Neues Konzept zur Steigerung der Zuverlässigkeit einer ARM-basierten Prozessorarchitektur unter Verwendung eines CGRAs

Konstantin Lübeck, David Morgenstern, Thomas Schweizer, Dustin Peterson,
Wolfgang Rosenstiel, Oliver Bringmann

Eberhard Karls Universität Tübingen
Wilhelm-Schickard-Institut, Eingebettete Systeme
Sand 13, 72076 Tübingen
{luebeck,morgenst,tschweiz,peterson,rosenstiel,bringman}@informatik.uni-tuebingen.de

Abstract. Zuverlässigkeit von eingebetteten Systemen spielt im Kontext der fortschreitenden Miniaturisierung eine immer wichtigere Rolle. In diesem Beitrag stellen wir ein neues Konzept zur Steigerung der Zuverlässigkeit einer ARM-basierten Prozessorarchitektur unter Verwendung einer bereits kostengünstig gehärteten grobgranularen rekonfigurierbaren Architektur (CGRA) vor. Dieser kostengünstig gehärtete CGRA wird dazu in die Pipeline eines ARM-Prozessors integriert. Dabei wird die Struktur der ARM-Prozessorpipeline nicht verändert, sondern lediglich der Befehlssatz um drei Instruktionen erweitert. Dafür haben wir eine Toolchain erstellt, welche diese Integration automatisiert.

1. Einleitung

Die Zuverlässigkeit jederzeit gewährleisten zu können ist eine ganz besondere Herausforderung in sicherheitsrelevanten eingebetteten Systemen der Medizintechnik, der Luft- und Raumfahrt und auch des Automobilbereichs. Dabei muss insbesondere der fortschreitenden Miniaturisierung in der Halbleitertechnologie und der damit verbundenen erhöhten Anfälligkeit gegenüber Defekten beim Herstellungsprozess, dem höheren Verschleiß im Betrieb und auch der erhöhten Strahlungsanfälligkeit durch die redundante Ausführung von Komponenten Rechnung getragen werden.

Allgemein sind unter Redundanz alle bei Fehlerfreiheit entbehrlichen Mittel zu verstehen, die im Fehlerfall die ausgefallene Funktionalität ersetzen können. Zu unterscheiden ist die statische und die dynamische Redundanz. Der klassische Vertreter für die statische Redundanz ist das Verfahren der N-fach modularen Rendundanz, wie beispielsweise TMR (Triple Modular Redundancy). Bei diesem Verfahren wird dieselbe Funktion von mehreren Modulen (mindestens zwei) gleichzeitig ausgeführt. Ein Voter entscheidet hier nach dem Mehrheitsprinzip welches Ergebnis korrekt ist und kann somit auch bestimmen welche Komponente beschädigt ist (siehe Abbildung 1a).

Im Gegensatz zur statischen Redundanz hat die dynamische Redundanz den Vorteil, dass nur ein einziges Modul die Funktionalität erbringen muss. Der Fehlerfall wird durch die Komponente selbst oder durch eine externes Modul festgestellt. Anschließend übernimmt dann ein redundantes

R. Wimmer (Hrsg.): MBMV 2016
ISBN: 978-3-00-052380-9

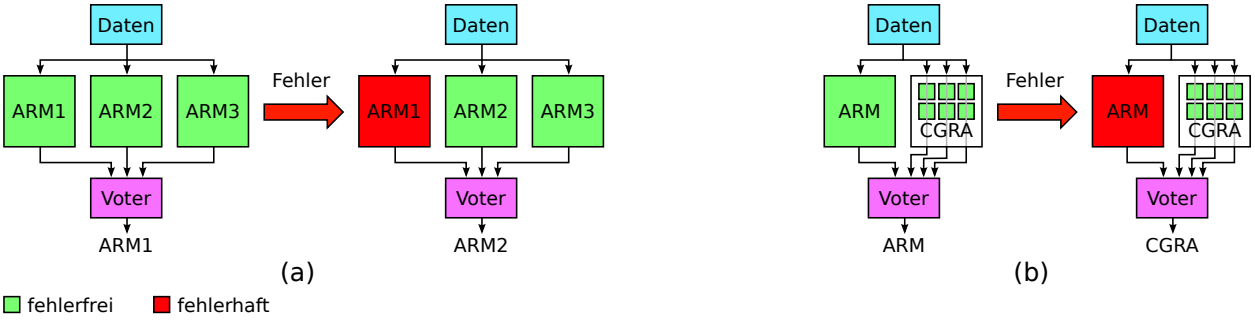

Abbildung 1: Statische Redundanz

Modul die erforderliche Funktionalität und ermöglicht dadurch einen möglichst langen fehlerfreien Betrieb des Systems (graceful degradation, siehe Abbildung 2a).

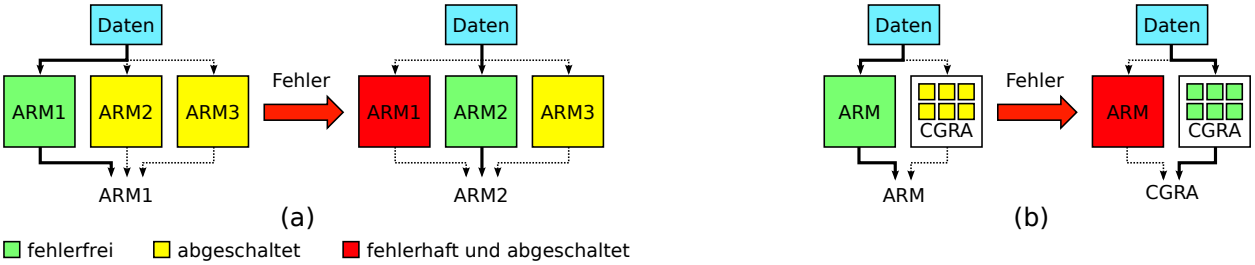

Abbildung 2: Dynamische Redundanz

In eingebetteten Systemen kann das Verfahren der Redundanz beim Speicher, der Kontrolllogik und dem Prozessor angewendet werden. So bietet der Cortex-R5 Prozessor von ARM die Möglichkeit, das System einer Single-CPU, mit zwei unabhängigen CPUs oder mit einer Master-CPU und einer redundanten CPU zu betreiben.

Wir verfolgen einen neuen Ansatz zur Realisierung der Redundanz in eingebetteten Prozessoren. Unser Ansatz basiert auf den besonderen Eigenschaften von rekonfigurierbaren Architekturen. Rekonfigurierbare Architekturen bieten die Möglichkeit zur Veränderung der Funktionalität über die Zeit. Bei statisch rekonfigurierbaren Architekturen geschieht dies in der Regel nur einmalig für jede Anwendung (*single context*). Dynamisch rekonfigurierbare Architekturen erlauben dagegen durch extrem schnelles Rekonfigurieren (in weniger als einem Taktschritt) zur Laufzeit, die Funktionsblöcke eines Bausteins, für unterschiedliche Operationen der gleichen Anwendung, wiederzuverwenden. Wir bezeichnen solche Architekturen als *prozessorartig rekonfigurierbar* oder auch als *multi context* Architekturen. Prozessorartig rekonfigurierbare Architekturen sind grobgranularer und benötigen daher weniger Konfigurationsdaten als herkömmliche FPGAs [4]. Sie ermöglichen somit das Speichern mehrerer Kontexte in einem kleinen Speichermodul innerhalb der rekonfigurierbaren Architektur, wobei eine Zeile in diesem Speicher einem Kontext entspricht. Der Wechsel eines Kontextes kann somit innerhalb eines Taktschritts durchgeführt werden, da lediglich eine andere Speicherzeile ausgewählt werden muss. Dadurch kann die Rekonfiguration mit der Ausführung Schritthalten wodurch sich ein zusätzlicher Freiheitsgrad ergibt. Dieser Freiheitsgrad zeichnet dynamisch rekonfigurierbare Architekturen vor allen anderen Architekturen für die Realisierung der Redundanz aus. So wurde bereits in [11] gezeigt, dass durch die Möglichkeit der dynamischen Rekonfiguration kostengünstig TMR in grobgranularen rekonfigurierbaren

Architekturen (engl. *coarse grained reconfigurable architectures*, CGRAs) realisiert werden kann. Weiterhin kann durch diesen Freiheitsgrad ein dynamisches Remapping [3] zur Laufzeit durchgeführt werden. Dadurch kann eine defekte Komponente eines CGRAs umgangen und ein verlängerter fehlerfreier Betrieb des CGRAs ermöglicht werden. Dies bezeichnen wir als gehärtet. Abbildung 3 zeigt beispielhaft dynamisches Remapping innerhalb eines CGRAs in der räumlichen sowie der zeitlichen Domäne.

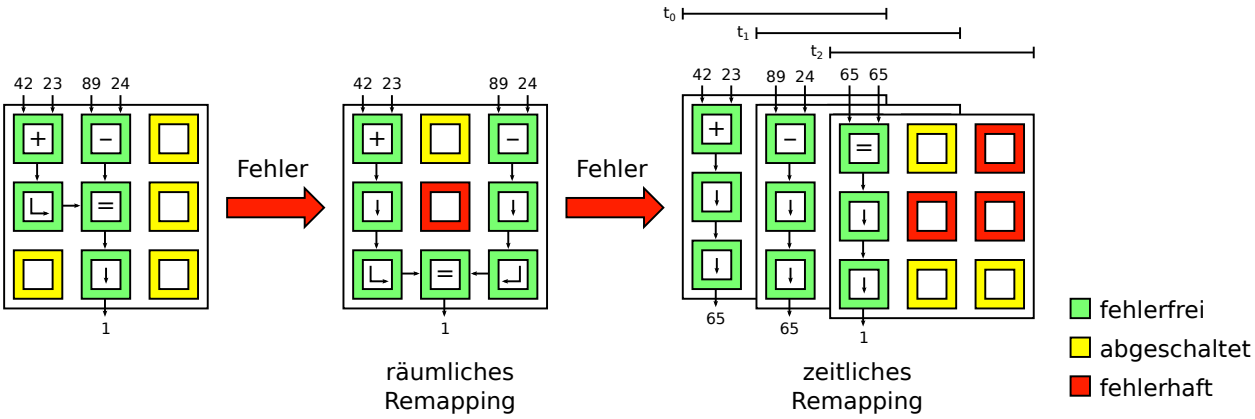

Abbildung 3: Gehärteter CGRA

Unser Ziel ist es, solch einen kostengünstig gehärteten CGRA in die Pipeline eines eingebetteten Prozessors zu integrieren. Dieser Ansatz bietet mehrere Vorteile gegenüber der Erhöhung der Anzahl der Prozessorkerne:

- Die redundante Komponente ist schon gehärtet (siehe Abbildung 3).
- Die Komponente ist flexibel und erlaubt eine einfache Befehlssatzerweiterung, d. h. der Prozessor kann nach der Herstellung um neue Befehle erweitert werden, die im Nicht-Fehlerfall dem Prozessorkern zur Verfügung stehen. Diese Befehlssatzerweiterungen können komplexe arithmetische Operationen abbilden und somit die Performanz des Prozessors steigern.
- Die Komponente ist sowohl für die statische als auch die dynamische Redundanz geeignet (siehe Abbildung 1b und Abbildung 2b).
- Der CGRA ermöglicht es, dynamisch zusätzliche redundante Komponenten zur Verfügung zu stellen.

Diese Vorteile sind nur dann voll nutzbar, wenn es gelingt, den CGRA so zu konfigurieren und zu integrieren, dass kein Geschwindigkeitsverlust bei der Ausführung eines Befehls auf dem CGRA im Vergleich zur Ausführung auf dem Prozessor entsteht und der Flächenbedarf in annehmbaren Größen bleibt. In diesem Beitrag haben wir uns auf die Integration des CGRAs in die Prozessorpipeline fokussiert und präsentieren die nötigen Schritte dieser Integration. Konkret stellen wir ein Konzept zur Integration einer grobgranularen rekonfigurierbaren Komponente in eine ARM-Prozessorpipeline vor.

Dieser Beitrag gliedert sich wie folgt: Abschnitt 2 gibt einen kurzen Überblick zum Stand der Technik. Abschnitt 3 beschreibt die Grundlagen zum ARM Cortex-M3 und des eingesetzten

CGRAs. In Abschnitt 4 wird das Konzept der Integration dargestellt. Abschnitt 5 zeigt auf, wie die Integration durchgeführt wurde. Abschnitt 6 präsentiert die Ergebnisse und Abschnitt 7 fasst diesen Beitrag zusammen und gibt einen Ausblick für zukünftige Arbeiten.

2. Stand der Technik

Grobgranulare rekonfigurierbare Architekturen lassen sich in zwei Klassen einteilen: Die erste Klasse sind *loosely-coupled CGRAs*. Dabei agiert der CGRA als Co-Prozessor, welcher über einen Bus durch den Prozessor angesprochen wird. Dies ermöglicht eine starke Anpassbarkeit mit wenig Einschränkungen, jedoch kann es aufgrund von Bus-Latenzen zu Verzögerungen bei der Verwendung des CGRAs kommen. Die zweite Klasse sind *tightly-coupled CGRAs*. Diese werden direkt in die Prozessorpipeline integriert und erweitern unmittelbar, ohne Verzögerung durch Busse, die Ausführungseinheit um spezialisierte Instruktionen. [2]

Unsere Arbeit gehört zur Klasse der *tightly-coupled CGRAs*. Im Gegensatz zu CHESS [7], MATRIX [8] oder DySER [5] verwenden wir die damit mögliche Instruktionssatzerweiterung nicht zur Steigerung der Performanz, sondern planen die Zuverlässigkeit des Prozessors mit redundanten Berechnungen durch ein bereits gehärtetes CGRA [3] zu erhöhen.

3. Grundlagen

3.1. ARM Cortex-M3

Der ARM Cortex-M3 ist ein kommerzieller 32-Bit-Embedded-Prozessor aus der Familie der ARMv7 RISC-Prozessoren mit geringem Energieverbrauch, niedriger Gatterzahl, kurzer Interrupt-Latenz sowie kostengünstigem Debugging. Seine Pipeline ist aus den folgenden drei Stufen aufgebaut: Fetch, Decode und Execute. Er verfügt über insgesamt 16 General-Purpose Register. Der ARM Cortex-M3 implementiert den Thumb-2-Befehlssatz, welcher es ermöglicht 16-Bit- und 32-Bit-Instruktionen innerhalb eines Programms einzusetzen. [1]

Wir haben für unser Vorhaben den ARM Cortex-M3 in LISA modelliert (siehe Abschnitt 5.1).

3.2. Language for Instruction Set Architecture

Die *Language for Instruction Set Architecture* (LISA) ist eine Modellierungssprache für Prozessoren und deren *Instruction Set Architecture* (ISA). LISA beschränkt sich jedoch nicht nur auf die reine Verhaltensbeschreibung eines Befehlssatzes, sondern bietet zudem die Möglichkeit, die grundlegenden Hardwarekomponenten eines Prozessors wie Speicherarchitektur, Register, Pipeline und Busse zu modellieren. Somit schließt LISA die Lücke zwischen der reinen Beschreibung des Instruktionssatzes durch *Instruction Set Simulatoren* und *Hardware Description Languages* wie Verilog oder VHDL, welche Prozessoren zwar detailliert auf der Hardwareebene abbilden können, aber keine Befehlssatzmodellierung anbieten. LISA ist eine Zusammenstellung von C-Makros. [13]

In [6] wird die Modellierung eines ARM7-Prozessors beschrieben.

Aus einem in LISA modellierten Prozessor lassen sich mit Hilfe des *Synopsys Processor Designers* ein Compiler, Assembler, Simulator und eine Hardwarebeschreibung in VHDL oder Verilog erzeugen.

3.3. Configurable Reconfigurable Core

Der *Configurable Reconfigurable Core* (CRC) ist eine grobgranulare rekonfigurierbare Rechnerarchitektur. Der CRC besteht aus einer zweidimensionalen Matrix – aufgebaut aus sogenannten Processing Elements (PEs), die in einem Koordinatensystem mit Zeilen und Spalten organisiert sind. Die einzelnen PEs sind über ein Double Nearest Neighbor-Netzwerk mit einer Datenbreite von 32 Bit untereinander verbunden. Dies bedeutet, dass jedes PE mit seinem direkten Nachbarn in allen vier Himmelsrichtungen über vier Datenleitungen (zwei Ausgänge und zwei Eingänge) verbunden ist, wobei es hier keine diagonalen Verbindungen zwischen den PEs gibt (siehe Abbildung 4). [10]

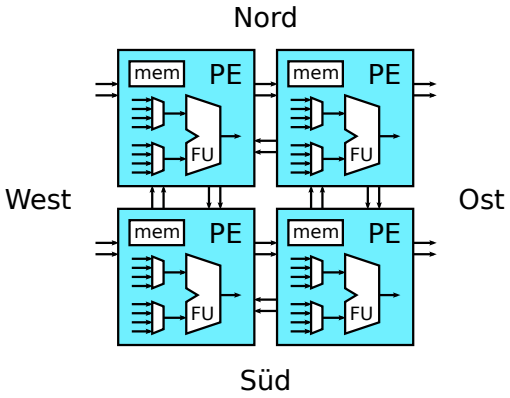

Abbildung 4: Configurable Reconfigurable Core

Die PEs verfügen jeweils über eine Functional Unit (FU), welche dort als ALU fungiert, Multiplexer zur Festlegung des Datenpfades, Register an den Ausgängen zur Übergabe der Daten sowie über einen Konfigurationsspeicher in welchem die Daten für das gewünschte Verhalten eines PEs abgelegt sind.

Die FU verfügt über insgesamt 16 nicht vorzeichenbehaftete Ganzzahl-Operationen:

- Vier arithmetische Operationen: Addieren, Subtrahieren, Multiplizieren, Dividieren.
- Sechs logische Operationen: und, oder, exklusiv oder, nicht, shift left, shift right.
- Sechs Vergleichsoperationen: gleich, ungleich, kleiner, größer, kleiner gleich, größer gleich.

Die Ausgänge der einzelnen Speicherzellen des Konfigurationsspeichers sind mit den Steuereingängen aller Multiplexer und dem Steuereingang der FU verbunden. Das Verhalten des gesamten CRC wird vollständig durch die Inhalte der Konfigurationsspeicher der einzelnen PEs festgelegt.

Über die Verbindung der PEs untereinander und der selektierten Operation der FUs ist es möglich komplexe Instruktionen zu modellieren. Somit kann ein in die Prozessorpipeline integrierter, gehärteter CRC für dynamische Redundanz genutzt werden, z. B. wenn die Execute-Stufe der Prozessorpipeline beschädigt ist. Es können mehrere einfache Instruktionen parallel ausgeführt werden. Dadurch ist es möglich den CRC für statische Redundanz mit TMR zu nutzen [12]. Ein zusätzlicher Prozessor kann zwar ebenfalls für statische Redundanz genutzt werden, bietet aber nicht die Möglichkeit der Befehlssatzerweiterung zur Laufzeit.

Die Verarbeitung von Daten durch den CRC startet implizit nachdem an die Eingänge der westlichsten PEs Daten angelegt wurden. Die angelegten und anschließend durch die PEs verarbeiteten

Daten benötigen jeweils einen Takt um ein PE zu durchlaufen. Die Daten werden, nachdem sie ein PE durchlaufen haben, in den Registern der Ausgänge gespeichert, bis sie im nächsten Takt durch das benachbarte PE verarbeitet werden.

4. Konzept

Unser Ziel ist es, ein CRC *tightly-coupled* in die Pipeline des ARM Cortex-M3 zu integrieren, um die Zuverlässigkeit dieses kommerziellen Prozessors zu erhöhen. Wir verwenden dazu ein CRC, welches im *single context*-Modus betrieben wird und ein vereinfachtes Modell des ARM Cortex-M3 welches unter Abschnitt 5.1 beschrieben wird.

Um dieses Ziel zu erreichen, muss der CRC durch den ARM Cortex-M3 rekonfiguriert werden können (siehe Abschnitt 5.2). Außerdem ist die Anbindung des CRC an die Registerbank des ARM Cortex-M3 notwendig, um ihn mit Daten zu versorgen und berechnete Daten zu speichern (siehe Abschnitt 5.3). Die Pipelinestruktur des Prozessors soll dabei erhalten bleiben, da ansonsten weitläufige Änderungen am Programmiermodell des ARM Cortex-M3 erforderlich wären.

In Abbildung 5 ist zu sehen, welche Komponenten hinzugefügt oder verändert werden müssen um den CRC in die Pipeline zu integrieren. Die Steuerung des CRC wird vollständig durch die Decode-Stufe der Pipeline übernommen. Somit kann der CRC parallel zur Execute-Stufe redundante Berechnungen durchführen oder im Fehlerfall die Arbeit der Execute-Stufe übernehmen.

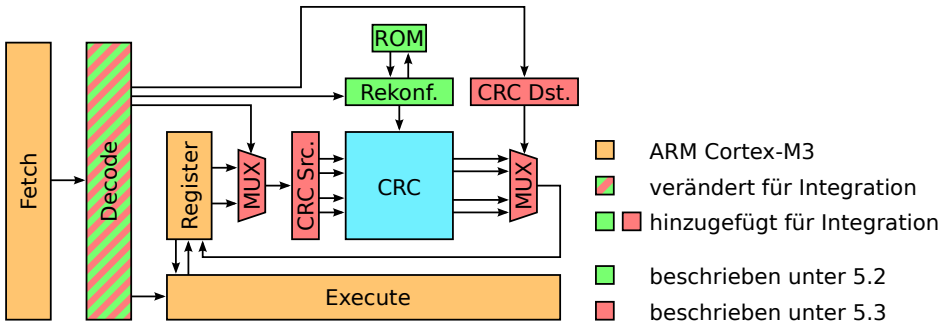

Abbildung 5: CRC integriert in den Pipeline des ARM Cortex-M3

5. Implementation

5.1. ARM Cortex-M3 Modell

Für die Integration des CRC in den ARM Cortex-M3 haben wir ein instruktionssatzkompatibles Modell des ARM Cortex-M3 in LISA implementiert.

Unser Modell hat im Vergleich zum Original eine eingeschränkte Funktionalität. Es wird ein vereinfachtes Speichermodell mit nur einem Taktzyklus Verzögerung verwendet. Die Latenzzeit des Speichers ist für die Performanz zwar relevant, spielt jedoch unter dem Aspekt der Zuverlässigkeit keine Rolle. Auch Co-Prozessor-Unterstützung wird nicht gewährleistet, diese wird jedoch nicht benötigt, da wir den CRC *tightly-coupled* in das Modell des ARM Cortex-M3 integrieren. Außerdem werden Interrupts, Exceptions und Debugging Support nicht unterstützt, da es sich dabei um Spezialfälle handelt, die in unserer konzeptuellen Umsetzung vorerst außer Acht gelassen werden. Der Thumb2-Befehlssatz wird hingegen nahezu vollständig unterstützt.

Dieses Modell des ARM Cortex-M3 kann mithilfe des *Synopsys Design Compilers* und der Nangate Open Cell Library in Version v2010_12 mit einer Strukturbreite von 45nm [9] erfolgreich synthetisiert werden.

5.2. Rekonfiguration durch die Pipeline des ARM Cortex-M3

Der CRC verfügt über einen Rekonfigurationscontroller, der dafür zuständig ist die CRC-Konfiguration aus einem externen Speicher auszulesen und in den Konfigurationsspeichern der PEs abzulegen. Dieser Rekonfigurationscontroller wird durch den ARM Cortex-M3 gesteuert. Aktuell verwenden wir als externen Speicher einen idealisierten ROM mit einer Datenbreite von 32 Bit und einem Taktzyklus Verzögerung.

`crcconfig`-Instruktion Um den Rekonfigurationsprozess des CRC aus der Pipeline des ARM Cortex-M3 zu starten, wird dessen Instruktionssatz um die `crcconfig`-Instruktion erweitert (siehe Tabelle 1), welche in der Decode-Stufe aktiv wird. Die Instruktion hat eine Registernummer als Argument. In diesem Register liegt die Speicheradresse der CRC-Konfiguration, die vom Rekonfigurationscontroller ausgelesen werden soll. Der Rekonfigurationsprozess läuft dann parallel zur Pipeline, wodurch andere Instruktionen zeitgleich zur Rekonfiguration des CRC ausgeführt werden können.

Syntax	Codierung
`crcconfig rd`	`111011 00 0000 0000 1111 0000 0000 rd`
`crcdata rd rm rn rp rq`	`111011 01 rd rm 1111 rn rp rq`
`crcrun`	`111011 10 0000 0000 1111 0000 0000 0000`

Tabelle 1: Syntax und Codierung der CRC-Befehle

5.3. Integration des CRC in den Datenpfad des ARM Cortex-M3

Um den CRC in den Datenpfad des ARM Cortex-M3 zu integrieren, wird der Thumb-2-Befehlssatz um die Instruktionen `crcdata` und `crcrun` erweitert (siehe Tabelle 1). Außerdem werden die zusätzlichen Register `CRC Source` und `CRC Destination` benötigt (siehe Abbildung 5).

`CRC Source` und `CRC Destination` Register Für jeden westlichen Eingang bzw. jeden östlichen Ausgang des CRC wird ein `CRC Source` bzw. `CRC Destination` Register angelegt. In den `CRC Source` Registern werden die Eingabewerte des CRC zwischengespeichert. Dadurch müssen die Eingangsdaten für einen CRC mit beliebiger Zeilen- und Spaltenzahl nicht bis zum Berechnungszeitpunkt in den General-Purpose Registern gehalten werden. Die `CRC Destination` Register steuern eine Multiplexer-Struktur welche die Ausgabedaten des CRC in die General-Purpose Register überträgt.

ISBN: 978-3-00-052380-9

`crcdata`-Instruktion Mithilfe der `crcdata`-Instruktion werden Eingabewerte und Zielregister für eine spezifische PE-Zeile des CRC definiert. Das Anlegen von Daten an mehrere PE-Zeilen erfordert also ensprechend mehrere `crcdata`-Instruktionen.

`crcdata` hat fünf Registernummern als Argumente. Der Inhalt des ersten Registers definiert die PE-Zeile für die die Instruktion gültig ist. Die Register zwei und drei enthalten die Eingangsdaten der PE-Zeile und werden in den `CRC Source` Registern abgelegt. Die beiden letzten Registernummern definieren, wohin die Ergebnisse der PE-Zeile geschrieben werden sollen und werden in den `CRC Destination` Registern gespeichert. Die `crcdata`-Instruktion wird in der Decode-Stufe aktiv.

`crcrun`-Instruktion Die tatsächliche Ausführung einer CRC-Berechnung wird von der `crcrun`-Instruktion übernommen. Diese prüft zunächst, ob sich der CRC aktuell in der Rekonfiguration befindet. Eine Ausführung während des Rekonfigurationsprozesses führt ansonsten zu unvorhersehbarem Verhalten des CRC. Es kann nicht dem Compiler überlassen werden, sicherzustellen, dass `crcrun` erst nach vollendetem Rekonfigurationsprozess aufgerufen wird, da sich dessen Dauer aufgrund von Speicherzugriffen zur Laufzeit ändern kann. Deswegen haben wir uns dafür entschieden die Pipeline des ARM Cortex-M3 während der Ausführung der `crcrun`-Instruktion solange anzuhalten bis der CRC fertig rekonfiguriert ist. Anschließend werden die Daten aus den `CRC Source` Registern an den CRC angelegt und die Berechnung dadurch implizit gestartet (siehe Abschnitt 3.3). Nach vollendeter Berechnung werden die Ergebnisse über die `CRC Destination` Register in die General-Purpose Register des ARM Cortex-M3 zurückgeschrieben.

Auch die `crcrun`-Instruktion wird in der Decode-Stufe der Prozessorpipeline aktiv. So ist es auch bei defekter Execute-Stufe möglich, den CRC zu rekonfigurieren und Berechnungen auf diesem durchzuführen.

6. Ergebnisse

6.1. Validierung

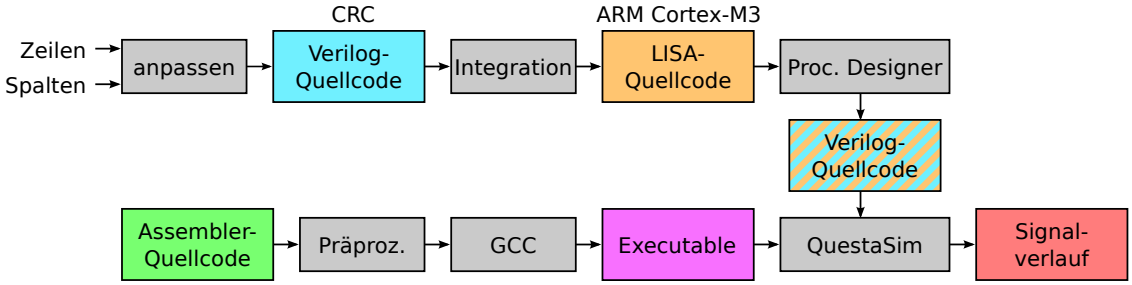

Abbildung 6: Toolchain

Unser neues Konzept sieht vor, dass durch die Integration eines CGRAs in die ARM-Prozessorpipeline die Möglichkeit zum redundanten Rechnen gegeben wird. Zur Validierung dieses Konzepts, d. h. um festzustellen, ob die Integration des CRC in die Pipeline des ARM Cortex-M3 korrekt ist, wird ein CRC mit zwei Zeilen und zwei Spalten in die Pipeline integriert. Eine hierfür erstellte Toolchain, in welcher die Zeilen- und Spaltenzahl des CRCs angegeben wird, wird genutzt, um die Integration des CRC Verilog-Moduls in das LISA-Prozessormodell des ARM

Cortex-M3 automatisch durchzuführen. Anschließend kann der ARM Cortex-M3 mit integriertem CRC mithilfe des *Synopsys Processor Designers* in ein Verilog-Design exportiert werden. Auf diesem Verilog-Design können anschließend mit der Software *QuestaSim* von *Mentor Graphics* Assembler-Programme für den ARM Cortex-M3 taktgenau simuliert werden. Dazu wird mit einem eigens für dieses Projekt entworfenen Präprozessor und dem Compiler aus der *GNU Compiler Collection* (GCC) Assembler-Quellcode für den ARM Cortex-M3 ausführbar gemacht. Der Präprozessor übersetzt die in Abschnitt 5 vorgestellten Befehle in `.word`-Direktiven, da der GCC Compiler die menschenlesbaren Ausdrücke `crcconfig`, `crcdata`, `crcrun` nicht kennt (siehe Abbildung 8). Mit *QuestaSim* können während der Simulation Registerinhalte und Signalverläufe für eine Auswertung aufgezeichnet werden. Abbildung 6 zeigt die komplette Toolchain.

Das Assembler-Programm, welches für die Validierung verwendet wird, rekonfiguriert den CRC über die `crcconfig`-Instruktion und legt anschließend mit der `crcdata`-Instruktion Daten an den CRC an. Schließlich wird mit der Instruktion `crcrun` die Berechnung durch den CRC gestartet (siehe Abbildung 8). Über die Signalaufzeichnungen von *QuestaSim* kann danach bestimmt werden, ob die Berechnungen des CRC korrekt sind. Es werden insgesamt vier Konfigurationen auf den CRC geladen: Nop, Routing, Multiply-Accumulate, dritte binomische Formel; diese sind in Abbildung 7 dargestellt.

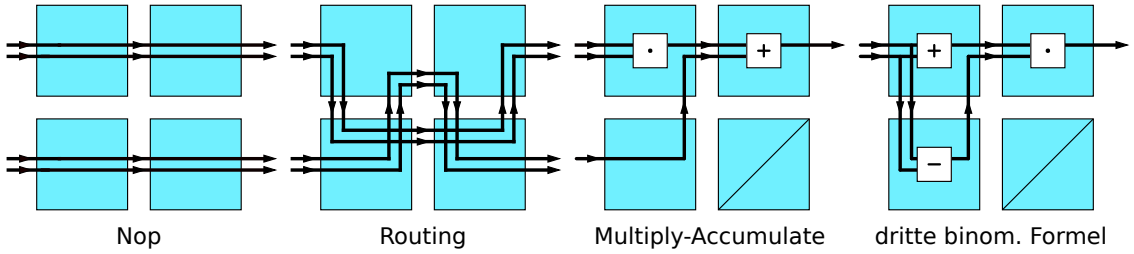

Abbildung 7: Konfigurationen für die Validierung des integrierten CRC

Abbildung 9 zeigt eine Aufzeichung der Inhalte der General-Purpose Register `r0` bis `r4` des ARM Cortex-M3 bei der Ausführung des Assembler-Quellcodes in Abbildung 8, hierbei wird der CRC mit Multiply-Accumulate konfiguriert. Zuerst wird mit `crcconfig` die Rekonfiguration des CRC gestartet. Parallel zur Rekonfiguration werden über `movs`-Instruktionen die Eingangsdaten für den CRC in den Registern `r0` bis `r3` abgelegt. Anschließend werden die Eingangsdaten mit den `crcdata`-Instruktionen in die `CRC Source` Register weitergeleitet. Nachdem die Rekonfiguration des CRCs abgeschlossen ist wird die Verarbeitung der Daten, durch den CRC, mit `crcrun` gestartet. Das Ergebnis der Berechnung wird zum Schluss in das Register `r4` geschrieben. Hier wird die Berechung $11 \cdot 3 + 9 = 42$ gezeigt.

Die Simulationen zeigen, dass der CRC innerhalb des ARM Cortex-M3 korrekt konfiguriert wird und die Berechnungen fehlerfrei durchgeführt werden.

Wir können hiermit zeigen, dass unser Konzept zur Integration einer grobgranularen rekonfigurierbaren Komponente in eine ARM-Prozessorpipeline erfolgreich umgesetzt werden kann. Mithilfe des integrierten CRC wird eine Steigerung der Zuverlässigkeit des ARM Cortex-M3 durch statische oder dynamische Redundanz prinzipiell ermöglicht.

```
[...]
main:
    movs r10, #0
    movs r11, #1
    movs r12, #2

    crcconfig r12

    movs r0, #11
    movs r1, #3
    movs r2, #9

    crcdata r10 r0 r1 r4 r5
    crcdata r11 r2 r3 r6 r7

    crcrun
    [...]
_exit:
    [...]
```

```
[...]
main:
    movs r10, #0
    movs r11, #1
    movs r12, #2

    .word 0xf00cec00

    movs r0, #11
    movs r1, #3
    movs r2, #9

    .word 0xf145eda0
    .word 0xf357edb2

    .word 0xf000ee00
    [...]
_exit:
    [...]
```

Abbildung 8: Ausschnitt des Assembler-Quellcodes zur Validierung der Integration (links vor der Verarbeitung durch den Präprozessor, rechts nach der Verarbeitung durch den Präprozessor)

6.2. Synthese

Das über die Toolchain erstellte Verilog-Design des ARM Cortex-M3 mit integriertem CRC wurde mit dem *Synopsys Design Compiler* erfolgreich synthetisiert. Als Technologie-Bibliothek haben wir ebenso wie in Abschnitt 5.1 die Nangate Open Cell Library in Version v2010_12 verwendet. Ein Takt von 10ns (enspricht 100MHz) wird an den Takteingang des ARM Cortex-M3 angelegt. Diese Timing-Bedingung kann für den ARM Cortex-M3 mit integriertem CRC mit einer Zeilen- und Spaltenanzahl von 2×2 bis 8×8 eingehalten werden.

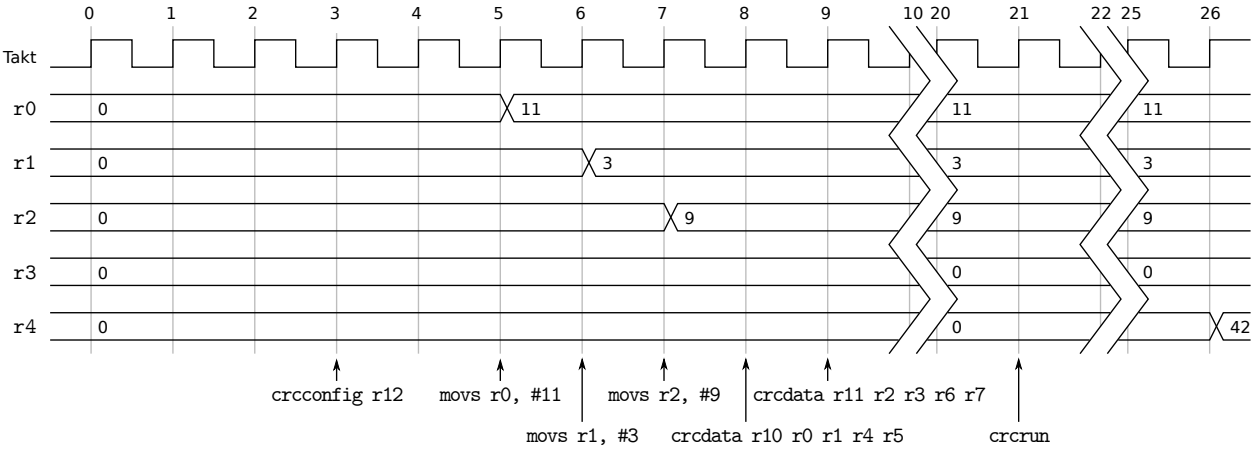

Abbildung 9: Aufzeichnung der Registerinhalte für Multiply-Accumulate

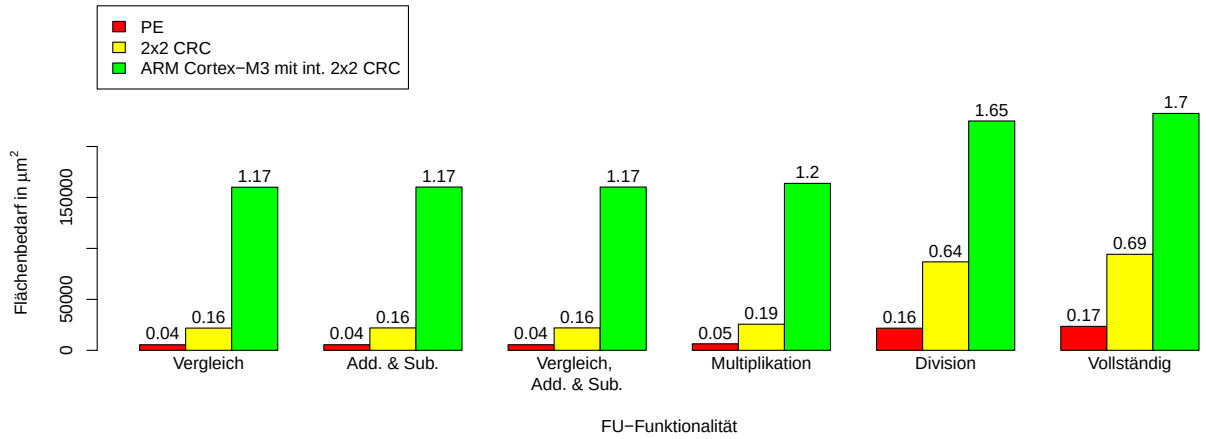

Abbildung 10: Flächenbedarf eines einzelnen PEs, eines 2×2 CRC und des ARM Cortex-M3 mit integriertem 2×2 CRC in Abhängigkeit der FU-Funktionalität. Die Zahlen oberhalb der Balken geben den Anteil des Flächenbedarfs an einem ARM Cortex-M3 ohne integrierten CRC an.

Flächenbedarf Durch die Synthese können auch Flächenbedarfsabschätzungen getroffen werden. Abbildung 10 stellt den Flächebedarf eines einzelnen PEs, eines 2×2 CRC und des ARM Cortex-M3 mit integriertem 2×2 CRC dar. Die Zahlen oberhalb der Balken geben den Anteil des Flächenbedarfs an einem ARM Cortex-M3 ohne integrierten CRC an, dieser liegt bei $136403 \mu m^2$. Es ist zu beachten, dass der Flächenbedarf mit der Anzahl der PEs wächst und die FU-Funktionalität bei der Integration eines $n \times m$ CRC einen deutlichen Einfluss auf den Flächenbedarf hat. Durch die Integration eines 2×2 CRC mit voller Funktionalität der FUs steigt die Fläche des ARM Cortex-M3 von $136403 \mu m^2$ auf $232416 \mu m^2$ an. Dies enspricht einer Zunahme um 70%. Dieser Flächenzuwachs kann jedoch reduziert werden, indem die Multiplikation und die Division aus der FU entfernt werden. Es ist möglich eine Division bzw. Multiplikation mit der Aneinanderreihung mehrerer Subtraktionen bzw. Additionen auf dem CRC zu realisieren. Somit könnte man auf die Division und Multiplikation in den FUs der PEs verzichten und den Flächenbedarf einschränken. Ebenfalls ist es denkbar nur bestimmte PEs mit einer Division bzw. Multiplikation auszustatten um Fläche einzusparen. Während geringere Funktionalität der FUs deutliche Flächeneinsparungen mit sich bringen führt eine höhere Funktionalität zu höherer Redundanz. Es gilt also anwendungsspezifisch abzuwägen welche Funktionalität den FUs in den PEs des CRC auf kosten der Fläche zugestanden werden sollen.

7. Zusammenfassung und Ausblick

In diesem Beitrag haben wir die Integration des CRC in den kommerziellen ARM Cortex-M3 Prozessor vorgestellt. Die Integration eines beliebig großen CRC wurde mithilfe unserer Toolchain vollständig automatisiert. Für die Integration wurde der Thumb-2-Befehlssatz um drei neue Instruktionen erweitert. Hierfür musste die Pipelinestruktur des ARM Cortex-M3 nicht verändert werden. Durch die Synthese konnte gezeigt werden, dass eine Übertragung auf einen FPGA prinzipiell möglich ist.

In zukünftigen Arbeiten soll ein Compiler verwendet werden, welcher entweder selbst sicherheitskritische Codeabschnitte identifiziert oder der Entwickler signalisiert dem Compiler über Pragmas

diese Bereiche welche dann redundant auf dem CRC ausführt werden sollen (statische Redundanz). Hierfür soll der CRC von *single context* auf *multi context* erweitert werden, wodurch der bislang zeitintensive Konfigurationsprozess drastisch beschleunigt werden kann. Eine Anbindung des Rekonfigurationscontrollers an den Hauptspeicher des ARM Cortex-M3 wird dafür sorgen, dass der ROM nicht mehr als Konfigurationsspeicher benötigt wird. Ferner soll erkannt werden, ob die Execute-Stufe des ARM Cortex-M3 defekt ist, damit der gehärtete CRC diese ersetzen kann, um weiterhin Berechnungen durchführen zu können (dynamische Redundanz).

Literatur

[1] ARM: *Cortex-M3 Technical Reference Manual*. r1p1 Auflage, 2005.

[2] Cong, J., H. Huang, C. Ma, B. Xiao und P. Zhou: *A Fully Pipelined and Dynamically Composable Architecture of CGRA*. In: *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, Seiten 9–16, Washington, DC, USA, 2014. IEEE Computer Society, ISBN 978-1-4799-5111-6. `http://dx.doi.org/10.1109/.10`.

[3] Eisenhardt, S., A. Küster, T. Schweizer, T. Kuhn und W. Rosenstiel: *Spatial and Temporal Data Path Remapping for Fault-Tolerant Coarse-Grained Reconfigurable Architectures*. In: *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Vancouver, Canada, Oktober 2011.

[4] Hartenstein, R.: *A decade of reconfigurable computing: a visionary retrospective*. In: *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, Seiten 642–649, 2001.

[5] Ho, C. H., V. Govindarajuz, T. Nowatzki, R. Nagaraju, Z. Marzecy, P. Agarwal, C. Frericks, R. Cofell und K. Sankaralingam: *Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation*. In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, Seiten 203–214, March 2015.

[6] Hoffmann, A., H. Meyr und R. Leupers: *Architecture Exploration for Embedded Processors with LISA*. Springer US, 1. Auflage, 2002.

[7] Marshall, A., T. Stansfield, I. Kostarnov, J. Vuillemin und B. Hutchings: *A Reconfigurable Arithmetic Array for Multimedia Applications*. In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, Seiten 135–143, New York, NY, USA, 1999. ACM, ISBN 1-58113-088-0. `http://doi.acm.org/10.1145/296399.296444`.

[8] Mirsky, E. und A. DeHon: *MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources*. In: *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, Seiten 157–166, Apr 1996.

[9] Nangate Inc.: *Library NangateOpenCellLibrary_typical Corner typical*. Thursday Feb 17 15:07 2011 Auflage, Feb 2011.

[10] Oppold, T., T. Schweizer, J.F. Oliveira, S. Eisenhardt und W. Rosenstiel: *CRC - Concepts and Evaluation of Processor-Like Reconfigurable Archtitectures*. In: *it - Information Technology, DOI: 10. 1524/itit, 49. 3. 157*, Band 49, 2007.

[11] Schweizer, T., A. Küster, S. Eisenhardt, T. Kuhn und W. Rosenstiel: *Using Run-Time Reconfiguration to Implement Fault-Tolerant Coarse Grained Reconfigurable Architectures*. In: *International Parellel and Distributed Processing Symposium Workshops (IPDPSW)*, Shanghai, China, Mai 2012. IEEE.

[12] Schweizer, T., P. Schlicker, S. Eisenhardt, T. Kuhn und W. Rosenstiel: *Low-Cost TMR for Fault-Tolerance on Coarse-Grained Reconfigurable Architectures*. In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, November 2011. IEEE.

[13] Zivojnovic, V., S. Pees und H. Meyr: *LISA-machine description language and generic machine model for HW/SW co-design*. In: *VLSI Signal Processing, IX, 1996., [Workshop on]*, Seiten 127–136, Oct 1996.