht den CRC in Assembler zu programmieren, da der C-Code bereits die effizienteste Version ist.
Über den CRC-Algorithmus lässt sich sagen, dass die LookupTable-Version definitiv die beste Implementierung ist und dass die Bitversion basierend auf der Länge der Nachricht ausgewählt werden soll, damit man Geschwindigkeit und Genauigkeit am effizientesten ausbalanciert. Wir waren jedoch nicht in der Lage bei der Assemblerprogrammierung SIMD-Befehle mit einzubauen (siehe 3.1.4 CRC32 in Assembler). Dies wäre ein Anhaltspunkt für weitere Verbesserungen der Implementation.

Literatur

[1] ARM Procedure Call Standard for the ARM Architecture,
zuletzt aufgerufen am 23.01.2018
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802b/CMEQ_advsimd_zero_vector.html`

[2] E. Stein.
Taschenbuch Rechnernetze und Internet, chapter Fehlererkennung durch CRC, pages 86–87.
Fachbuchverlag Leipzig, 2. edition, 2004.
Auszug s. Moodle

[3] CRC with table value.
`https://blog.csdn.net/xx326664162/article/details/51718857`

[4] ASCII converter
`https://www.rapidtables.com/convert/number/ascii-hex-bin-dec-converter.html`

[5] Online CRC calculation
`http://www.ghsi.de/pages/subpages/Online`