Dynamisch adaptive Mikroarchitekturen mit optimierten Speicherstrukturen und variablen Befehlssätzen

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS (Dr.-Ing.)

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik
am Karlsruher Institut für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-inform. Tanja Renate Harbaum

geboren in Wuppertal

Tag der mündlichen Prüfung:
25. Juni 2019

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr. rer. nat. Marc Weber

Dynamisch adaptive Mikroarchitekturen mit optimierten Speicherstrukturen und variablen Befehlssätzen

1. Auflage Juli 2019

Für Till, Maya, Fabian und Ida.

Abstract

The demands laid down microarchitectures are increasing steadily, and much of the technological innovation of recent decades has only been made possible by the progress of the semiconductor industry and the accomplished increases in integrated circuit performance. A further increase of the performance of integrated circuits is no longer self-evident, because physical limits will be reached soon. New architectures have to be designed in order to meet the increasing demands at this point.

Within the scope of this work, flexible microarchitectures were designed and evaluated, which optimize various application-specific parameters of the architecture. The designed microarchitectures meet the constraints by a novel and efficient usage of the available resources. A content adaptive memory structure for efficient processing of pre-analyzed data were designed. The designed concept can be used flexibly, because of the automatic generation, in addition it is adaptable and shows the potential, which lies in a shifting of the complexity from the application at runtime to analyzes in advance. Another approach is the concept of transparent and dynamic hardware acceleration of an adaptive processor. An automatism were designed to realize this concept and integrated into an reconfigurable processor. Thus the processor is able to detect and accelerate compute-intensive kernel independently at runtime. In this way, the adaptive processor not only combines the generosity of a general-purpose processor with the flexibility of a reconfigurable system, but is also capable of accelerating applications at runtime independently of the software developer or compiler. This leads to an independent adaptability of the processor to the application and thus enables a performance increase of a running kernel, which was not considered during the development phase of the processor.

Overall, with a relatively low development effort, powerful and flexible microarchitectures can be designed and realized, if the focus is on the efficient usage of available resources.

Zusammenfassung

Die Anforderungen, die an Mikroarchitekturen gestellt werden, steigen stetig, ein Großteil der technologischen Innovationen der letzten Jahrzehnte ist erst durch den Fortschritt der Halbleiterindustrie und den damit verbundenen Performanzsteigerungen integrierter Schaltkreise möglich geworden. Eine weitere Performanzsteigerung integrierter Schaltkreise ist durch das Erreichen von physikalischen Grenzen nicht mehr selbstverständlich. Es müssen neue Architekturen entworfen werden, um an diesem Punkt auch weiterhin die steigenden Anforderungen erfüllen zu können.

Im Rahmen dieser Arbeit wurden flexible Mikroarchitekturen entworfen und evaluiert, die applikationsspezifisch verschiedene Parameter der Architektur optimieren. Die entworfenen Mikroarchitekturen erfüllen die gestellten Anforderungen durch eine neuartige und effiziente Nutzung der vorhandenen Ressourcen. Es wurde eine inhalts-adaptive Speicherstruktur entworfen, welche für eine effiziente Verarbeitung von im Voraus analysierten Daten ausgelegt ist. Das entworfene Konzept bleibt durch die automatische Generierung flexibel einsetzbar und ist adaptierbar. Das System zeigt zudem das Potential auf, welches in einer Verschiebung der Komplexität des Anwendungsfalls zur Laufzeit auf Analysen im Vorfeld liegt. Ein weiterer Ansatz ist das Konzept einer transparenten und dynamischen Hardwarebeschleunigung eines adaptiven Prozessors. Für die Realisierung wurde ein Automatismus entworfen und dem Prozessor zur Verfügung gestellt, mit der dieser eigenständig zur Laufzeit rechenintensive Kernel detektieren und beschleunigen kann.

Auf diese Weise verbindet der adaptive Prozessor nicht nur die Generalität eines Allzweck-Prozessor mit der Flexibilität eines rekonfigurierbaren Systems, sondern ist zusätzlich in der Lage unabhängig vom Softwareentwickler oder Compiler Anwendungen zur Laufzeit zu beschleunigen. Dies führt zu einer eigenständigen Anpassungsfähigkeit des Prozessors an die Anwendung und ermöglicht somit eine Performanzsteigerung eines Kernels,

welcher während der Entwicklungsphase des Prozessors nicht berücksichtigt worden ist.

Zusammenfassend kann gezeigt werden, dass mit relativ geringem Entwicklungsaufwand leistungsstarke und flexible Mikroarchitekturen entworfen und realisiert werden können, wenn ein Hauptaugenmerk auf die effiziente Nutzung der vorhandenen Ressourcen gelegt wird.

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftliche Mitarbeiterin am Institut für Technik der Informationsverarbeitung (ITIV) der Fakultät Elektrotechnik und Informationstechnik (ETIT) am Karlsruher Institut für Technologie (KIT). Ein großer Anteil dieser Arbeit entstand zudem in enger Zusammenarbeit mit dem Institut für Prozessdatenverarbeitung und Elektronik (IPE). In den vergangenen Jahren habe ich viele neue Erfahrungen gesammelt, Einblicke in spannende Forschungsthemen erhalten, mich in der Lehre einbringen dürfen und vor allem viele interessante Menschen kennen gelernt. Ich möchte mich hiermit bei allen bedanken, die mich während meiner Promotionszeit unterstützt haben.

Ein ganz besonderer Dank gilt meinem Doktorvater Herrn Prof. Jürgen Becker, welcher mir nach dem Studium die Promotion am ITIV ermöglichte. Er unterstützte mich stets in meinen Vorhaben und ließ mir den Raum, meine eigenen Ideen zu verfolgen und auch zu verwirklichen.

Auch möchte ich mich herzlich bei Herrn Prof. Weber für seine hervorragende Betreuung meines Promotionsvorhabens und die Übernahme des Korreferats bedanken. Vielen Dank auch an die weitere Prüfungskommission, bestehend aus Herrn Prof. Cornelius Neumann, Herrn Prof. Michael Heizmann und Herrn Prof. Marwan Younis.

Ebenso danken möchte ich meinen tollen Kollegen und Kolleginnen am ITIV, mit denen es nie langweilig wurde. Allen voran meinen aktuellen Kollegen aus dem InvasIC Projekt Leonard Masing, Nidhi und Fabian Lesniak. Besonders erwähnen möchte ich an dieser Stelle Jens Becker und Andreas Lauber, die als langjährige Kollegen immer ein offenes Ohr für mich hatten und mir unterstützend zur Seite standen. Ganz besonders herzlich möchte ich mich bei Birgit Hirzler bedanken, die stets das Chaos am ITIV im Griff hat und einem immer weiter helfen kann. Auch Matthias Balzer, Andreas Kopmann, Christian Amstutz, Oliver Sander, Thomas Blank und Thomas Schuh vom IPE möchte ich herzlich danken für die

fachlichen Diskussionen, Gespräche auf Dienstreisen und dass ich mich am IPE stets wohl gefühlt habe.

Weiterhin möchte ich mich bei allen bedanken, die zur Erstellung dieser Arbeit beigetragen haben, sei es durch Anregungen, Diskussionen oder auch durch Aufmunterungen. Hier möchte ich vor allem Julia Zöhler und Beate Pal für ihre Unterstützung beim Korrekturlesen dieser Arbeit danken. Zudem gilt mein Dank auch den Studierenden, die ich während meiner Promotionszeit betreuen durfte und die einen Beitrag in diversen Themengebieten geliefert haben.

Schließlich möchte ich mich noch bei meiner Familie für die Unterstützung und vor allem auch die Ablenkung bedanken. Mein größter Dank gilt meinem Ehemann Till und meinen Kindern Maya, Fabian und Ida, die mir jeden Tag aufs Neue zeigen was wirklich wichtig ist.

Karlsruhe, im Juni 2019
Tanja Harbaum

Inhaltsverzeichnis

1 Einleitung

Bereits in den Dreißigerjahren wurde der Grundstein für die Entwicklung der heutigen Halbleiter gelegt, als der p-n-Übergang entdeckte wurde [93]. In den Vierzigerjahren wurde dann von den *Nokia Bell Labs* der erste Transistor entwickelt und in den Fünfzigerjahren entstand schließlich der erste Siliziumtransistor, welcher von *Texas Instruments* entwickelt wurde [18]. Noch bevor 1971 der erste Prozessor von *Intel* auf den Markt kam, wurde von Gordon Moore die These aufgestellt, dass sich die Zahl der Transistoren pro Flächeneinheit in regelmäßigen Abständen verdoppelt. Noch heute gilt diese Vorhersage, welche als das *Mooresche Gesetz* zu einem Leitsatz der Halbleiterindustrie wurde [83]. In Abbildung 1.1 wird die Prozessorentwicklung seit dem ersten Prozessor anhand verschiedener Parametern wie beispielsweise der Anzahl von Transistoren widergespiegelt. Die Anzahl von Transistoren, die auf einer Fläche integriert werden können, wird vor allem von der kleinsten fotolithografisch herstellbaren Strukturgröße bestimmt. Der erste Prozessor von 1971 benötigte nur 2 300 Transistoren bei einer Strukturgröße von 10 μm, bereits zehn Jahre später stieg die Anzahl von Transistoren auf 134 000 pro Prozessor-Chip und einer Strukturgröße von 1,5 μm [19]. Kommerzielle Mikroprozessoren und auch Mikroarchitekturen bestehen heute aus mehreren Milliarden Transistoren, welche eine Strukturgröße von bis zu 7 nm besitzen [118]. Die kommerzielle Herstellung integrierter Schaltungen mit einem 5 nm-Prozess soll innerhalb der nächsten Jahre starten [78] und das *Mooresche Gesetz* hat sich über die Jahrzehnte hinweg bewahrheitet, obwohl sein Ende immer wieder vorhergesagt wurde. Doch mittlerweile werden, neben wirtschaftlichen Grenzen, bei der Strukturgröße auch physikalische Grenzen erreicht. Die Forschung ist mittlerweile gar bei Einzelatom-Transistoren angelangt [78] und somit ist ein Ende der Miniaturisierung in naher Zukunft zu erwarten. In Zuge dessen wird auch zwangsweise das *Mooresche Gesetz* sein Ende finden, ebenso wie das schon erreichte Ende des sogenannten *Dennard Scalings*, das eine gleichbleibende Leistungsdichte bei einer Reduktion der Transistorgröße

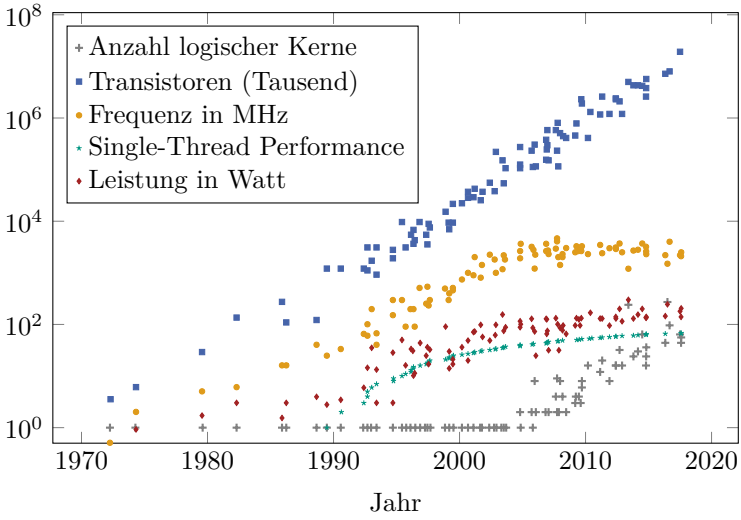


Abbildung 1.1: Prozessorentwicklung seit 1971 [94].

beschreibt. Das *Dennard Scaling* ermöglicht somit eine Steigerung der Taktfrequenz bei gleichbleibenden Stromverbrauch pro Fläche [42] und in Folge dessen ist bis vor rund zehn Jahren eine stetige Steigerung der Taktfrequenz, wie in Abbildung 1.1 zu erkennen, erzielt worden. Das Ende des *Dennard Scalings* verdeutlicht die physikalischen Grenzen, an denen die CMOS-Technologie stößt, und es muss nach weiteren Möglichkeiten gesucht werden, wenn die Performanz weiterhin gesteigert werden soll. Neue Technologien, wie die Quanten- und Nanotechnologie, könnten langfristig eine Alternative darstellen, doch jede neue Technologie erfordert eine lange Vorlaufzeit und nachhaltige Forschung und Entwicklung von mehreren Jahrzehnten [99]. Ein weiterer Trend ist die 3D-Technologie, welche mehrere Siliziumschichten miteinander verknüpft und somit die logische Dichte erhöht und die Datenwege minimiert [99]. Viele am Markt verfügbare Speicherbausteine beinhalten mittlerweile eine solche sogenannte *Chip-Stacking* Technologie und zeigen das Potential auf, welches in der Umstrukturierung von bestehenden Mikroarchitekturen steckt. Schon nach Ende des *Dennard Scalings* ist diese Entwicklung angestoßen worden und

ist in Abbildung 1.1 durch die steigende Anzahl von logischen Kernen zu erkennen. Mehrkernarchitekturen steigern jedoch nicht automatisch die Performanz, da die Anwendungen sich für eine parallele Ausführung eignen müssen und ein zusätzlicher Mehraufwand für eine massive Parallelisierung entsteht, welcher ebenfalls Gegenstand der Forschung ist [43]. Neben Mehrkernarchitekturen stellt seit einigen Jahren die Erweiterung von Prozessoren durch zusätzliche Beschleunigerkomponenten eine weitere Möglichkeit der Steigerung der Performanz dar. Teilweise werden solche Hardwarebeschleuniger auf dem Siliziumchip des Prozessors integriert, ein Beispiel hierfür sind die mittlerweile sehr verbreiteten *Graphics Processing Unit*s (GPUs), welche effizient und hochparallel Daten verarbeiten können [85]. Ebenso wie bei den Mehrkernarchitekturen muss die Anwendung sich aber auch bei dem Einsatz von GPUs für eine Parallelisierung eignen. Ein Ende der momentanen *Multicore*-Ära scheint schon in naher Zukunft zu liegen, da die Steigerung der Performanz nicht mit einer Steigerung an logischen Kernen skaliert [46]. Um momentan die durch den Wegfall des *Dennard Scalings* entstandene Lücke der Performanzsteigerung zu schließen, werden neue Mikroarchitekturen entworfen werden müssen, die effizient für eine Vielzahl an Anwendungen einsetzbar ist.

1.1 Motivation und Beitrag der Arbeit

Die Anforderungen, die an Mikroarchitekturen gestellt werden, steigen stetig, unabhängig von dem *Mooreschen Gesetz* oder dem Ende des *Dennard Scalings*. Ein Großteil der technologischen Innovationen der letzten Jahrzehnte ist erst durch den Fortschritt der Halbleiterindustrie und den damit verbundenen Performanzsteigerungen integrierter Schaltkreise möglich geworden. Das Voranschreiten der Forschung und auch die Integration von Technologie in der Gesellschaft erhöhen die Anforderungen ständig. Auf der einen Seite stehen hochkomplexe Forschungsanlagen, die unglaublich große Mengen von Daten erzeugen, die ohne eine geeignete Auswertung durch Rechenzentren keine Aussagekraft besitzen, auf der anderen Seite möglichst kleine und mobile Geräte, deren Stromverbrauch bei gleich bleibender Leistung minimiert werden sollen. Wie im vorangegangenen Abschnitt erläutert, tritt nun bald das Ende des *Mooreschen Gesetzes* ein und eine weitere Performanzsteigerung integrierter Schaltkreise ist

nicht mehr selbstverständlich. Es müssen neue Architekturen entworfen werden, um an diesem Punkt die steigenden Anforderungen erfüllen zu können. In dieser Arbeit werden anwendungsspezifische Mikroarchitekturen entworfen und evaluiert, die durch eine neuartige und effiziente Nutzung der vorhandenen Ressourcen die gestellten Anforderungen erfüllen. Im Vorfeld werden die zu speichernden Daten analysiert und die Speicherarchitektur wird infolge der in dieser Analyse bestimmten Parameter für diese Speicherinhalte spezifisch generiert. Eine Leistungssteigerung des Gesamtsystems soll hierbei durch eine im Vorfeld durchgeführte Analyse der Anwendung und somit angepasste Speicherstruktur erzielt werden. Als Anwendungsfall dient das Triggersystem des *Compact Muon Solenoid* Experimentes, welches besonders hohe Anforderungen an die Latenz und an die zu verarbeitende Datenrate stellt. Im Rahmen dieser Arbeit wird eine inhaltsadaptive Speicherstruktur entworfen, die innerhalb kürzester Zeit eine große Datenmenge auf bestimmte Merkmale hin untersucht. Anders als die bestehende, für diesen Einsatz spezifizierte Hardware, soll die im Rahmen dieser Arbeit zu entwerfende Architektur flexibel einsetzbar sein und auf Änderungen des Gesamtsystems eingestellt werden können. Neben der inhaltsadaptiven Speicherstruktur werden zudem adaptive Prozessoren untersucht, die eine dynamische Beschleunigung zur Laufzeit ermöglichen. Dieses Konzept soll um einen Automatismus ergänzt werden, der dem Prozessor eine automatische, transparente und zur Laufzeit dynamische Hardwarebeschleunigung ermöglicht. Ebenso wie bei der inhaltsadaptiven Speicherstruktur sollen hier die bereits vorhandenen Ressourcen und Strukturen effizienter genutzt werden. Der adaptive Prozessor soll für die Ausführung verschiedener Anwendungen optimiert werden, ohne die Vorteile eines *General-Purpose* Prozessoren zu verlieren oder deutlich mehr Ressourcen zu beanspruchen. Dies soll durch die Nutzung einer rekonfigurierbaren Einheit realisiert werden, die Hardwarebeschleuniger flexibel und feingranular zur Laufzeit programmieren kann. Ein Hauptaugenmerk liegt hierbei auf dem zur Verfügung gestellten Automatismus, der keine Unterstützung des *Compilers* oder Softwareentwicklers voraussetzt, und es dem Prozessor ermöglicht, eigenständig Anwendungen zu erkennen, die sich für eine Beschleunigung in einem dedizierten Beschleuniger eignen.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit ist in neun Kapitel unterteilt und wie folgend aufgebaut: In Kapitel 2 werden zunächst die für das Verständnis der folgenden Arbeit notwendigen Grundlagen erläutert. Es werden die zu Grunde liegenden Technologien beziehungsweise Rechnerstrukturen vorgestellt und die verschiedenen Anwendungsfälle für den Einsatz der zu entwickelnden Mikroarchitekturen dargelegt. In Kapitel 3 werden die aktuellen Systeme, welche als Umfeld für diese Arbeit dienen, aufgearbeitet um deren Merkmale für den Aufbau einer flexiblen Architektur nutzen zu können. Aufbauend auf diesen Erkenntnissen wird in Kapitel 4 eine FPGA Speicherstruktur entworfen und evaluiert, welche die zur Verfügung stehenden Ressourcen möglichst effizient nutzt und die gestellten Anforderungen an die Latenz erfüllen. Die entworfene Speicherstruktur wird eingehend analysiert und auf ihr Potential hin bewertet. In dem folgenden Kapitel 5 wird diese Speicherstruktur um einen Mechanismus erweitert, der die zu speichernden Daten mittels Heuristiken gewinnbringend modifiziert. Die zu erreichenden Ressourcenersparnisse werden analysiert und bewertet. In Kapitel 6 wird der Architektur eine Pipeline-Struktur hinzugefügt, welche eine parallele Verarbeitung mehrerer Datenströme ermöglicht. Zudem wird in diesem Kapitel eine Latenzanalyse für den gegebenen Anwendungsfall durchgeführt und die Methodik der entworfenen Speicherstruktur zusammenfassend bewertet. In Kapitel 7 wird ein adaptiver Prozessor um ein Konzept für eine automatische, dynamische und transparente Hardwarebeschleunigung erweitert. Die Umsetzung wird anhand der realisierbaren Instruktionen, benötigten Ressourcen, Latenzen und der zu erreichenden Beschleunigung evaluiert. Daran anknüpfend wird in Kapitel 8 ein modularer Beschleuniger für den adaptiven Prozessor entworfen, der die ihm zur Verfügung stehenden rekonfigurierbaren Ressourcen ausschöpft und für mehrere Anwendungsfälle einsetzbar ist. Abschließend wird in Kapitel 9 eine Bewertung der erzielten Ergebnisse vorgenommen und ein Ausblick gegeben.

2 Grundlagen

In diesem Kapitel werden alle Grundlagen, die zum Verständnis und zur Realisierung der vorliegenden Arbeit benötigt werden, vorgestellt.

2.1 Hochenergiephysik und der Large Hadron Collider

Die Europäische Organisation für Kernforschung (*Conseil européen pour la recherche nucléaire* (CERN)) wurde 1954 gegründet, um die Erforschung der Grundbestandteile der Materie in Europa zu fördern [68]. Seitdem ist die Anzahl der anfänglichen 12 Gründungsstaaten auf 22 Mitgliedstaaten im Jahre 2019 gestiegen [27, 56]. Über die Jahre hinweg wurden einige Teilchenbeschleuniger am CERN errichtet, unter anderem der erste Linearbeschleuniger am CERN LINAC 1 [1], der *Low Energy Antiproton Ring* LEAR[2] und der Große Elektron-Positron-Speicherring *Large Electron-Positron Collider* LEP[3]. Dieser wurde in dem Tunnel installiert, in dem heute der Große Hadronen-Speicherring *Large Hadron Collider* (LHC) eingerichtet ist. Abbildung 2.1 zeigt die zur Zeit aktiven Teilchenbeschleuniger und Experimente [82]. Diese Teilchenbeschleuniger bilden eine Beschleunigerkette und ein Teilchen wird schrittweise durch jeden dieser Beschleuniger geleitet und somit immer weiter beschleunigt. Zahlreiche fundamentale Erkenntnisse über den Aufbau der Materie und die Grundkräfte der Physik wurden am CERN errungen. Beispielsweise wurde 1983 experimentell erstmals das W- und Z-Boson nachgewiesen [37], ebenso wurde 2012 ein Teilchen gefunden, das in allen gemessenen Eigenschaften mit dem gesuchten Higgs-Boson übereinstimmt [35]. Neben den

[1] 1958-1992
[2] 1982-1996
[3] 1989-2000

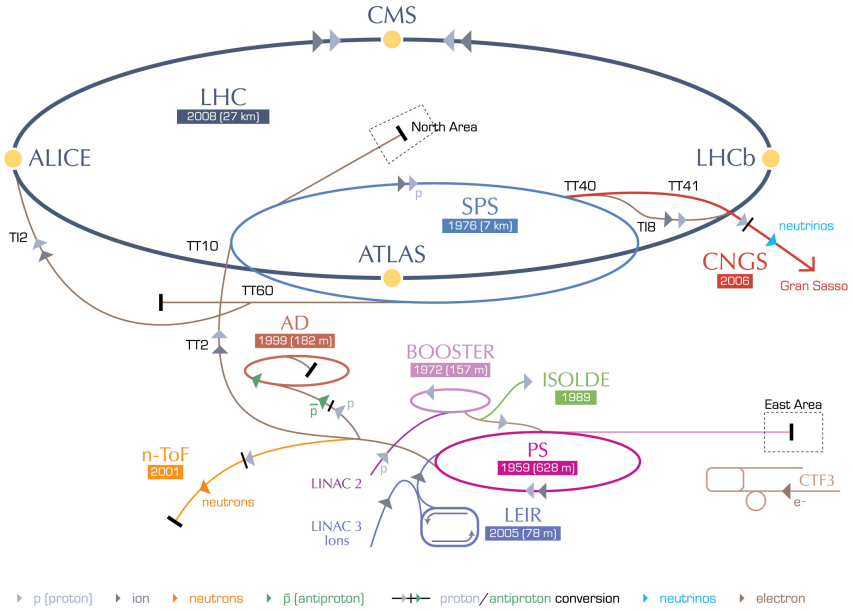


Abbildung 2.1: Übersicht der Teilchenbeschleuniger und Experimente am LHC [82].

Errungenschaften im Bereich der Physik wurden am CERN auch in der Computertechnik zahlreiche Entwicklungen angestoßen. So wurden beispielsweise 1989 die Grundlagen des World Wide Webs als Nebenprodukt der eigentlichen Forschungsarbeit entwickelt [16]. Um die enormen Datenmengen, welche an den Detektoren der einzelnen Teilchen-Experimente am CERN entstehen, auswerten zu können, wurde 2009 das *LHC Computing Grid* [20] entwickelt. Dieses stellt ein System für verteiltes Rechnen dar und weltweit sind über 140 Organisationen, vorwiegend aus dem universitären beziehungsweise forschenden Bereich, daran beteiligt. Auch mehrere Organisationen aus Deutschland sind involviert, unter anderem das *Karlsruher Institut für Technologie* (KIT) mit ihrem *Grid Computing Centre Karlsruhe* (GridKa) [95].

Der *Large Hadron Collider* ist momentan der leistungsstärkste Teilchenbeschleuniger der Welt und wurde 2009 am CERN in Betrieb genommen [47].

Der Ringtunnel, in dem der LHC liegt, hat einen Umfang von 27 km, liegt im Schnitt 100 m unter der Erde und überschreitet die Grenze zwischen Frankreich und der Schweiz. Der LHC beschleunigt Hadronen in zwei gegenläufigen Strahlen und es gibt vier Kollisionspunkte, an denen sich die Strahlen kreuzen. An diesen Kollisionspunkten liegen, wie in Abbildung 2.1 dargestellt, die Detektoren verschiedener Experimente. Die Wissenschaftler erhoffen sich vor allem neue Erkenntnisse bezüglich des Higgs-Bosons und eine Präzisierung des Standardmodells, welches alle bekannten Elementarteilchen und die Wechselwirkungen zwischen ihnen beinhaltet [73]. Um dies zu erreichen, werden die Hadronen (meist Protonen) immer weiter beschleunigt, um eine höhere Energie zu erreichen. Im Jahr 2010 wurde zum ersten Mal eine Energie von 3,5 TeV pro Strahl, also insgesamt eine Schwerpunktsenergie von 7 TeV erreicht [47]. Im Laufe der Zeit konnte diese Schwerpunktsenergie weiterhin schrittweise gesteigert werden und 2015 wurden 13 TeV erzeugt. Nach einer weiteren Umrüstungsphase soll eine Schwerpunktsenergie von 14 TeV erzielt werden, somit würden die Protonen dann auf bis zu 99, 9999991% Lichtgeschwindigkeit beschleunigt werden [26]. Diese Protonen zirkulieren in dem Ring in Bündeln, sogenannten *Bunches*, und ein Bündel besteht aus 115 Milliarden Protonen. Dies ist nötig, da einzelne Protonen aufgrund ihrer geringen Größe nicht gezielt zu einer Kollision gebracht werden können und durch die hohe Anzahl der Protonen eine Kollision deutlich wahrscheinlicher statt findet. Zwei solche Bündel folgen in einem zeitlichen Abstand von 25 ns, was ungefähr 7,5 m entspricht, und zeitgleich werden 2808 Bündel in dem Ring zirkuliert [21]. Die Bündel bewegen sich in einer Vakuum-Röhre mit einem Durchmesser von 56 mm und werden durch über 9 500 supraleitenden Magneten auf dieser Ringbahn gehalten. Hierbei besitzt jedes dieser Bündel eine Länge von ungefähr 7,5 cm [64] und an den Kollisionspunkten werden die entgegengesetzt beschleunigten Bündel dann auf einer Fläche mit einem Querschnitt von 16,4 μm zusammengeführt. Trotz der hohen Anzahl von Protonen pro Bündel, kommt es durchschnittlich bei dieser Zusammenführung nur zu 20 Kollisionen von Teilchen. Diese zeitgleichen Kollisionen werden als *Pile-Up* bezeichnet. Um die Anzahl der Kollisionen, die Entstehung und somit auch die Entdeckung von neuen Teilchen zu erhöhen, wird der LHC stetig weiter entwickelt. Mit jeder Leistungssteigerung können neue Erkenntnisse gewonnen werden, die vorher unerreichbar waren. Um dies zu ermöglichen, stehen im Wesentlichen zwei Parameter zur Verfügung, die erhöht werden können: Die Schwerpunktsenergie der Kollision und die Luminosität. Mit

einer Schwerpunktsenergie von 14 TeV ist für den LHC aufgrund seiner Tunnelgröße eine physikalische Grenze erreicht, die nur durch eine Verstärkung des Magnetfeldes noch weiter gesteigert werden könnte. Auch hierzu gibt es Ansätze [106], die jedoch nicht in naher Zukunft umgesetzt werden. Stattdessen wird die Luminosität, welches die Anzahl der Teilchenbegegnungen pro Zeit und Fläche beschreibt, erhöht [22] und an dem Konzept *High Luminosity-Large Hadron Collider* (HL-LHC) (HL-LHC) gearbeitet. Die Luminosität $\mathcal{L}$ wird durch

$$\mathcal{L} = \frac{N_a \cdot N_b \cdot j \cdot v/U}{A} \tag{2.1}$$

definiert [90] und wird in $\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ gemessen. N_a und N_b repräsentieren die Anzahl der Teilchen pro Bündel, welche aufeinander treffen, und j die Anzahl der gleichzeitig im Ring befindlichen Bündel. v ist die Geschwindigkeit der Teilchen und U der Umfang des Ringes, sie bilden die Wiederholfrequenz $f = v/U$ mit der die Bündel aufeinander treffen. Der Strahlquerschnitts A beschreibt die Fläche am Kollisionspunkt. Bei einer Gauß-Verteilung der Teilchen um das Strahlenzentrum (mit der horizontalen und vertikalen Standardabweichungen σ_x und σ_y) kann A präzisiert werden durch:

$$A = 4\pi\sigma_x\sigma_y. \tag{2.2}$$

Im Jahr 2011 wurde beim LHC eine Luminosität $\mathcal{L} = 7,7 \cdot 10^{33}\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ erreicht [96]. Die Anzahl zu erwartender Ereignisse pro Zeiteinheit wird als die zu erwartende Ereignisrate $\dot{N}$ bezeichnet und wird durch die Luminosität $\mathcal{L}$ und den Wirkungsquerschnitt σ_p definiert:

$$\dot{N} = \sigma_p \cdot \mathcal{L} \tag{2.3}$$

Der Wirkungsquerschnitt σ_p ist hierbei ein Maß für die Wahrscheinlichkeit, dass zwischen einem einfallenden Teilchen und einem anderen Teilchen eine Reaktion stattfindet und wird in Barn (b^{-1}) gemessen. Ein Barn entspricht einer Fläche von $10^{-28}\mathrm{m}^2$ [90]. Die integrierte Luminosität $\mathcal{L}$ ist zudem ein Maß für die Anzahl der Kollisionen, die über einen gewissen Zeitraum stattgefunden haben:

$$\mathcal{L}_{int} = \int_{t_0}^{t_1} \mathcal{L} dt \tag{2.4}$$

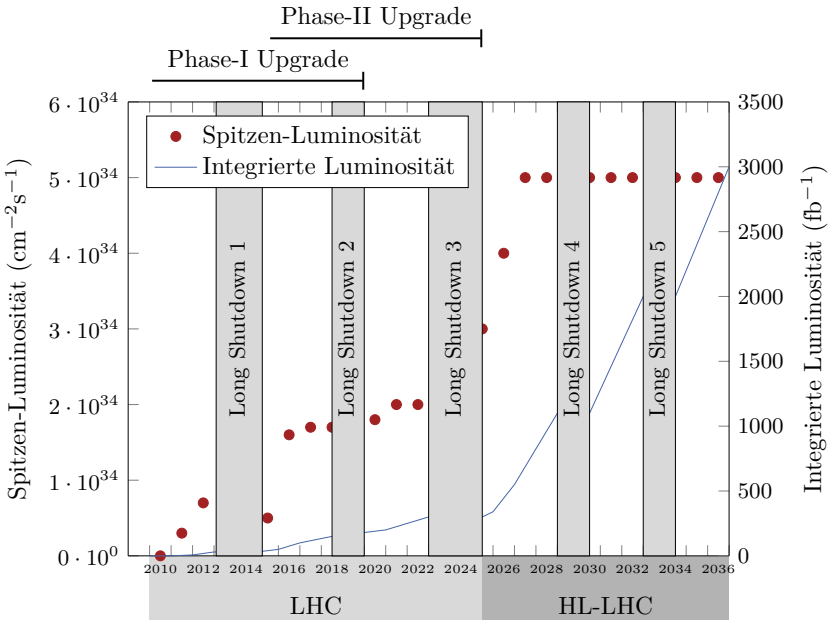


Abbildung 2.2: Plan für die Steigerung der Luminosität am LHC (Offizieller Stand 2019) [4].

Diese wird in inversen Barn (b^{-1}) gemessen und ein inverses Femtobarn entspricht hier in etwa 100 Billionen Proton-Proton-Kollisionen. Bevor der LHC in Betrieb genommen wurde, haben alle Hadronen-Speicherringe zuvor eine integrierte Luminosität $\mathcal{L}_{int} = 10\,\mathrm{fb}^{-1}$ erzeugt. Nach drei Jahren Betrieb hat der LHC diesen Wert bereits überschritten [96] und es sollen bis 2022 $300\,\mathrm{fb}^{-1}$ erreicht werden. Um dies zu erreichen muss die Luminosität weiterhin gesteigert werden. Die Wiederholfrequenz f kann, wie zuvor beschrieben, kaum noch erhöht werden. Eine Erhöhung der Anzahl j von Bündeln würde die Kollisionsrate erhöhen und somit die verschiedenen Experimente am LHC zu einem Austausch großer Teile der momentanen Detektoren zwingen. Somit bleiben die Anzahl der Teilchen pro Bündel N_a und N_b und die Fokussierung der Teilchen am Kollisionspunkt als mögliche Optimierungs-Parameter. Um die Anzahl der Teilchen pro Bündel zu

steigern wird an den Vorbeschleunigern und an der Injektion der Teilchen in den LHC gearbeitet, das heißt an der Übergabe der vorbeschleunigten Teilchen in den LHC. Zudem werden Quadrupolmagnet eingesetzt, um den Teilchenstrahl besser zu fokussieren. Hierzu sind unter anderem spezielle Hohlraumresonatoren geplant, die die Bündel kurz vor dem Kollisionspunkt so drehen, dass sie möglichst zentral kollidieren. Auf diese Weise soll der *Pile-Up* von derzeit durchschnittlich 20, also 20 Kollisionen bei einer Zusammenführung zweier Bündel, auf durchschnittlich 140 gesteigert werden. Es wird sogar ein Spitzenwert von bis zu 200 Kollisionen erwartet, was die Anzahl der erzeugten Sekundärteilchen enorm erhöht. Abbildung 2.2 zeigt die geplante Luminosität und die daraus resultierende integrierte Luminosität für den LHC, beziehungsweise den Nachfolger HL-LHC und die geplanten Betriebsunterbrechungen (*Long Shutdown*).

2.1.1 CMS Experiment

Das *Compact Muon Solenoid* (CMS) Projekt [33] ist eines der vier großen Experimente am LHC am CERN. Mehr als 6000 Mitarbeiter aus mehr als 50 Ländern sind an diesem Experiment beteiligt und haben einen Detektor entwickelt, welcher Proton-Proton-Kollisionen untersucht, die im Zentrum des Detektors auftreten. Ebenso wie der Detektor von *A Toroidal LHC ApparatuS* (ATLAS) gehört der CMS Detektor zu den Universaldetektoren des LHC mit deren Daten ein möglichst breites Spektrum an physikalischen Fragestellungen untersucht werden soll. Diese beiden Experimente ergänzen sich, da sie mit unterschiedlichen Technologien die selben Forschungsziele anstreben und sich somit gegenseitig verifizieren. Der CMS-Detektor ist 21,6 m lang, hat einen Durchmesser von 14,6 m und wiegt 14 000 t. Um möglichst viele verschiedenartige Teilchen detektieren zu können, besteht der Detektor aus mehreren Subdetektoren, welche zwiebelschalenartig um den Kollionspunkt herum angeordnet sind. In Abbildung 2.3 ist der schematische Aufbau des CMS-Detektors mit den wesentlichen Komponenten dargestellt. Der CMS-Detektor besteht hauptsächlich aus drei Teilen: Dem sogenannten *Inner Tracker*, der die Spur eines elektrisch geladenen Teilchen möglichst präzise darstellen soll, dem Kalorimetern für geladene und neutrale Teilchen und einem äußeren Ring mit Myonen-Detektor und Magneten. Diese drei Teile bestehen wiederum aus folgenden Subdetektoren, welche von der Strahlenachse aus beginnend aufgelistet sind:

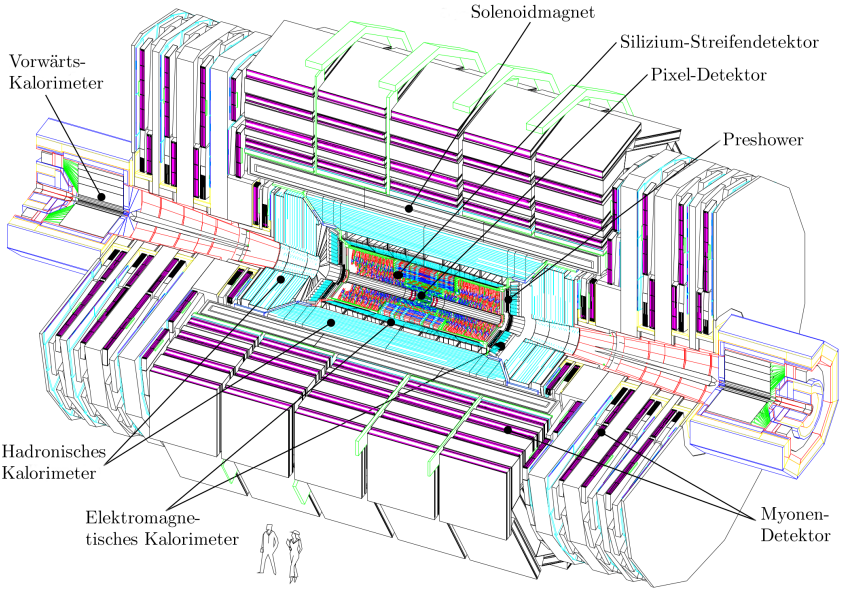


Abbildung 2.3: Schematische Darstellung des CMS-Detektors [34].

Pixeldetektor Dieser Detektor liegt direkt um die Strahlachse, ist rund 92 cm lang und besteht aus drei Lagen mit insgesamt 66 Millionen Pixeln [91]. Diese Pixel sind jeweils $100 \times 150\,\mu\text{m}^2$ groß und liefern eine Ortsauflösung von $5 - 10\,\mu\text{m}$ [2]. Dieser Subdetektor soll vor allem Teilchen identifizieren, die kurz nach der Entstehung zerfallen, wie beispielsweise b-Quarks und c-Quarks. Da diese Sensoren sich in einem Abstand von nur 4 bis 11 cm um die Strahlachse befinden [11], ist dieser Detektor einer sehr hohen Strahlenbelastung ausgesetzt und besitzt deswegen eine kurze Lebensdauer. Aus diesem Grund wurden die Sensoren bereits einmal vollständig ersetzt.

Silizium-Streifendetektor Mit über $200\,\text{m}^2$ Silizium ist der Streifendetektor der bisher größte Silizium-Streifendetektor der Welt und ist 5,8 m lang mit einem Durchmesser von 2,6 m [11]. Er liefert eine Ortsgenauigkeit von bis zu $10\,\mu\text{m}$ und besteht aus mehr als 15 000 Modulen [49]. In Unterabschnitt 3.1.1 wird näher auf die Funktionsweise dieses

Subdetektors eingegangen, da die Anordnung dieser Sensoren und die gelieferten Sensordaten im Rahmen dieser Arbeit eine wichtige Rolle einnehmen.

Elektromagnetisches Kalorimeter (ECAL) Dieses Kalorimeter misst die Energie von elektromagnetisch wechselwirkenden Teilchen wie Photonen und Elektronen. Dies geschieht durch Absorption solcher Teilchen in einem Szintillatormaterial. Hierzu werden über 75 000 Bleiwolframat-Kristalle ($PbWO_4$) verwendet, welche zur Gruppe der anorganischen Szintillatoren gehören, eine kurze Zerfallszeit im Nanosekundenbereich besitzen und über eine hohe Dichte verfügen [51]. Aufgrund dieser Eigenschaften werden die elektromagnetisch wechselwirkenden Teilchen innerhalb kürzester Zeit absorbiert und die kinetische Energie an die Bleiwolframat-Kristalle abgegeben. Diese Kristalle sind jeweils mit einer Photodiode bestückt [104], über welche dann das durch die Szintillatoren erzeugte Licht ausgelesen und somit die Energie der Photonen oder Elektronen mit einer Auflösung von weniger als einem Prozent bestimmt werden kann. Um eine bessere Teilchenidentifikation zu erlangen, wird zusätzlich noch ein sogenannter *Preshower*-Detektor aus Siliziumsensoren in den Endkappenbereichen integriert. Dieser besteht aus zwei orthogonalen Ebenen aus Siliziumstreifensensoren, die mit zwei Ebenen aus Bleiabsorbern verschachtelt sind [75]. Das elektromagnetisches Kalorimeter liegt zylinderförmig mit einem Abstand von 1,3 m um die Strahlachse, ist etwa 50 cm dick und 8 m lang.

Hadronisches Kalorimeter (HCAL) Die Energie von hadronischen Sekundärteilchen wie Neutronen und Protonen wird im hadronischen Kalorimeter gemessen. Dieses Kalorimeter besteht aus Messingplatten und Lagen von Szintillatoren, die abwechselnd zueinander angeordnet sind [24]. Die Szintillatoren werden über Wellenlängenschieber ausgelesen. Das Kalorimeter ist 9 m lang, hat einen äußeren Durchmesser von 6 m und ist einen Meter dick.

Solenoidmagnet Mit einer Länge von 12,5 m und einem inneren Durchmesser von 6 m ist der Solenoidmagnet der größte supraleitende Magnet, der je erbaut wurde [31]. Der Magnet erzeugt eine magnetische Flussdichte von 4 T und erhöht die Impulsauflösung für hochenergetische, geladene Teilchen, indem die Teilchen durch das

Magnetfeld gekrümmt werden. Der Solenoid umschließt die beiden Kalorimeter, dies hat den Vorteil, dass die Teilchen nicht den Magneten durchqueren müssen und dadurch an Energie verlieren, bevor sie in den verschiedenen Detektoren gemessen werden.

Eisenjoch Um den Solenoidmagneten herum befindet sich das Eisenjoch, welches mit Myonenkammern verwoben ist. Es ist 14 m lang, wiegt 10 000 t und ist nur für Myonen und schwach wechselwirkende Teilchen wie Neutrinos durchgängig [69].

Myonspektrometer Wie in Abbildung 2.4 abgebildet, werden fast alle Teilchen, abgesehen von den Myonen, bereits in den anderen Detektoren absorbiert. In den anschließenden Myonkammern wird der Impuls von dieser Myonen bestimmt, indem die durch das Magnetfeld bedingte Krümmung der Myonenbahnen gemessen wird. Die einzelnen Myonenkammern haben eine Länge von 2 bis 4 m und vier solcher Kammern befinden sich in einem Segment des Eisenjochrades [32]. Mit jeder dieser Myonenkammern kann ein dreidimensionales Spursegment rekonstruiert werden.

Vorwärts-Kalorimeter An beiden Außenseiten schließen Vorwärts-Kalorimeter den CMS-Detektor ab. Der Subdetektor ist 1,5 m lang und hat einen Radius zwischen 3, 7 und 14 cm um die Strahlachse herum [80]. Auch hier kommt ein Szintillatormaterial zum Einsatz. Wegen der Nähe zur Strahlachse wird Stahl und strahlungsbeständige Quarzfasern genutzt. Wenn ein Teilchen absorbiert wird, kann die Energie des Teilchen über eine Strahlung, die in den Quarzfasern entsteht, bestimmt werden.

In Abbildung 2.4 wird der Weg und die Absorption verschiedener Teilchen durch die diversen Subdetektoren des CMS-Detektors schematisch dargestellt. Die Spur der geladenen Teilchen (wie beispielsweise Myonen und Elektronen) wird durch das Magnetfeld gekrümmt und diese gekrümmte Spur wird durch den Siliziumstreifendetektor erfasst. Photonen werden durch das Magnetfeld nicht beeinträchtigt und auch nicht durch den Streifendetektor gemessen und werden dann in dem elektromagnetischen Kalorimeter absorbiert. Ebenso werden die Elektronen dort absorbiert, die Hadronen passieren diesen Detektor und werden dann im hadronischen Kalorimeter absorbiert. Nur die Myonen gelangen bis zu den in dem Eisenjoch verbauten Myonenkammern und werden dort detektiert.

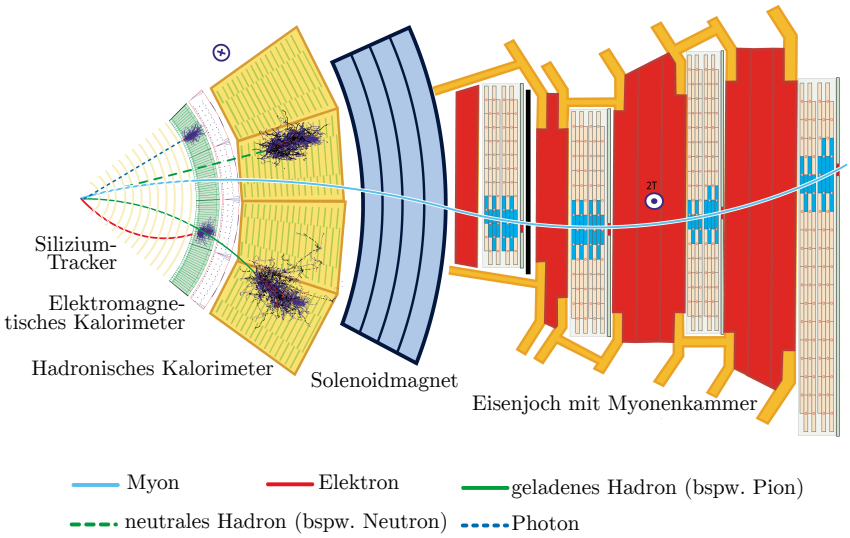


Abbildung 2.4: Wege verschiedener Teilchen durch den CMS-Detektor [13].

2.1.2 Koordinatensystem

Es wird ein rechtshändiges Koordinatensystem zur eindeutigen Bezeichnung der Positionen in dem CMS-Detektor verwendet. Der Ursprung des Koordinatensystems liegt im Kollisionspunkt. Die x-Achse zeigt zum Mittelpunkt des LHC-Ringes, die y-Achse senkrecht nach oben und die z-Achse verläuft entlang der Strahlachse entgegengesetzt des Uhrzeigersinnes des LHC-Ringes [23]. Anstelle des kartesischen Koordinatensystems wird eine Präsentation mit räumlichen Polarkoordinaten verwendet, die Transversalebene wird durch die x- und y-Achse gebildet und definiert den Azimutwinkel ϕ. Der Polarwinkel θ liegt in der Längsebene, die durch die y- und z-Achse gebildet wird. Der Abstand eines Punktes zur Strahlachse wird durch den Radius r definiert und liegt in der (x, y)-Ebene. In Abbildung 2.5 ist das Koordinatensystem schematisch dargestellt. Auf der linken Seite ist der Detektor mit darüber gelegten Achsen und eingezeichneten Winkeln aus Sichtweise des LHC-Mittelpunktes zu sehen. Rechts ist der Detektor im Querschnitt mit Blickrichtung zur Strahlachse zu sehen.

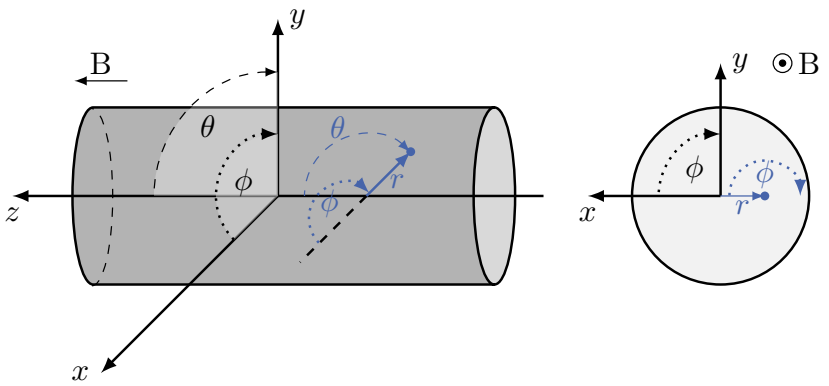
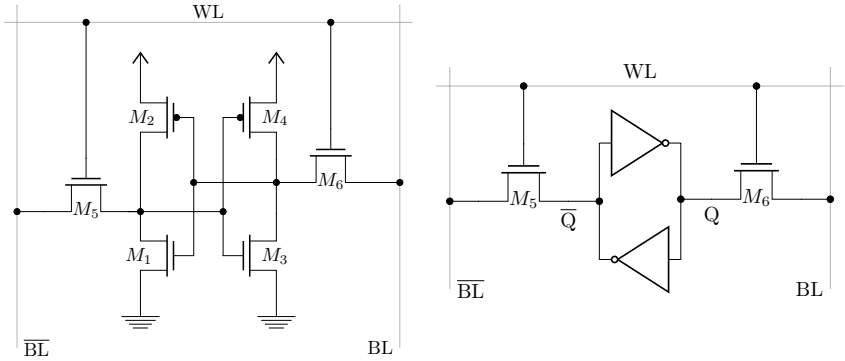


Abbildung 2.5: CMS-Koordinatensystem mit Azimutwinkel ϕ, Polarwinkel θ und Radius r.

Ein Punkt kann durch die drei Parameter Azimutwinkel ϕ, Polarwinkel θ und Radius r eindeutig beschrieben werden. In Abbildung 2.5 ist ein solcher Punkt mit seinen drei Paramtern in blau eingezeichnet. Anstelle des Polarwinkels θ wird häufig die Pseudorapidität η verwendet, die wie folgt definiert ist:

$$\eta = -\ln\left[\tan\left(\frac{\theta}{2}\right)\right] \tag{2.5}$$

Diese Darstellung hat den Vorteil, dass die Pseudorapidität η nahezu invariant unter einem Lorentz-Boost ist. Dies bedeutet, dass bei Hadron-Hadron-Kollisionen der Fluss der erzeugten Teilchen pro Pseudorapiditäts-Intervall fast konstant ist.



(a) 6-Transistor SRAM-Zelle in CMOS-Technik. **(b)** Darstellung mit Inverterschleife.

Abbildung 2.6: Aufbau einer SRAM-Zelle.

2.2 Speicherelemente

In den folgenden Abschnitten werden die Grundlagen erläutert, welche für das Verständnis der technische Realisierung von Speicherelementen benötigt werden. Diese Speicherelemente bilden die Basis vieler heutiger rekonfigurierbaren Architekturen und Speicherstrukturen.

2.2.1 Statisches RAM

Ein statisches RAM (englisch *Static Random-Access Memory* (SRAM)) gehört zur Gruppe der flüchtigen Speicherelemente und zeichnet sich durch kurze Zugriffszeiten und hohe Taktfrequenzen aus [97]. In Abbildung 2.6 wird der Aufbau einer Standard SRAM-Zelle mit sechs Transistoren dargestellt. Transistoren sind spannungsgesteuerte Schaltungselemente und die Ansteuerung erfolgt über die sogenannte *Gate-Source*-Spannung. Bei den Transistoren M_2 und M_4 handelt es sich um p-Kanal MOSFETs, welche negative Schalter realisieren [77]. Die restlichen Transistoren sind n-Kanal MOSFETs, welche positive Schalter darstellen und bei einer positiven *Gate-Source*-Spannung leiten. Die Transistoren M_1 und M_2 bilden hierbei

ebenso wie die Transistoren M_3 und M_4 zusammen einen Inverter und bilden somit eine Inverterschleife, welche in Abbildung 2.6b dargestellt wird. Diese Inverterschleife ist für die eigentliche Speicherung des Inhaltes der Speicherzelle zuständig und verstärkt den aktuellen Zustand des jeweils anderen Inverters, solange eine Betriebsspannung anliegt und die *Wordline* WL nicht geschaltet ist. Dieser Zustand wird *Standby* genannt und die Transistoren M_5 und M_6 sind in diesem Fall nicht leitend. Dies hat zur Folge, dass keine Verbindung zwischen der Speicherzelle und den *Bitlines* BL und $\overline{\text{BL}}$ besteht. Bei einem Lesezugriff werden zunächst die beiden *Bitlines* auf die Hälfte der Betriebsspannung aufgeladen und dann die *Wordline* WL geschaltet, um die Schreib-Leseschalter M_5 und M_6 zu schalten. Dieser Vorgang verbindet die zwei Inverterschleifen mit der jeweiligen *Bitline* und der Speicherinhalt Q der Zelle kann nun ausgelesen werden. Um die Speicherzelle mit einem Wert zu beschreiben, wird dieser auf die *Bitline* gelegt. Anschließend wird die *Wordline* WL geschaltet und somit wiederum die Inverterschleife mit den *Bitlines* verbunden. Der anliegende Wert wird dann in die Speicherzelle geschrieben, da die Größe der Transistoren, welche die Inverterschleifen bilden, so gewählt wurde, dass sie durch die *Bitlines* überschrieben werden.

2.2.2 Assoziativspeicher

Bei einem inhaltsadressierten Speicher oder auch Assoziativspeicher (englisch *Content Addressable Memory* (CAM)) handelt es sich um eine spezielle Speicherstruktur, die mittels eines Speicherinhaltes angesprochen wird und nicht wie üblich über eine Speicheradresse. Abbildung 2.7 stellt schematisch den konzeptionellen Aufbau eines Assoziativspeichers dar: Ein n Bit Wort wird angelegt und parallel Bit für Bit über die sogenannten *Searchlines* an die eigentliche Speicherzelle weiter geleitet [86]. Falls der angelegte Wert an der *Searchline* mit einem Speicherwort übereinstimmt, wird dies über die *Matchline* an einen Encoder weiter gegeben. Dieser gibt dann die Adresse der übereinstimmenden Speicherzelle weiter. Abbildung 2.8 zeigt den Aufbau einer solchen CAM-Zelle, die in diesem Fall aus einer SRAM-Zelle und den zusätzlichen Vergleichs-Transistoren M_7 bis M_{10} besteht [111]. Diese 10-T NOR Typ CAM-Zelle ist in der Lage innerhalb eines Taktzyklus zu ermitteln, ob der angelegte Wert, über die *Searchline*, mit dem Speicherinhalt der SRAM-Zelle übereinstimmt und dies gegebenenfalls

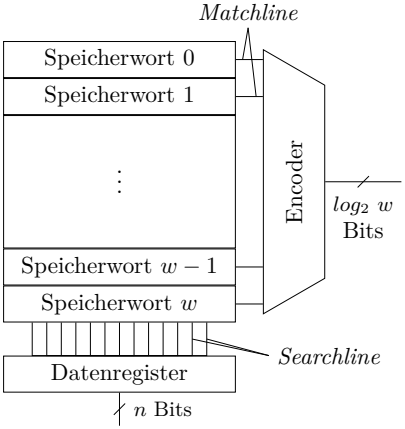


Abbildung 2.7: Funktionsprinzip eines Assoziativspeichers.

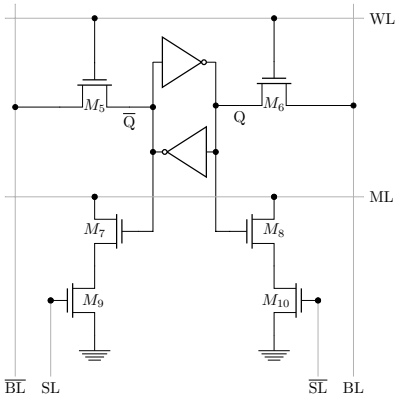


Abbildung 2.8: Aufbau einer CAM-Zelle mit CMOS Technologie.

über die *Matchline* zu indizieren. Die NMOS Transistoren M_7 und M_9 sowie M_8 und M_{10}, bilden hier jeweils ein *Pull down* Netz [77], so dass bei einer Nichtübereinstimmung von *Searchline* und gespeichertem Wert Q einer dieser zwei Netze die *Matchline* mit Masse verbindet. Eine Übereinstimmung der *Searchline* mit dem gespeichertem Wert Q deaktiviert hingegen beide Netze und trennt die Verbindung zwischen *Matchline* und Masse. In Abbildung 2.9 ist ein Assoziativspeicher bestehend aus drei Zellen dargestellt, welcher über eine NOR *Matchline* ausgewertet wird. Eine Suchanfrage an einen solchen Speicher erfolgt in drei Phasen: Zuerst werden die *Searchlines* vorab auf Masse gezogen, um die *Matchline* von der Masse zu trennen, danach wird die *Matchline* über den Transistor M_{pre} geladen. Danach werden die *Searchlines*mit dem Suchwort beschaltet und die *Matchline* kann ausgelesen werden [86]. Dies geschieht über einen Leistungsverstärker (englisch *Match-Line Sense Amplifier* (MLSA)) [9], welcher dann das Ergebnis des Vergleiches indiziert. Diese Weise der Auswertung liefert sehr schnell ein Ergebnis, da und die Überprüfung jedes Bits parallel geschieht und bei einer Nichtübereinstimmung eines einzelnen Bits direkt die *Matchline* über zwei Transistoren auf Masse gezogen wird.

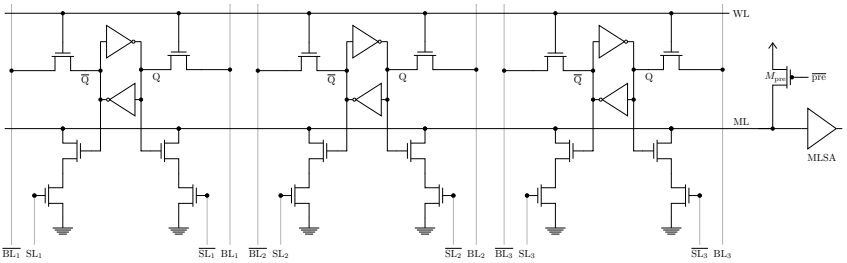


Abbildung 2.9: Aufbau eines Assoziativspeichers mit drei Zellen und einer NOR *Matchline*.

2.3 Skalierbare Mikroprozessorarchitekturen

In Kapitel 7 dieser Arbeit wird eine Mikroprozessorarchitektur mit einer adaptive Einheit so erweitert, dass sie zur Laufzeit auf Anforderungen reagieren und geeignete Beschleuniger laden kann. In diesem Kapitel werden die zum Verständnis nötigen Grundlagen erläutert und die verwendete Mikroarchitektur mit ihrem Instruktionssatz eingeführt.

SPARC (englisch *Scalable Processor Architecture* (SPARC)) bezeichnet eine Prozessor Architektur Variante, welche eine 32-bit *Reduced Instruction Set Computer* (RISC) Architektur zu Grunde liegt [55] und von Sun Microsystems entwickelte wurde. Ein SPARC Prozessor besteht aus einer Integer Einheit (englisch *Integer Unit* (IU)), einer Fließkomma Einheit (englisch *Floating-Point Unit* (FPU)) und einem optionalen Co-Prozessor (CP), jeweils mit eigenen Registern [100]. Dies erlaubt die Nebenläufigkeit dieser Einheiten und führt dazu, dass jede Einheit für sich konsistent ist. Die SPARC-Architektur ist in freier Lizenz verfügbar [112] und SPARC International, Inc unterstützt mehrere Implementierungen verschiedener Hersteller [54]. Die Befehlssatzarchitektur (englisch *Instruction Set Architecture* (ISA)) wird durch die SPARC-Spezifikation festgelegt [100], somit verfügen alle Implementierungen über den selben Befehlssatz. In Unterabschnitt 2.3.3 wird die LEON3-Implementierung von Gaisler Research [53] beschrieben, welche als Grundlage in der vorliegenden Arbeit verwendet wurde.

2.3.1 Registerfenster

Eine Besonderheit der SPARC Architektur stellt der Registersatz (englisch *Register File* (RF)) dar, welcher durch überlappende Register realisiert wird. Die für den Prozessor verfügbare Registerauswahl wird als Fenster bezeichnet, die Anzahl der Fenster kann zwischen 2 und 32 Fenstern variieren und jedes Fenster besteht aus 32 Registern. Dem Prozessor stehen jederzeit 8 globale Register (*globals*), 8 Eingangsregister (*ins*), 8 lokale Register (*locals*) und 8 Ausgangsregister (*outs*) zur Verfügung. Die Eingangs- und Ausgangsregister von zwei benachbarten Fenstern überlappen sich hierbei, somit können Ausgabewerte aus einem Fenster in das nächste übernommen werden, ohne dass diese kopiert werden müssen. Die Anzahl der möglichen Register variiert demnach zwischen 40 Registern bei zwei Fenstern und 520 Registern bei 32 Fenstern. Auf die globalen Register kann jederzeit zugegriffen werden und die Anzahl von acht dieser Register ist bei jeder Konfiguration identisch. In Abbildung 2.10 ist eine Konfiguration mit vier Fenstern (w_0 bis w_3) bestehend aus insgesamt 72 Registern dargestellt, wobei die globalen Register nicht eingezeichnet sind. Das aktuelle Fenster wird durch den *Current Window Pointer* (CWP) angezeigt und ist in der Abbildung 2.10 grau markiert. Dieser CWP kann nur durch SAVE-, RESTORE-Instruktionen oder Traps verändert werden. Bei einer SAVE-Instruktion wird der CWP dekrementiert und das nächste Fenster aktiviert. Analog hierzu wird bei einer RESTORE-Instruktion der CWP inkrementiert und das vorherige Fenster reaktiviert.

2.3.2 Instruktionssatz

Alle Instruktionen der SPARCv8 Spezifikation sind im 32bit-Format kodiert. Die drei möglichen Instruktions-Formate sind in Abbildung 2.11 dargestellt und werden durch das 31. und 32. Bit (Befehlscode, englisch *opcode* (op)) unterschieden. Diese Instruktions-Formate besitzen einheitliche Bereiche für den Befehlscode und die Adressen, dies erhöht die Code-Verdichtung und reduziert den benötigten Speicherbedarf[88]. Der Befehlscode unterteilt die Instruktionen in vier Bereiche:

op==1: CALL (*Call and Link*)

op==0: SETHI (*Set High* 22 Bits) & bedingte Verzweige-Instruktionen

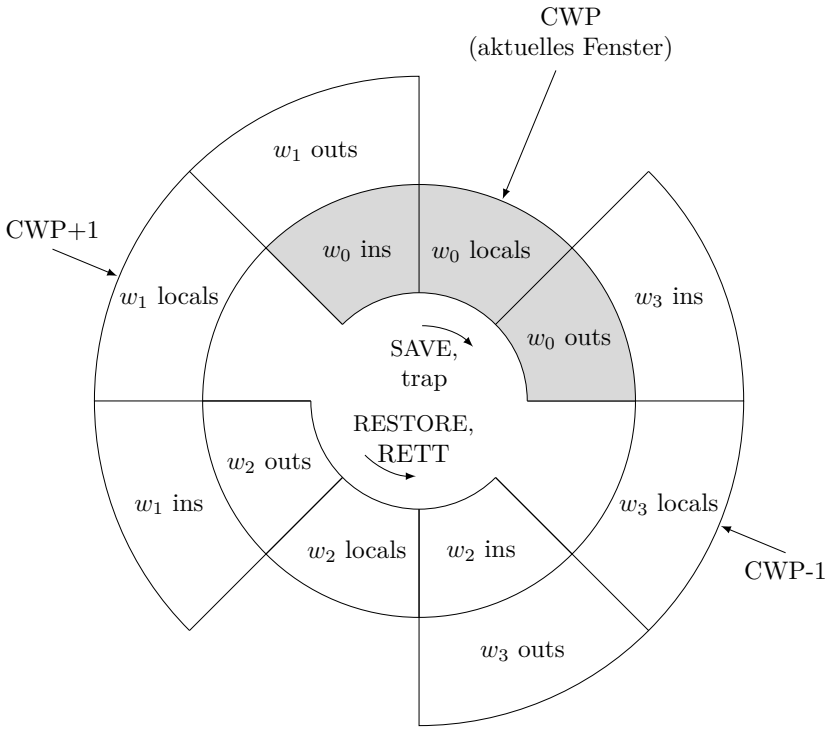


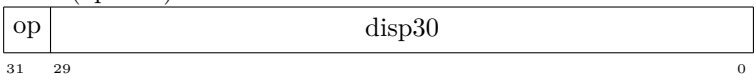
Abbildung 2.10: Beispiel überlappender Register-Fenster mit vier Fenstern.

op==3: Speicher-Instruktionen

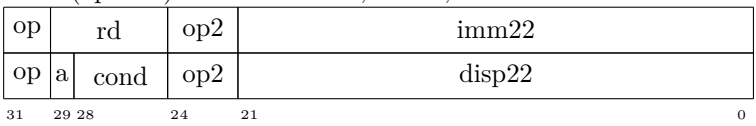
op==2: Arithmetische, logische und Schiebe-Operationen und verbliebene Instruktionen.

CALL-Befehle mit dem Befehlscode 1 stellen einen unbedingten *Program Counter* (PC)-relativen Sprung mit einer 30 Bit Verschiebungsvariablen dar und schreibt den Wert des PC in ein Ausgangsregister, sie nutzen das Instrukionsformat 1. Verzweige-Instruktionen und SETHI beinhalten den Befehlscode 0 und nutzen somit das Instruktionsformat 2. Ihnen steht jeweils ein 22 Bit großer Immediate zur Verfügung um bedingte

Format 1 (op==1): CALL



Format 2 (op==0): SETHI & Bicc, FBfcc, CBccc



Format 3 (op==2 oder op==3): übrige Instruktionen

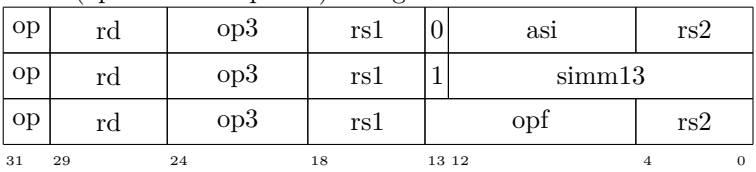


Abbildung 2.11: SPARCv8 Instruktionsformate [100].

PC-relative Sprünge zu realisieren oder wie bei SETHI die obersten 22 Bit eines Datenwortes zu setzen. Die verbleibenden Speicher-, Multiprozessor-, Arithmetisch-logischen- und Spezial-Instruktionen sind im Instruktionsformat 3 kodiert. Die meisten Befehle arbeiten mit zwei Register-Operanden (oder einem Register-Operanden und einer Konstante) und schreiben das Ergebnis in ein drittes Register, dieses Format wird Dreiadressformat genannt. Nur die Speicher-Instruktionen (Befehlscode 3) können auf das Speichermodell zugreifen, und es gibt zwei Arten um auf Speicherbereiche zuzugreifen: Entweder durch die Adressierung durch zwei Register (Registerwert plus Registerwert) oder durch ein Register mit Versatz (Registerwert plus Immediate) [100]. SPARC ist eine Big-Endian Architektur, dies bedeutet, dass das höchstwertige Byte zuerst gespeichert wird und somit die kleinste Speicheradresse besitzt. Die Speicheradresse eines Datenwortes ist gleich der Speicheradresse des höchstwertigen Bytes. Die kleinste adressierbare Einheit ist ein Byte, bestehend aus acht Bits und Datenwörter werden byteweise adressiert, wie in Abbildung 2.12 dargestellt. Ein Datenwort besteht aus vier Bytes und auf das höchstwertige Byte wird durch das Adressbit = 0 zugegriffen, auf das Byte mit dem niedrigsten Stellenwert mit dem Adressbit = 3. Analog dazu besteht ein

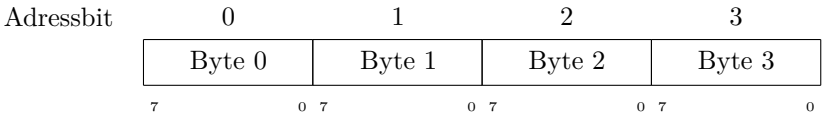


Abbildung 2.12: SPARCv8 Adressierungsschema.

Halbwort aus zwei Bytes und ein Datenwort kann aus zwei Halbwörtern zusammengesetzt werden. Ein Doppelwort besteht dementsprechend aus zwei Datenwörtern, bzw. acht Bytes, und das höherwertige Datenwort kann über das Adressbit=0 angesprochen werden. Auf das niederwertige Datenwort wird durch das Adressbit=4 zugegriffen, was das erste Byte des zweiten Datenwortes adressiert.

2.3.3 LEON3 Implementierung

Die LEON Prozessorfamilie wurde ursprünglich von der Europäischen Weltraumorganisation (englisch *European Space Agency* (ESA)) entwickelt und stellt das erste vollständige Mikroprozessor-Design zur Verfügung, welches unter einer Open-Source-Lizenz veröffentlicht wurde [112]. Der in dieser Arbeit verwendete Prozessor LEON3 ist eine Implementierung der SPARC V8 Architektur und ist unter der GNU *General Public License* (GPL) veröffentlicht. Dieser Prozessor ist Teil des *GRLIB* Softwarepaketes und wird laufend weiter entwickelt [41]. Dieses Softwarepaket enthält alle benötigten Quelldateien des Chipentwurfs (englisch *Intellectual Property Core* (IP-Core)), um ein komplettes Ein-Chip-System (englisch *System On Chip* (SOC)) zu entwerfen [29, 30]. In Abbildung 2.13 werden die wesentlichen Bestandteile des LEON3 als Blockdiagramm dargestellt. Der Instruktions-Speicher ist vom Daten-Speicher logisch und physisch getrennt, somit stellt der LEON3 eine Havard-Architektur dar und es ist möglich Instruktionen und Daten gleichzeitig zu laden. Die Minimalkonfiguration besteht aus einer siebenstufigen Pipeline, Registern, einem Daten- und Instruktions-Cache und einer *Advanced High-performance* Bus (AHB) Schnittstelle [41]. Diese Grundkonfiguration kann durch mehrere optionale Einheiten, wie einem Hardwaremultiplizierer, einer Speicherverwaltungseinheit (englisch *SPARC V8 Reference Memory Management*

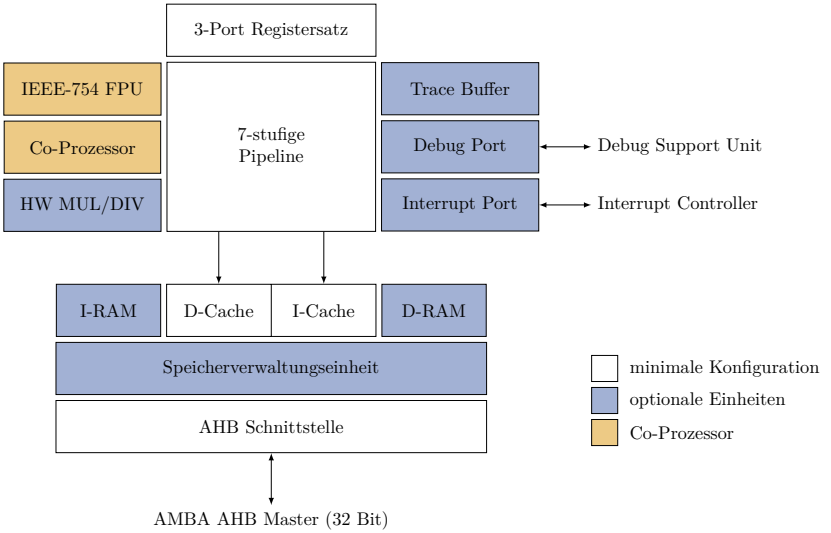


Abbildung 2.13: LEON3 Prozessor Blockdiagramm.

Unit (SRMMU)) oder weiteren Ports besipielsweise für Interrupts erweitert werden. Zudem existieren zwei Schnittstellen, die üblicherweise von einer *Floating-Point Unit* und einem Co-Prozessor genutzt werden. Neben dem SPARCv8 Instruktionssatz beinhaltet der LEON3 noch die CASA (englisch *Compare and Swap*) und UMAC (englisch *Unsigned Multiply and Accumulate*) beziehungsweise SMAC (englisch *Signed Multiply and Accumulate*) Instruktionen der SPARCv9-Spezifikation [101]. In den folgenden Abschnitten wird detailliert auf die für diese vorliegende Arbeit relevanten Einheiten der LEON3 Prozessorarchitektur eingegangen.

Pipeline

Die LEON3 *Integer Unit* (IU) realisiert die vollständige SPARCv8 Spezifikation mit einer siebenstufigen Pipeline und getrennten Instruktions- und Daten-Caches. Die Anzahl der Registerfenster ist standardmäßig auf acht gesetzt, kann aber zwischen zwei und 32 nach SPARCv8 Spezifikation variieren [29]. Standardmäßig verfügt der LEON3 somit über 136

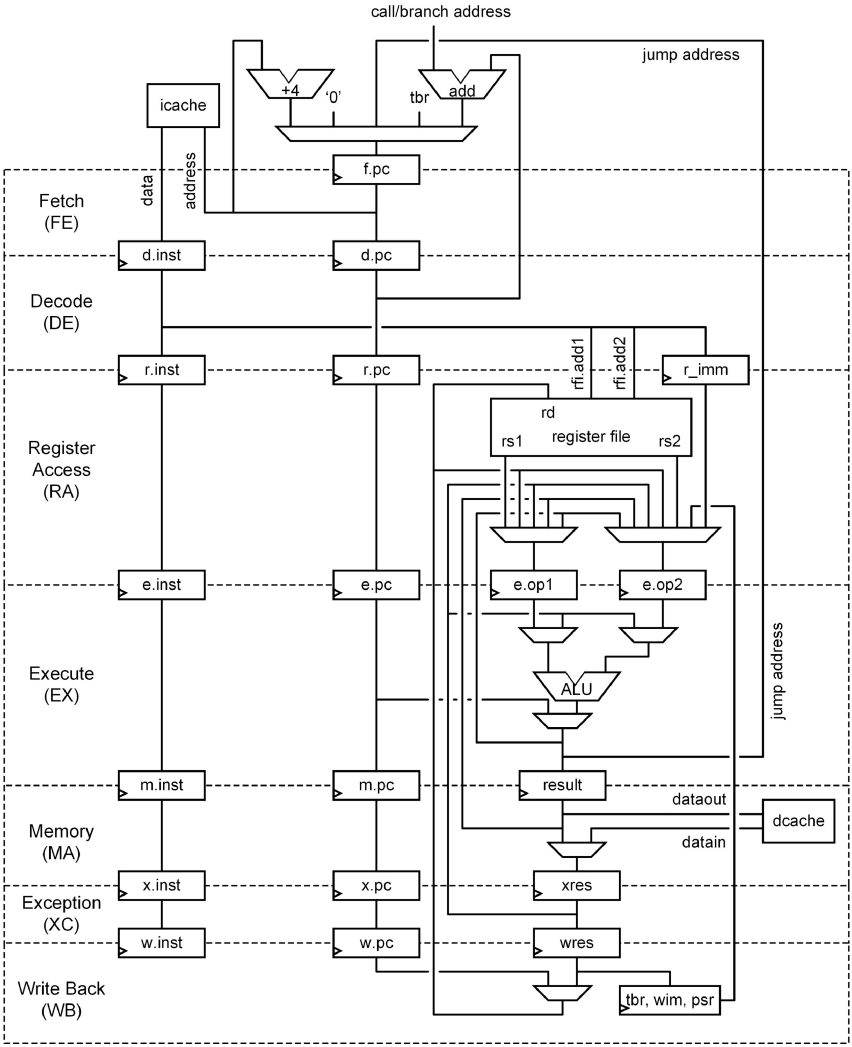


Abbildung 2.14: LEON3 *Integer Unit* Pipeline [12].

Register. Bei der Implementierung wurde der Fokus auf Leistungsfähigkeit und geringe Komplexität gelegt. Hierfür sind zur Beschleunigung der Multiplizier- und Dividieroperationen Hardwareeinheiten implementiert, welche *Multiply-Accumulate* (MAC) Operationen durchführen können. Zudem ist eine statische Sprungvorhersage realisiert, die davon ausgeht, dass der Sprung immer erfolgt (*branch always*). Dies bedeutet, dass der nächste Befehl immer geladen wird, auch wenn er eventuell nicht ausgeführt wird. Abbildung 2.14 zeigt den Datenpfad der Pipeline des LEON3 mit folgenden sieben Stufen [29]:

***Instruction Fetch* (FE)** Die Instruktion wird aus dem Cache oder aus dem AHB geladen. Am Ende der *Fetch* Stufe ist die Instruktion gültig und wird in der *Integer Unit* gespeichert.

***Decode* (DE)** Die Instruktion wird dekodiert und Sprungadressen für CALL und BRANCH Instruktionen werden erzeugt.

***Register Access* (RA)** Die Operanden werden aus dem RF oder internen Nebenleitungen geladen.

***Execute* (EX)** Arithmetische, logische und Schiebe-Operationen werden ausgeführt und für Speicher- und Sprung-Operationen werden die Adressen berechnet. Bei *Multicycle*-Instruktionen wird die Pipeline angehalten, bis die Berechnung abgeschlossen ist.

***Memory* (ME)** In dieser Stufe wird aus dem Datencache zugegriffen und Daten können gelesen oder gespeichert werden.

***Exception* (XC)** *Traps* und *Interrupts* werden gelöst und für lesende Cache-Zugriffe werden die Daten entsprechend ausgerichtet.

***Write* (WR)** Die Ergebnisse der arithmetischen, logischen und Schiebe-Operationen sowie der Speicher-Operationen werden in das RF zurückgeschrieben.

2.4 Rekonfigurierbare Architekturen

Heutige intelligente technische Systeme stellen hohe Anforderungen an Ressourceneffizienz, vor allem eine hohe Leistungsfähigkeit mit gleichzeitiger hoher Energieeffizienz stehen im Vordergrund. Dabei spielen Faktoren wie Flexibilität und Fehlertoleranz eine immer größere Rolle. Klassische

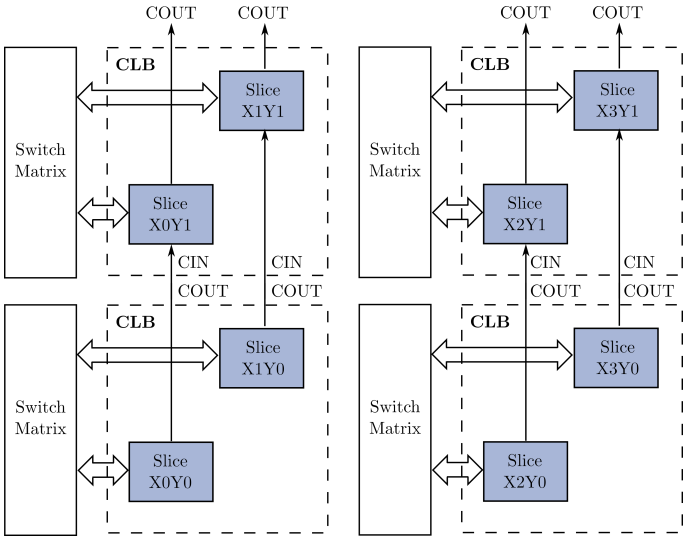


Abbildung 2.15: Zeilen- und Spaltenverbindungen zwischen *Configurable Logic* Blöcke (CLBs) und *Slices* [116].

Hardwarearchitekturen können diese Anforderungen meist nicht mehr erfüllen und der Trend geht zu rekonfigurierbaren Architekturen, die ihre interne Struktur anpassen können [72]. Die traditionell statische Aufteilung in Hardware und Software kann durch den Einsatz von rekonfigurierbaren Systemen wie beispielsweise *Field Programmable Gate Array* (FPGA) Plattformen, welche feingranulare konfigurierbare Logikelemente beinhaltet, aufgeweicht werden. Einerseits lassen sich bewährte, effektive Software-Entwicklungsmethoden für die Programmierung von FPGAs nutzen. Andererseits eröffnen FPGAs dem Software-Entwickler bisher ungeahnte Dimensionen der parallelen Rechenleistung [57]. In der vorliegenden Arbeit wird sowohl diese mögliche Parallelität genutzt, als auch die Anwendung an die Beschaffenheit des FPGAs angepasst. Aus diesem Grund wird in diesem Unterkapitel näher auf den Aufbau eines FPGAs eingegangen. Ein *Field Programmable Gate Array* ist ein Halbleiter-Logikbaustein, in dem logische Schaltungen geladen werden können und somit programmierbar sind. Ein FPGA besteht unter anderem aus einer großen Anzahl von

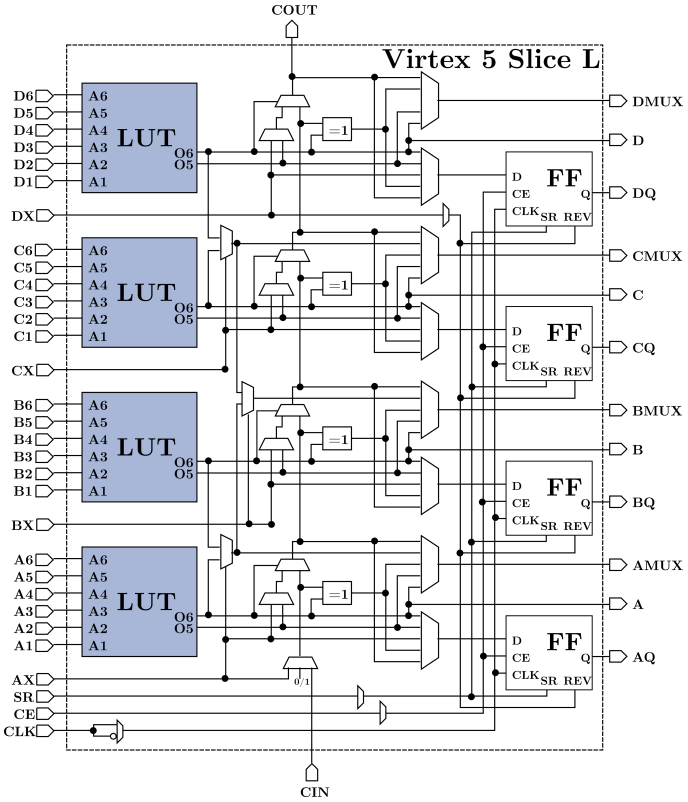


Abbildung 2.16: Diagramm eines Slices der Virtex 5 Familie [114].

logisch programmierbaren Einheiten (englisch CLB), die durch Switch-Matrizen miteinander verbunden sind. Die meisten kommerziellen Anbieter von FPGAs verwenden statische Speichertechnologien (SRAM) in ihren Geräten, um die Routing-Verbindungen und die logischen Einheiten zu programmieren [48]. Die SRAM-basierte Struktur ist aufgrund ihrer Umprogrammierbarkeit und der Verwendung von Standard-CMOS-Prozessen die dominierende Technologie bei FPGAs. In Abbildung 2.15 ist der Grundaufbau eines CLBs und deren Verbindungsstruktur für einen FPGA des Herstellers Xilinx dargestellt. Die Hauptbestandteile einer solchen Logikeinheit sind zwei sogenannte *Slices*, auf einem aktuellen FPGA der Virtex 7 Famile sind bis zu 300 000 solcher *Slices* auf einem Chip integriert [116]. Ein *Slice* besteht wiederum aus mehreren Basis-Bauteilen wie Flipflop, Multiplexern und vier sogenannter programmierbarer *Lookup*-Tabellen (LUTs). Diese Tabellen stellen das Herzstück eines FPGAs dar und speichern die eigentliche Logikfunktionen, die realisiert werden sollen. In Abbildung 2.16 ist ein *Slice* vom einfachen Typ L (Logik), der den Großteil der integrierten *Slices* ausmacht, dargestellt. Neben *Slices* des Typs L gibt es zudem Slices mit zusätzlichem *distributed RAM* und Shiftregistern, welche als Typ M (*Memory*) bezeichnet werden. Ein CLB besteht entweder aus zwei Slices des Typs M oder zwei Slices des Typs L, die integrierten *Lookup*-Tabellen unterscheiden sich jedoch nicht. Jede dieser *Lookup*-Tabellen hat sechs voneinander unabhängige Eingänge (A_1 bis A_6) und zwei unabhängige Ausgänge (O_5 und O_6). Solche *Lookup*-Tabellen mit sechs Eingängen (6-LUT) sind in modernen FPGAs verbaut und können folgende logische Funktionen implementieren:

(A) Eine beliebig definierte boolesche Funktion mit sechs Eingängen.

(B) Zwei beliebig definierte boolesche Funktionen mit fünf Eingängen, sofern diese beiden Funktionen gemeinsame Eingänge haben.

(C) Zwei beliebig definierte boolesche Funktionen mit maximal 3 und 2 Eingängen.

Die Standardstruktur einer *k-Lookup*-Tabelle verwendet 2^k SRAM-Zellen zum Speichern der Konfigurationsbits, welche die logische Funktion der jeweiligen *Lookup*-Tabelle beinhaltet. In Abbildung 2.17 ist der vereinfachte Grundaufbau einer 3-*Lookup*-Tabelle mit drei Eingängen A_1 bis A_3 und einem Ausgang O_1 skizziert. Es werden acht SRAM-Zellen benötigt, um eine beliebig definierte boolesche Funktion mit drei Eingängen

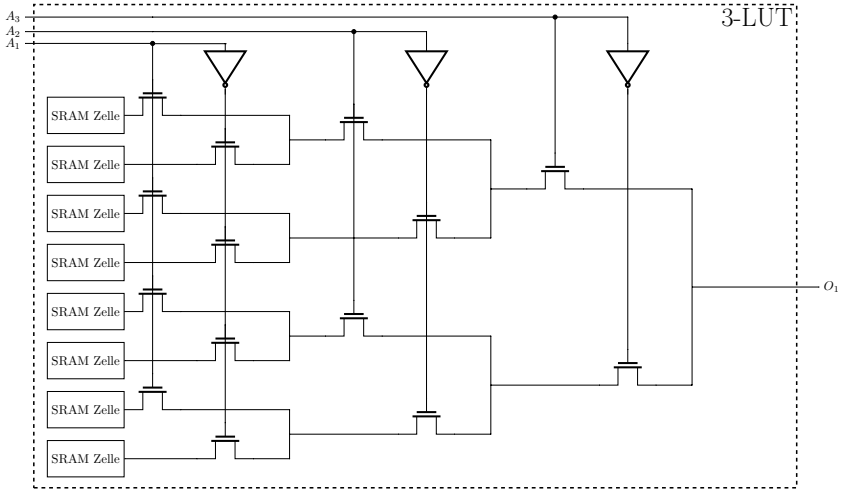


Abbildung 2.17: Aufbau eines Standard FPGA-Logikblocks [113].

zu implementieren. Die Eingänge schalten die Durchgangs-Transistoren und die angelegten Konfigurationsbits werden je nach Beschaltung der Eingänge zusammengeführt und zum Ausgang O_1 weiter geleitet.

2.4.1 Interne Speichereinheiten

Moderne FPGAs enthalten zusätzlich noch optimierte Makroblöcke für allgemeine Funktionen, wie digitale Signalprozessoren (engl. *Digital Signal Processor* (DSP)) oder auch zusätzlichen internen Speicher (*Block Random Access Memory* (BRAM)). Die neueren Generation der Virtex Plattform des Herstellers Xilinx bieten eine immer größere Anzahl dieser Speicherblöcke und es stehen bis zu 10 000 dieser Blöcke zur Verfügung [117], die jeweils zwei unabhängige 18 kB *Random Access Memory* (RAM) Blöcke enthalten. Diese BRAMs Blöcke sind in Spalten angeordnet und dadurch kaskadierbar, was sowohl eine höhere Wortbreite oder auch Speichertiefe erlaubt. Zudem bietet die Ultrascale Plattform noch neue Speicherblöcke, den sogenannten UltraRAM, durch den die die Speicherkapazität bis zu einem Faktor von sechs erhöht wird [115]. Insgesamt können bis zu 60 MB

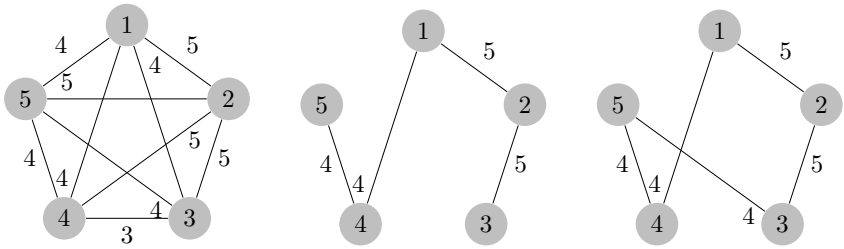
Daten auf der momentan neusten FPGA Familie des Herstellers Xilinx gespeichert werden. Da der Speicher intern verbaut ist, können in jedem Takt Daten gelesen oder geschrieben werden.

2.5 Algorithmen

In diesem Kapitel werden alle in dieser Arbeit verwendeten Algorithmen eingeführt und erläutert. Zunächst werden die Sortierverfahren, welche in Kapitel 5 verwenden werden um Speicherinhalte zu sortieren, beschrieben und Heuristiken für eine effiziente Umsetzung dieser Sortierverfahren für eine große Anzahl von Datensätzen vorgestellt. Des Weiteren werden die in Kapitel 7 verwendeten Algorithmen der Bildverarbeitung detailliert beschrieben, welche die Basis der realisierten Beschleuniger bilden.

2.5.1 *Traveling Salesman* Problem

Das *Traveling Salesman* Problem (TSP), auch Problem des Handlungsreisenden genannt [15], ist eines der meist untersuchten kombinatorischen Optimierungsprobleme und beschäftigt sich mit der Fragestellung, in welcher Reihenfolge eine bestimmte Anzahl von n Städten je einmal besucht werden sollen, um die Reisestrecke insgesamt möglichst kurz zu halten. Dabei soll die Reise bei der Stadt enden, in der sie gestartet ist. Dieses Problem gilt als NP-schwer, liegt in der Klasse NP und ist somit NP-vollständig. Dies bedeutet, dass es keine Garantie gibt, dass eine optimale Lösung innerhalb polynomialer Zeit errechnet werden kann [39]. Das TSP kann mit Hilfe der Graphentheorie modelliert werden, wobei die Städte als Knoten $v \in V$ und die Verbindungen der Städte als Kanten $\{u, v\} \in E$ eines gewichteten Graphen $G = (V, E, c)$ mit $E \subseteq \{\{u, v\} : v, w \in V\}$ und der Distanzfunktion zwischen zwei Städten $c : E \to \mathbb{R}$ repräsentiert werden [44]. Eine Reise zwischen mehreren Städten definiert einen Weg und ist ein nichtleerer Graph $W = (V', E')$ mit $V' = \{v_0, v_1, \ldots, v_k\} \subseteq V$ und $E' = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots \{v_{k-1}, v_k\}\} \subseteq E$, wobei $v_i, i \in \{1, \ldots, k\}$ paarweise verschieden sind. Eine Tour ist eine Reise zwischen mehreren Städten, bei der die Reise wieder an der Stadt endet, bei der sie anfing und wird als Kreis $T = (V'', E'')$ mit $E'' = E' \cup \{v_{k-1}, v_0\}$ moduliert, wobei $W = (V', E')$ die Reise mit $E' = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots \{v_{k-2}, v_{k-1}\}$



(a) Gewichteter Graph $G = (V, E, c)$.

(b) Weg: Graph $W = (V, E')$.

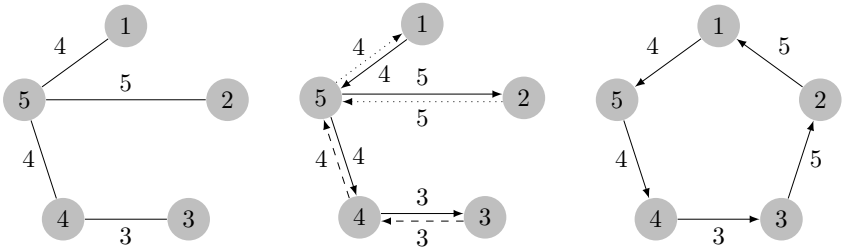
(c) Tour: $T = (V, E'')$.

Abbildung 2.18: TSP moduliert mit Hilfe der Graphentheorie.

und $k \geq 3$ darstellt. Wenn $V = V' = V''$ gilt, dann wird dieser Kreis als Hamilton-Kreis bezeichnet, im Falle des TSP wird ein solcher Hamilton-Kreis gesucht, da alle Städte bereist werden sollen. In Abbildung 2.18 werden die einzelnen Graphen für ein Beispiel mit fünf Knoten illustriert. Das *Traveling Salesman* Problem kann nun wie folgt definiert werden: Für einen gewichteten Graphen $G = (V, E, c)$ mit der Kostenfunktion $c : E \to \mathbb{R}$ wird ein Kreis $C = (V, E')$ mit $E' \subseteq E$ minimaler Länge gesucht, das bedeutet $\sum_{e \in E'} c(e)$ soll möglichst klein sein. In dem Beispiel in Abbildung 2.18c ist eine korrekte Tour dargestellt, die jedoch nicht minimal ist und somit keine optimale Lösung für das *Traveling Salesman* Problem darstellt. In dieser Arbeit soll das *Traveling Salesman* Problem auf einen großen Datensatz angewendet und hierbei eine optimale Lösung gefunden werden. Aufgrund der Größe des Datensatzes werden vorhandene Heuristiken angewendet [110], um eine ausreichend gute Lösung in polynomialer Zeit zu erhalten. In den folgenden Unterkapiteln werden die verwendeten Algorithmen und Heuristiken zur Lösung des *Traveling Salesman* Problems beschrieben.

Algorithmus mit minimalen Spannbäumen

Die Berechnung minimaler Spannbäume (englisch *Mininum Spanning Tree* (MST)) ist ein kombinatorisches Optimierungsproblem [58] und wird häufig als Unterprogramm für die Lösung des *Traveling Salesman* Problems



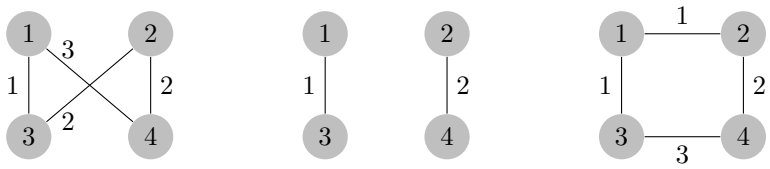
(a) Minimaler Spannbaum $M = (V, E', c)$.

(b) Tour entlang des Spannbaumes.

(c) Minimale Spannbaum-Tour.

Abbildung 2.19: Lösung für das TSP mittels eines minimalen Spannbaumes.

verwendet [84]. Der minimale Spannbaum $M = (V, E')$ ist ein azyklischer Sub-Graph eines Graphen $G = (V, E, c)$, der alle Knoten $v \in V$ mit den minimalen Kosten miteinander verbindet. Der Algorithmus von Kruskal ist ein Greedy-Algorithmus und berechnet für einen gegebenen ungerichteten, zusammenhängenden und gewichteten Graphen $G = (V, E, c)$ den minimalen Spannbaum $M = (V, E', c)$ mit $E' \subseteq E$ [39]. Bei diesem Algorithmus werden die Kanten $e \in E$ nach ihren Kosten sortiert und die Kanten aufsteigend in E' übernommen, solange dadurch kein Zyklus entsteht. In Abbildung 2.19a ist ein minimaler Spannbaum des in Abbildung 2.18a abgebildeten Graphen dargestellt. Der erzeugte minimale Spannbaum dient als Grundlage für die Erstellung einer Lösung für das *Traveling Salesman* Problem und in einem nächsten Schritt wird jede ungerichtete Kante $\{u, v\} \in E'$ durch zwei gerichtete Kanten $\{u, v\}$ und $\{v, u\}$ in E'' mit $u, v \in V$ ersetzt. Dadurch entsteht eine Tour $T_M = (V, E'', c)$ entlang des minimalen Spannbaumes wie in Abbildung 2.19b dargestellt, bei der jeder Knoten $v \in V$ zweimal besucht wird. In einem letzten Schritt wird dies korrigiert, um eine korrekte Lösung für das *Traveling Salesman* Problem zu erhalten. Hierfür werden die Kanten $e \in E''$ des Graphen T_M solange reduziert und durch weitere Kanten $e \in E$ ersetzt, bis der Graph $T_S = (V, E''', c)$ die Dreiecksungleichung $c(u, w) \leq c(u, v) + c(v, w)$ für alle $u, v, w \in V$ erfüllt. Das Ergebnis dieser Prozedur ist dann eine minimale Spannbaum-Tour und ist für das Beispiel in Abbildung 2.19c abgebildet.



(a) Gegebene Tour. **(b)** Entfernung von Kanten. **(c)** Optimierte Lösung.

Abbildung 2.20: Beispiel für die 2-Opt-Heuristik.

Diese Lösung stellt in den meisten Fällen keine optimale Lösung für das *Traveling Salesman* Problem dar.

Algorithmus mit Nächste-Nachbar-Suche

Nächste-Nachbar (englisch *Nearest Neighbor* (NN))-Suche findet in einer Vielzahl von Applikationen Anwendung, einschließlich der Datenkompression [10]. Im Falle des *Traveling Salesman* Problem kann der NN-Algorithmus genutzt werden, um mit wenig Aufwand eine Tour zu finden. Hierfür wird ein beliebiger Knoten $v \in V$ eines ungerichteten, zusammenhängenden und gewichteten Graphen $G = (V, E, c)$ als Startpunkt gewählt und die kostengünstigste Kante $e \in E$ zu einem Graphen $T_N = (V, E'', c)$ hinzugefügt, der den Knoten v mit einem bisher noch nicht verbundenen Knoten w des Graphen T_N verbindet. Dieser Knoten $w \in V$ dient als neuer Startpunkt und sukzessive werden die lokal kostengünstigsten Kanten hinzugefügt, bis der Graph $T_N = (V, E'', c)$ zusammenhängend ist, also alle Knoten miteinander verbunden sind. Die entstandene Tour T_N ist dann eine nicht optimale, aber gültige Lösung für das *Traveling Salesman* Problem.

2.5.2 k-Opt-Heuristik

Wie schon erwähnt, findet sowohl der Algorithmus mittels minimaler Spannbäume, als auch der mittels Nächste-Nachbar-Suche zwar eine gültige, aber keine optimale Lösung für das *Traveling Salesman* Problem. Die generierte Tour $T = (V, E', c)$ für einen Graphen $G = (V, E, c)$ kann

aber mittels Heuristiken weiter optimiert werden und als Startpunkt für diese Optimierung dienen. Die sogenannten k-Opt-Heuristiken bilden eine Klasse von Algorithmen, die eine gegebene Lösung für das *Traveling Salesman* Problem weiter optimieren [76]. Dabei werden k Kanten aus E' entfernt und durch k anderen Kanten aus E ersetzt, wenn dadurch die Gesamtkosten $\sum_{e \in E'} c(e)$ gesenkt werden. Bei $k = 2$ werden zwei Kanten entfernt und kreuzweise wieder eingefügt. In Abbildung 2.20 ist ein Beispiel für die 2-Opt-Heuristik dargestellt, bei der sich die Gesamtkosten durch das Austauschen von je zwei Kanten senken lassen.

2.5.3 Bildverarbeitung im Orts- und Frequenzbereich

Eine Bildverarbeitung kann sowohl im Frequenz- als auch im Ortsbereich des Bildes geschehen [70]. Der Wechsel von einem Bereich in den anderen wird durch die Fouriertransformation vollzogen. Die Fouriertransformation für ein diskretes 2-dimensionales Bild mit der Bildgröße $[0 \leq x < M][0 \leq y < N]$ sieht wie folgt aus [105]:

$$F(u,v) = \sum_{y=0}^{N-1} \left(\sum_{x=0}^{M-1} f(x,y) e^{-2i\pi \frac{ux}{M}} \right) e^{-2i\pi \frac{vy}{N}} \tag{2.6}$$

Die Fouriertransformation stellt die gewichtete Summation aller Bildpunkte dar, wobei das Bild in Sinus- und Kosinusfunktionen zerlegt wird. Je weiter ein Punkt (x, y) im Spektrum vom Bildmittelpunkt entfernt ist, desto höher ist seine darstellende Frequenz u bzw. v, welche die Frequenzvariablen des Bildes darstellen. Das heißt, im Bildinneren werden tiefe Frequenzen verwendet, in den äußeren Bereichen jedoch hohe. Die Funktion $F(u, v)$ ist komplex und lässt sich mit zwei Bildern (Imaginär- und Realteil) darstellen. Die Rücktransformation (inverse Fouriertransformation) lautet:

$$f(x,y) = \frac{1}{NM} \sum_{v=0}^{N-1} \left(\sum_{u=0}^{M-1} F(u,v) e^{-2i\pi \frac{ux}{M}} \right) e^{-2i\pi \frac{vy}{N}} \tag{2.7}$$

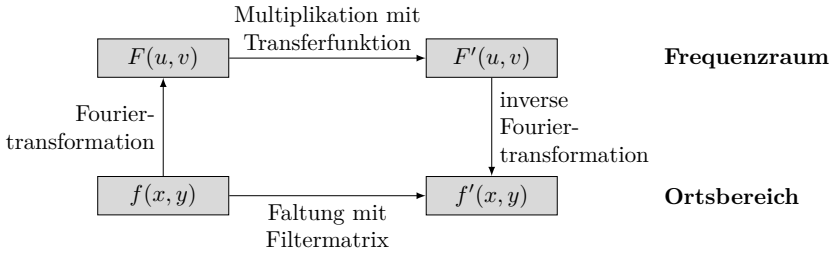


Abbildung 2.21: Bildverarbeitung im Frequenz- und Ortsbereich.

Der Faltungssatz besagt, dass eine Multiplikation im Ortsraum einer Faltung im Frequenzraum und umgekehrt entspricht:

$$f(x,y) * g(x,y) \quad \bullet\!\!-\!\!\circ \quad F(u,v)G(u,v) \tag{2.8}$$

$$f(x,y)g(x,y) \quad \bullet\!\!-\!\!\circ \quad F(u,v) * G(u,v) \tag{2.9}$$

In vielen Fällen wird ein Bild erst in den Frequenzraum überführt, dort bearbeitet und anschließend wieder in den Ortsbereich zurück gerechnet, wie die Abbildung 2.21 veranschaulicht. Besonders bei der Bildanalyse, wie Muster- oder Geschwindigkeitserkennung, Bildfilterung und Bildkompression wird dieses Verfahren angewendet. Dabei wird der Frequenzbereich manipuliert, was häufig eine starke Glättung zur Folge hat, da Filter frequenzselektiv arbeiten. Die Bearbeitung im Ortsbereich ist intuitiver und anschaulicher, meist findet die Manipulation von Grauwerten in diesem Bereich statt. Auch inhomogene Punktoperatoren werden im Ortsbereich angewendet. Dies sind Operatoren, die je nach Lage des Punktes im Bild andere Gewichtsfaktoren bei der Bearbeitung nutzen.

2.5.4 Faltung im Ortsbereich

Die Faltung zweier zweistelliger diskreter Funktionen ist wie folgt definiert [105]:

$$h(x,y) = f[x,y] * g[x,y] = \sum_{a=-\infty}^{\infty} \sum_{b=-\infty}^{\infty} f[a,b]g[x-a, y-b] \tag{2.10}$$

Abbildung 2.22: Mit Gauß-Filter geglättetes Bild (5×5 Matrix).

Wobei f die Filterfunktion bezeichnet, g die Bildfunktion und h das resultierende Bild. Die Bildfilterung ist die Faltung eines Bildes mit einer Filtermatrix. Der neue Bildwert ist die gewichtete Summe der Pixel, die unter der gespiegelten Matrix liegen. Als Gewichte dienen die Matrizenwerte der Filtermatrix und das zu bearbeitende Bild wird bei der Durchführung einer Faltung verkleinert.

Gauß-Filter

Der Gauß-Filter arbeitet in der Bildverarbeitung auf Graustufenbildern. Der Filter dient zur Glättung eines Bildes, die Übergänge werden dabei aufgeweicht und ein mögliches Bildrauschen wird somit verringert, wie die Abbildung 2.22 verdeutlicht.

Definiert wird der Gauß-Filter durch die zweidimensionale Gauß-Funktion [105]:

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.11}$$

Die fouriertransformierte Funktion für den Frequenzbereich ist wie folgt definiert [105]:

$$F(u,v) = e^{-\frac{u^2+v^2}{2}\sigma^2} \tag{2.12}$$

Die Stärke der Glättung wird ausschließlich über den Parameter σ bestimmt, der den Radius der glockenförmigen Funktion repräsentiert. Bei der Erstellung der Filtermatrix wird die Funktion nur approximiert, dadurch wird der Rechenaufwand gesenkt. Je größer die $n \times n$ Filtermatrix gewählt wird, desto höher ist die Güte der Approximation. Im Ortsbereich wird eine

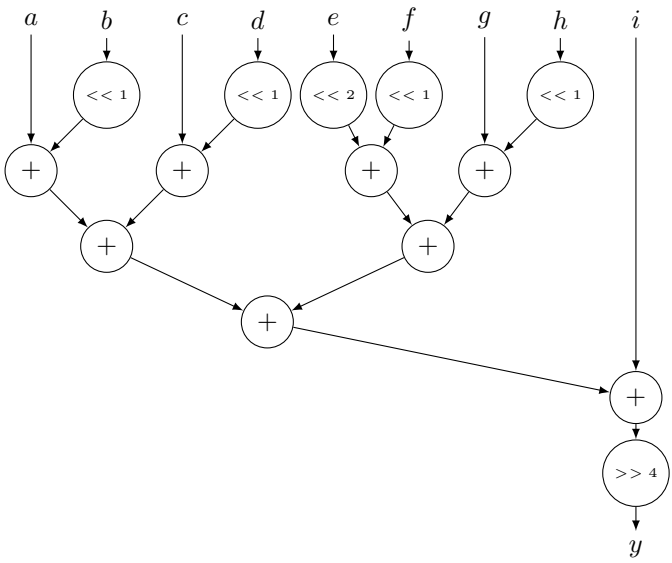


Abbildung 2.23: Datenpfad Gauß Filter.

Größe von $n = \lfloor 2\sigma \rfloor \cdot 2 + 1$ als ausreichend beurteilt. Die Approximation von $f(x)$ durch einen 3×3 Filter für $\sigma = 0,85$ lautet:

$$f_G = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{2.13}$$

In Abbildung 2.23 ist der Datenpfad für eine Faltung mit der Matrix f_G auf ein 3×3 Pixel großes Teilbild mit den Pixelwerten $a - i$ illustriert.

Sobel-Filter

Der Sobel-Filter ist ein richtungsabhängiger Kantendetektions-Filter, der mit Hilfe der ersten Ableitung der Bildpunkt-Helligkeitswerte Kanten ermittelt. Mögliche Kanten befinden sich in den Teilen des Bildes, an denen der Differenzenquotient zweier benachbarter Grauwerte lokal am höchsten

ist. Zusätzlich enthält der Sobel-Filter eine Glättung quer zur Gradientenrichtung, indem er eine diskrete Approximation der Gauß-Verteilung verwendet. Dadurch soll die Entstehung von Artefakten minimiert werden. Die Sobel-Funktion in x-Richtung, die auch vertikale Sobel-Funktion genannt wird, ist wie folgt definiert [105]:

$$S_x = \frac{\partial g(x,y)}{\partial x} \tag{2.14}$$

Wobei g das zu filternde Bild darstellt. Kombiniert mit der Glättung ergibt sich folgende Approximation:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{2.15}$$

Analog gilt dies für die horizontale Sobel-Funktion, in y-Richtung [105]:

$$S_y = \frac{\partial g(x,y)}{\partial y} \tag{2.16}$$

Und deren Approximation:

$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{2.17}$$

Die vertikale Sobel-Funktion S_x liefert eine gute vertikale, jedoch eine schlechte horizontale Kantendetektion. Umgekehrt ist dies bei der horizontalen Sobel-Funktion S_y der Fall. Der Grauwertgradienten-Betrag M liefert durch die Kombination der beiden Sobel-Funktionen eine richtungsunabhängige Information:

$$M = \sqrt{S_x^2 + S_y^2} \tag{2.18}$$

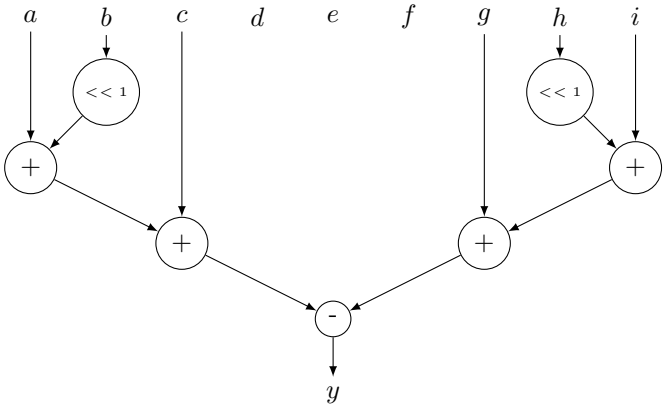


Abbildung 2.24: Datenpfad Sobel y Filter.

Die Richtung φ eines Gradienten erhält man durch:

$$\varphi = \begin{cases} \arctan(\frac{S_y}{S_x}), & \text{falls } g_x \neq 0 \\ 0°, & \text{falls } g_x = 0 \text{ und } g_y = 0 \\ 90°, & \text{falls } g_x = 0 \text{ und } g_y \neq 0 \end{cases} \tag{2.19}$$

$\varphi = 0$ beschreibt eine vertikale Kante, positive Werte beschreiben eine Drehung gegen den Uhrzeigersinn. In Abbildung 2.24 ist der Datenpfad für eine Bearbeitung eines Pixels mittels der S_y Matrix auf ein 3×3 Pixel großes Teilbild mit den Pixelwerten $a - i$ dargestellt. Für eine Faltung mit der S_x Matrix werden nur die Reihenfolge der Eingabewerte variiert, der Ablauf bleibt jedoch identisch.

Laplace-Filter

Ebenso wie die Sobel-Operatoren wird der Laplace-Operator in der Bildverarbeitung zur Kantendetektion eingesetzt. Hierfür wird die zweite

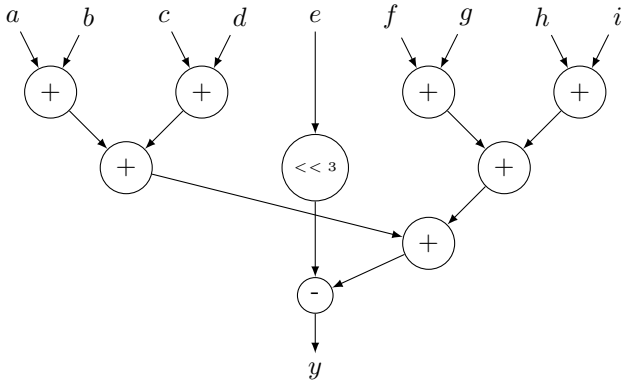


Abbildung 2.25: Datenpfad Laplace Filter.

Ableitung des Signals verwendet und eine Kante wird als Nulldurchgang erkannt. Der Laplace-Operator ist wie folgt definiert [105]:

$$L(x,y) = \frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2} \tag{2.20}$$

und die Faltungsmatrix erhält man durch die Diskretisierung der Differenzenquotienten:

$$\Delta^2_{xy} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \tag{2.21}$$

Diese Filtermatrix erkennt sowohl vertikale, horizontale als auch Kanten im 45° Winkel. In Abbildung 2.25 ist der Datenpfad für eine Bearbeitung eines Pixels mittels der Δ^2_{xy} Matrix auf ein 3×3 Pixel großes Teilbild mit den Pixelwerten $a - i$ dargestellt.

3 Stand der Technik

In diesem Kapitel wird der aktuelle Entwicklungsstand der für diese Arbeit relevanten Forschungsthemen erläutert. Zu Beginn des Kapitels wird auf den gegenwärtigen Aufbau des CMS-Triggersystems und des Siliziumdetektors eingegangen. Darauf aufbauend wird die Mustererkennung mittels Assoziativspeicher eingeführt und die bestehende dedizierte Hardware beschrieben. Die Gegebenheiten des Detektors und die Verarbeitung durch die Hardware stellen die Basis für die in dieser Arbeit entwickelte Speicherarchitektur dar, die in den folgenden Kapiteln ausführlich dargelegt wird. Abschließend wird die in dieser Arbeit verwendete invasive Mikroarchitektur und der invasive Prozessor *i*-Core beschrieben, auf der die Entwicklung des Konzeptes einer transparenten und dynamischen Hardwarebeschleunigung basiert, welches dann in Kapitel 7 spezifiziert wird.

3.1 Der CMS-Detektor

In diesem Unterkapitel werden die für diese Arbeit relevanten Subdetektoren des CMS-Detektors und ihre geplanten Erweiterungen für den *High Luminosity-Large Hadron Collider* beschrieben.

3.1.1 Silizium-Streifendetektor

Der Silizium-Streifendetektor des CMS-Detektors besteht aus einzelnen Modulen, die wiederum aus Silizium-Sensoren und einer Auslese-Einheit bestehen. Es existieren zwei verschiedene Module: Die *2S*-Module (*Strip-Strip sensors*) und die *PS*-Module (*Pixel & Strip sensors*) [1]. Diese Sensoren sind in Doppellagen angeordnet und verwerfen durch diese Anordnung direkt Daten, die von Teilchen mit niedrigem Impuls erzeugt werden. In

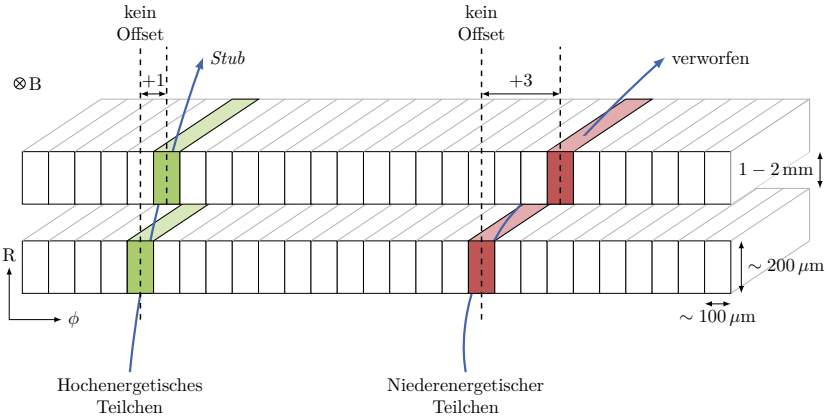


Abbildung 3.1: Konzept der Trigger-Module: Hochenergetische Teilchen aktivieren die zwei Schichten des Sensors mit einem geringen Offset zueinander (in grün gekennzeichnet).

Abbildung 3.1 wird das Konzept dieser Vorfilterung anhand eines *2S*-Moduls skizziert [62]. Die Flugbahn von hochenergetischen Teilchen wird durch das Magnetfeld weniger stark gekrümmt als von niederenergetischen Teilchen [81]. Von der unteren Sensorlage ausgehend muss das Teilchen, welches beide Sensorlagen passiert, in einem bestimmten Bereich in der oberen Sensorlage gemessen werden, damit die Sensordaten weiter verarbeitet werden. In Abbildung 3.1 sind zwei Teilchenbahnen und deren Interaktionspunkte mit den Sensormodulen eingezeichnet. Das niederenergetische Teilchen (rechts) durchläuft die äußere Sensorlage mit einer Abweichung von drei Sensorabschnitten relativ zur inneren Sensorlage. Durch diesen Offset wird Transversalimpuls p_T des Teilchen bestimmt und in diesem Fall die Daten nicht weiter verarbeitet, da von Teilchen mit geringen Transversalimpuls p_T keine neuen Erkenntnisse erwartet werden. Das hochenergetische Teilchen (links) in Abbildung 3.1 passiert beide Sensorlagen mit einem Offset von nur einem Sensorabschnitt und die Daten werden durch den CMS *Binary Chip* (CBC) ausgelesen und weiter geleitet [60]. Die Sensordaten der beiden Lagen werden dann zu einem sogenannten *Stub* zusammengefasst und der Offset der zwei Sensorabschnitte wird als *Bend* bezeichnet. Rund 99% aller Teilchenbahnen werden durch Teilchen mit

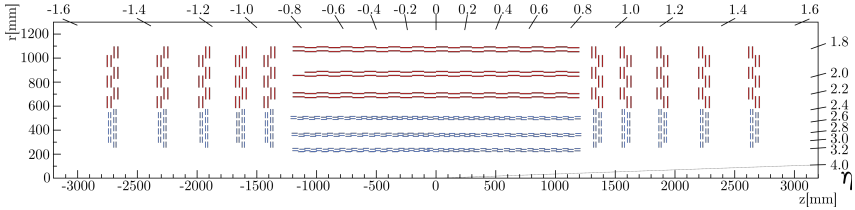


Abbildung 3.2: Sensorlagen Anordnung des äußeren Trackers des CMS-Silizium-Streifendetektors.

einem Transversalimpuls p_T von weniger als 2 GeV erzeugt und sind somit nicht relevant für die weitere Datenverarbeitung [87]. Diese Filterung anhand von zwei Sensorschichten reduziert die Datenmenge um einen Faktor von bis zu 50 [89]. Der Streifendetektor besteht aus mehr als 1400 dieser Sensoren, welche transversal um die Strahlachse herum verteilt liegen. In der Mitte des Detektors liegt der Interaktionspunkt der gegenläufig beschleunigten Strahlenbündel, somit würde eine kugelförmige Anordnung eine optimale Abdeckung bieten, der Aufbau von Sensoren, die eine solche Anordnung ermöglichen, ist jedoch sehr komplex. Aus diesem Grund wurde eine zylindrische Anordnung der Sensoren für den aktuellen Streifendetektor verwendet, welche in Abbildung 3.2 dargestellt ist. Im mittleren Bereich des Detektors bilden die Module eine gebogene Oberfläche, wie in Abbildung 3.3 skizziert ist und dieser Abschnitt des Detektors wird als Zylinderzone (englisch *Barrel Section*) bezeichnet. Der Messbereich wird durch die sogenannten Endkappen an beiden Seiten des Zylinders erweitert. Die Module der Endkappen stehen orthogonal zur Strahlachse und sollen so eine möglichst gute kugelförmige Abdeckung um den Interaktionspunkt ermöglichen. Die *PS*-Module liegen näher an der Strahlachse (bis 60 cm Radius) und liefern eine höhere Auflösung als die *2S*-Module, welche dafür eine größere Abdeckung bieten [109]. Die aktive Fläche der *PS*-Module liegt bei rund 44 cm und besteht aus 1 920 Streifensensoren mit einer Länge von 2,5 cm und 30 720 Pixeln [25] mit einer Breite von 1,446 mm. Die *2S*-Module verfügen über fast die doppelte aktive Fläche und bestehen aus zwei Schichten mit jeweils 1 016 Streifensensoren, welche 5 cm lang sind. Für den geplanten Ausbau des Detektors für den *High Luminosity-Large Hadron Collider* sollen die *PS*-Sensormodule teilweise gekippt werden, damit die Teilchenbahnen senkrecht zu den Modulen verlaufen [17]. Durch

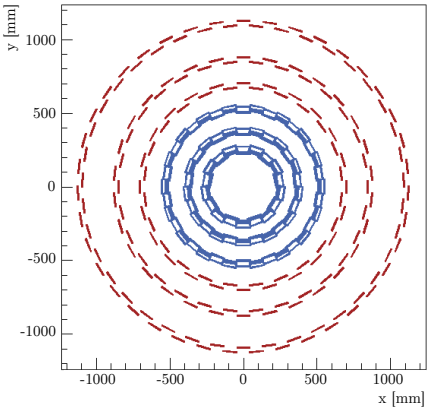


Abbildung 3.3: Sensorlagen Anordnung des äußeren Trackers des CMS-Detektors mit Sicht in Richtung Strahlachse.

Modulart	Anzahl	Aktive Fläche (in m^2)	Kanäle (in Millionen)
Strip-Strip	8 424	154,8	34,2
Pixel & Strip	5 748	51	187,6
gesamt	14 172	205,8	221,8

Tabelle 3.1: Anzahl der Sensormodule des äußeren Trackers für das gekippte Layout [4, 98].

diese gekippte Anordnung werden in diesen Lagen weniger Sensormodule benötigt, was zu einer Ersparnis von Material führt, mit welchem die durchfliegenden Teilchen interagieren könnten. Auf der anderen Seite steigt dadurch jedoch die Komplexität der Montage der einzelnen Module und deren Verkabelung. Insgesamt werden über 200 m^2 aktive Fläche Silizium verteilt auf über 14 000 Sensormodule verbaut. In Tabelle 3.1 ist die Anzahl der Module für den geplanten Aufbau mit gekippten Modulen aufgeschlüsselt sowie die Anzahl der benötigten Auslesekanäle.

3.1.2 Das Level 1 Triggersystem

Für den *High Luminosity-Large Hadron Collider* muss das Triggersystem des CMS-Detektors überarbeitet werden, um die neuen Herausforderungen von bis zu 200 Kollisionen pro Event stemmen zu können. Die zu erwartenden Datenraten nach der Vorfilterung durch die Sensormodule von bis zu 20 Tbit/s [87] bei einer Ereignisrate von 40 MHz können nicht vollständig abgespeichert werden. Eine Reduktion der Daten um einen Faktor von mindestens 50 wird angestrebt, die Ereignisrate soll auf der ersten Triggerstufe auf 750 kHz reduziert werden [36]. Die Daten der Kalorimeter und des Myonenspektrometers alleine sind nicht ausreichend, um eine solche Datenreduzierung durchzuführen, vor allem die Impulsauflösung des Myonenspektrometers ist hierzu nicht hoch genug. Aus diesem Grunde werden erstmals die Daten des Silizium-Streifendetektors in die Entscheidung der ersten Triggerstufe durch einen sogenannten *Level 1 Track Trigger* mit eingebunden. In Abbildung 3.4 wird das geplante Level 1 Trigger System schematisch mit den angestrebten Latenzen, der Ereignisraten und der L1 Akzeptanzrate dargestellt. Nach einer Latenz von 12,5 μs soll das Ergebnis der Akzeptanzprüfung des Level 1 Triggers abgeschlossen sein [38]. Für die Bereitstellung der Spurinformationen durch den *Level 1 Track Trigger* werden 5 μs Latenz eingeplant, wobei die Vorfilterung durch die Module und die Übertragung der Triggerdaten von den Sensoren hin zu der *Level 1 Track Trigger*-Einheit inkludiert ist. Weitere 2,5 μs fallen für eine weitere Verarbeitung an: Die Spurinformationen werden aufbereitet, um den primären Vertex zu bestimmen und diese werden mit den Objekten, die vom Kalorimeter-Trigger berechnet wurden, abgeglichen. In einem nächsten Schritt werden diese aufbereiteten Spurinformationen mit den Spurdaten des Myonenspektrometers vereint. Diese Kombination ermöglicht Berechnungen von spurbezogenen L1 Objekten und deren Charakteristika. Diese Spuren werden verwendet, um die erkannten Myonen von den Kalorimeter Objekten abzugrenzen. All diese Berechnungen sollen 7,5 μs nach der Kollision der Teilchen abgeschlossen sein, danach werden noch für weitere Berechnungen und die endgültige Entscheidung der Triggerstufe 1 μs eingeplant. Abschließend wird für die Übertragung des Trigger-Signals zum Frontend eine weitere Mikrosekunde benötigt, so dass das Signal 9,5 μs nach der Kollision zur Verfügung steht. Diese Latenz wird um einen Sicherheitsfaktor von 30% erhöht und ergibt dann die Gesamtlatenz von

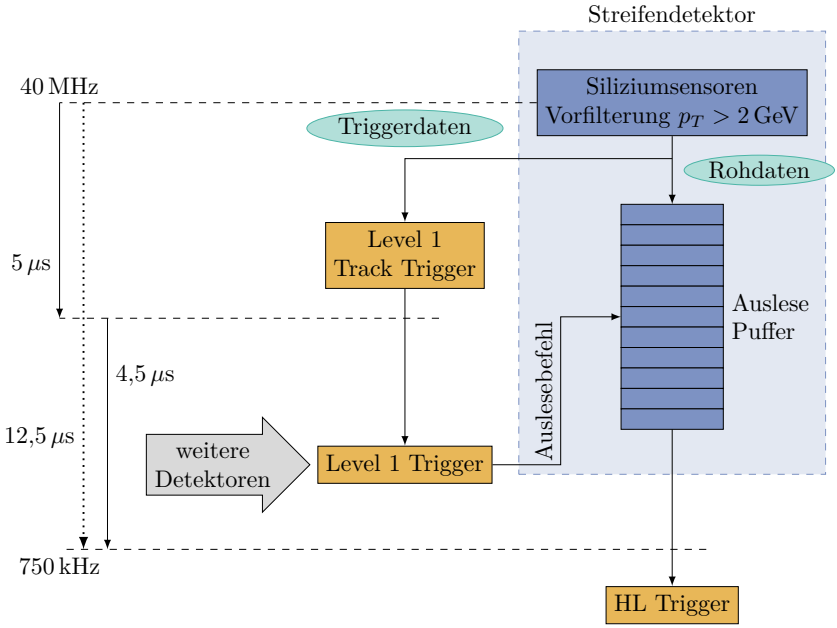


Abbildung 3.4: Aufbau des CMS Level 1 Triggersystems mit geplanten Latenzen der einzelnen Komponenten (durchgezogen) und des Gesamtsystems (gepunktet) mit einer inkludierten Pufferzeit.

12,5 μs, die nicht überschritten werden darf, da ansonsten die Rohdaten der Sensoren verworfen werden.

3.1.3 L1 Track Trigger

Wie in Unterabschnitt 3.1.2 beschrieben, liefert der *Level 1 Track Trigger* erste Spurinformationen zu erkannten Teilchenbahnen auf Basis der vorgefilterten Sensordaten des Silizium-Streifendetektors. Der *Level 1 Track Trigger* ist dem entsprechend kein klassischer Trigger, da er die Ereignisrate selbst nicht reduziert und ist Teil des Level 1 Triggers. Aufgrund der geringen Latenzzeit von 5 μs und der hohen Menge zu verarbeitender Daten, muss für die Umsetzung des *Level 1 Track Trigger* dedizierte Hardware entwickeln werden. Hierfür wird ein System mit bis zu 2 000 FPGAs veranschlagt und es existieren mehrere Konzepte, wie die Anforderungen an den *Level 1 Track Trigger* erfüllt werden können [87, 1]. Im Wesentlichen haben sich zu Beginn der vorliegenden Forschungsarbeit (Anfang 2013) drei Konzepte und deren Arbeitsgruppen für eine mögliche Realisierung positioniert:

Mustererkennung mit Assoziativspeicher Für dieses Konzept wird der Detektor, ausgehend vom Kollisionspunkt, in 48 ($8(\phi) \times 6(\eta)$) Sektoren geteilt. Die *Stubs* werden für jeden Sektor unabhängig voneinander mit vorausberechneten Spurbahnen abgeglichen. Dieser Vergleich wird massiv parallel mit Hilfe eines dedizierten *Associative Memory* Chips (AM-Chips) durchgeführt. Diese vorberechneten Spurbahnen sind grobgranularer als die Detektorauflösung und reduzieren die Daten einer Spurbahn auf sechs Ortspunkte und sind in einem Assoziativspeicher geladen. Innerhalb weniger Taktzyklen kann somit entschieden werden, ob eine Kombination von *Stubs* eine vorberechnete Spurbahn bilden, diese *Stubs* werden dann in einem weiteren Schritt für die Spurrekonstruktion genutzt. Durch die Reduzierung auf *Stubs*, die in relevanten Spurbahnen enthalten sind, kann die Spurrekonstruktion auf ein kombinatorisch weniger komplexes Problem transferiert und auf FPGAs berechnet werden. Im Schnitt müssen bis zu 300 *Stubs* pro Sektor mit rund ein bis zwei Millionen Spurbahnen abgeglichen werden [87].

Hough-Transformation Die Hough Transformation ist ein bekanntes Verfahren zur Erkennung von beliebigen parametrisierbaren geometrischen Objekten, wie beispielsweise Kreise und Geraden [45]. In diesem Ansatz wird der Algorithmus verwendet, um *Stubs* grob nach ihrem Transversalimplus zu gruppieren und anschließend einer Spurbahn zuzuordnen. Diese Eingruppierung findet jedoch nur in der (r, ϕ)-Ebene statt, und so können auch *Stubs* miteinander gruppiert werden, die nicht auf einer Spurbahn in der (r, z)-Ebene liegen. In einem zweiten Schritt muss dies dann korrigiert und die irrtümlich gruppierten *Stubs* wieder voneinander getrennt werden. Auch werden Spurbahnen, deren Ursprung zu weit vom Kollisionspunkt entfernt sind, verworfen. Spurbahnen, denen eine genügend hohe Anzahl von *Stubs* zugeordnet worden sind, werden dann für die weitere Verarbeitung ausgegeben. Um sowohl die Systemarchitektur als auch die Implementierung der Spurfindung durch die Hough Transform Spurfinders zu vereinfachen, wird ein vollständig zeitgemultiplextes FPGA-Design entwickelt [61].

FPGA basierter Tracklet Ansatz Dieser Ansatz verwendet sogenannte *Tracklets*, Fragmente von Teilchenbahnen, welche durch zwei benachbarte *Stubs* gebildet werden [28]. Ausgehend von diesem *Tracklets* werden dann *Stubs* in den weiteren Lagen sukzessive hinzugefügt, um anschließend zu einer Spur zusammengefasst zu werden. Diese Vorgehensweise einer Spurfindung ist in Software bereits erprobt, muss aber für den *Level 1 Track Trigger* auf FPGAs übertragen werden [87]. Der Detektor wird in der (r, ϕ)-Ebene in 28 Sektoren unterteilt und die Spurfindung wird dann lokal für einen Sektor auf einem FPGA berechnet. Hierfür überlappen die Sektoren leicht, dennoch kann es vorkommen, dass eine Spur über zwei benachbarte Sektoren verläuft und Daten für die Berechnung ausgetauscht werden müssen. Die größte Herausforderung bei diesem Konzept ist aber die große Anzahl von *Stub*-Paaren, die als Ursprung einer zu errechnenden Spurbahn existieren.

Das KIT ist unter anderem in der Arbeitsgruppe für die Realisierung mit Hilfe der Assoziativspeicher vertreten und beteiligt sich zudem in der Evaluierung des Gesamtkonzeptes [4]. Das Konzept der Mustererkennung mit Assoziativspeichern wird auch weiterhin von der ATLAS-Gruppe verfolgt und in den Detektor integriert. Auch in der vorliegenden Arbeit

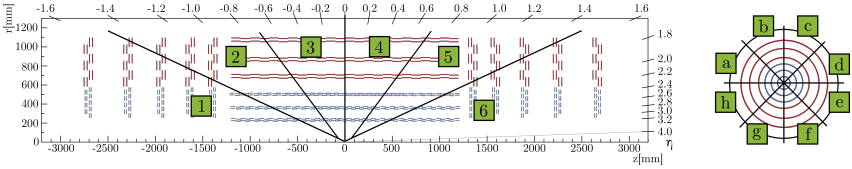


Abbildung 3.5: Einteilung des CMS-Detektors in 48 *Trigger Tower*.

wird dieser Ansatz verwendet und angestrebt die Eigenschaften des AM-Chips in einem FPGA zu realisieren.

3.2 Mustererkennung mit Assoziativspeicher

In diesem Unterkapitel wird auf die spezifischen Anforderungen des CMS-*Level 1 Track Triggers* und die Gegebenheiten des CMS-Detektors eingegangen. Des weiteren wird die dedizierte Hardware vorgestellt, welche sich für eine Realisierung mittels Mustererkennung mit Assoziativspeicher anbietet und in der Vergangenheit genutzt wurde.

3.2.1 Geometrische Einteilung und generierte Spurbahnen

Wie in Unterabschnitt 3.1.2 beschrieben, wird der Detektor in 48 Sektoren unterteilt, diese Sektoren werden *Trigger Tower* genannt. Abbildung 3.5 illustriert diese *Trigger Tower*, welche sowohl durch Schnitte in der (z, η)-Ebene als auch in der (r, ϕ)-Ebene gebildet werden. Die Sektoren eins und sechs in der (z, η)-Ebene sind achsensymmetrisch zur Geraden $z = 0$, ebenso wie die Sektoren zwei und fünf beziehungsweise drei und vier. Die Sektoren a bis f in der (r, ϕ)-Ebene sind punktsymmetrisch um die z-Achse. Auf diese Weise entstehen drei verschiedene Ausführungen von Sektoren, welche im folgenden in der (z, η)-Ebene erläutert werden:

Barrel Die Sektoren drei und vier bilden die sogenannte *Barrel*-Sektion und beinhalten sechs Sensorlagen, drei mit *PS*-Modulen und drei mit *2S*-Modulen.

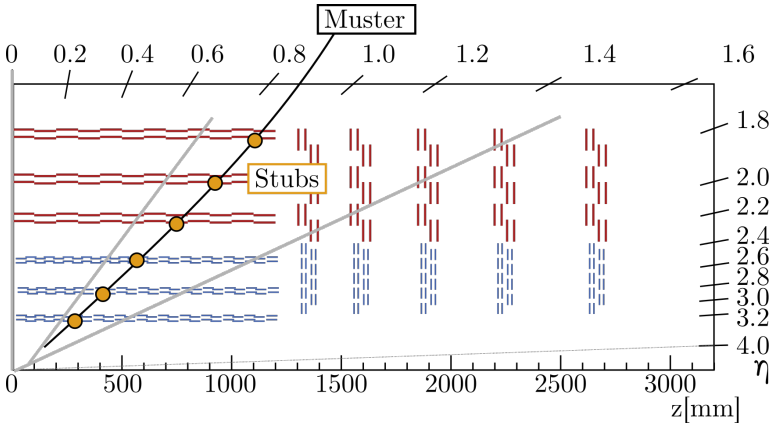


Abbildung 3.6: Ein Muster der *Pattern Bank* bestehend aus sechs Stubs (in orange gekennzeichnet) innerhalb eines hybriden Sektors.

Endcap Die Sektoren eins und sechs decken sowohl einen Teil der inneren *PS*-Module, als auch einen Großteil der Sensormodule der äußeren Endkappen ab.

Hybrid Zwischen den *Barrel*- und *Endcap*-Sektoren befinden sich die *Hybrid*-Sektoren zwei und fünf. Dieser Sektor beinhaltet *2S*-Module der Endkappen und Teile der sechs Sensorlagen des Barrels.

Die Sektorengrenzen sind so definiert, dass sie sich überlappen, die Daten der äußeren Sensorlagen werden in beiden benachbarten Sektoren ausgewertet. Dies ermöglicht es, eine Spurbahn, die über zwei benachbarte Sektoren verläuft, zu erkennen. Somit kann die Mustererkennung pro Sektor unabhängig von den anderen Sektoren durchgeführt und alle Sektoren parallel verarbeitet werden. Für jede der drei Sektortypen existiert eine sogenannte *Pattern Bank*, eine generierte Ansammlung von interessanten Spurbahnen. Eine Spurbahn wird durch die sechs Modulnummern beschrieben, welche von einem Teilchen auf dieser Bahn passiert werden, und wird als Muster bezeichnet. In Abbildung 3.6 ist eine Teilchenspur durch einen hybriden Sektor eingezeichnet, die *Stubs* markieren hier die durchlaufenden Sensormodule, deren Nummern das Muster bilden. Auch wenn die Spurbahn mehr als sechs Sensorlagen durchläuft, besteht das dazu gehörende Muster

nur aus sechs Modulnummern, welche im Voraus definiert wurden. Jede Modulnummer besteht aus 16 bit und ein Muster somit insgesamt aus 96 bit, die Sensorlagen sind geometrisch von der Strahlachse ausgehend aufsteigend sortiert. Die Modulnummer des ersten durchlaufenden Sensormoduls wird dementsprechend in dem Muster vorangestellt, gefolgt von der zweiten bis hin zur Modulnummer des sechsten Sensormoduls.

3.2.2 Mustererkennung

Algorithmen mit Mustererkennungen sind weit verbreitet, vor allen in Bereichen, bei denen eine große Datenmenge nach spezifischen Datensätzen durchsucht werden soll. Insbesondere in der Hochenergiephysik werden solche Algorithmen erfolgreich zur Bestimmung von Spurbahnen und deren Vertices, den ursprünglichen Kollisionspunkten der Teilchen, verwendet [59]. Die Mustererkennung in diesem Anwendungsbereich beruht auf der Annahme, dass die Charakteristiken der Spurbahnen der zu untersuchenden Teilchen bekannt sind und diese Bahnen durch Simulationen vorberechnet werden können. Es wird eine Datenbank erstellt, die möglichst alle relevanten Spurbahnen enthält, und die Eingangsdaten werden dann nach genau diesen Datensätzen durchsucht. Für das in Unterabschnitt 3.1.3 beschriebene Konzept des *Level 1 Track Triggers* mit Mustererkennung mittels Assoziativspeicher soll dieser Ansatz verfolgt werden. In Abbildung 3.7 wird diese Vorgehensweise illustriert. Der Detektor ist hier aus Sicht der Strahlachse dargestellt und besitzt der Übersicht halber nur vier der sechs Sensorlagen. Die Eingangsdaten, in diesem Falle die orange gekennzeichneten Stubs, werden mit den Mustern der Datenbank (genannt *Pattern Bank*) abgeglichen. Die grün dargestellten Muster Nummer drei und fünf sind in den Eingangsdaten vorhanden und in diesem Beispiel würden die acht *Stubs*, die diese Muster bilden, an die Spurrekonstruktion weitergeleitet werden. Die restlichen *Stubs* werden verworfen, auch wenn sie in Mustern vorkommen, die teilweise von den Eingangsdaten abgedeckt werden. In dem illustrierten Beispiel sind zwei *Stubs* des rot gekennzeichneten Musters Nummer 1 enthalten, dies reicht jedoch nicht aus um dieses Muster zu triggern und eine Speicherung der *Stubs* zu initiieren. Die verworfenen *Stubs* beeinflussen die weitere Verarbeitung durch den *Level 1 Track Trigger* nicht. Pro *Trigger Tower* existieren ungefähr eine Millionen solcher Muster, welche innerhalb kürzester Zeit mit den Triggerdaten verglichen

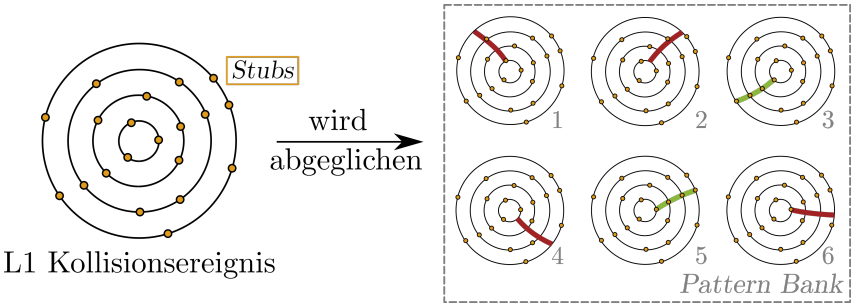


Abbildung 3.7: Die Triggerdaten werden mit berechneten Mustern verglichen.

werden müssen. Aufgrund der kurzen Latenzzeit des CMS *Level 1 Track Triggers* kann eine solche Mustererkennung nicht in Software durchgeführt werden, sondern muss in Hardware realisiert werden. Eine Realisierung mit dedizierten Chips, welche auch für das ATLAS-Experiment genutzt werden, wird im folgenden Unterabschnitt näher erläutert.

3.2.3 *Associative Memory* Chip

Der Entwurf der AM-Chips basiert auf der Aneinanderreihung von CAM-Zellen, welche in Unterabschnitt 2.2.2 vorgestellt wurden. In Abbildung 3.8 ist der schematische Aufbau des Chips skizziert, für jeden *Stub* eines Musters existiert eine CAM-Zelle, die mit einem Flip-Flop verbunden ist. Die Aneinanderreihung mehrerer dieser Speicherzellen mit nach gesetztem Flip-Flop beinhalten ein Muster. Dieses Muster sollen mit den Eingangsdaten abgeglichen werden. Alle *Stubs* einer Lage werden parallel über einen Bus eingelesen und mit dem Speicherinhalt der CAM-Zelle verglichen, das Ergebnis dieser Überprüfung wird in dem Flip-Flop gespeichert [5]. Falls ein Muster in mindestens fünf der sechs Lagen eine positive Überprüfung erzeugt, wird das Muster als erkannt markiert. Diese Schwellwert-Entscheidung wird durch eine nachgeschaltete Matrix realisiert und ermöglicht eine Erkennung der Muster, auch wenn einzelne Sensoren ausgefallen sind. Die Nummer des erkannten Musters wird dann durch den AM-Chip für die weitere Verarbeitung ausgegeben. Aufgrund

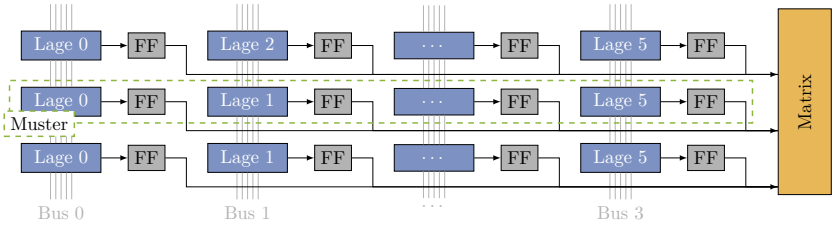


Abbildung 3.8: Schematischer Aufbau AM-Chip.

dieser Schwellwert-Entscheidung werden mehrere Muster für eine einzelne Spurbahn erkannt, und die nachfolgende Spurrekonstruktion muss diese Vervielfältigung der Daten handhaben können. Zudem müssen in einem weiteren Schritt die *Stubs* anhand der Musternummer wieder hergestellt werden. Dies erfordert sowohl einen zusätzlichen Speicher als auch weitere Rechenzeit. Die fünfte Generation des AM-Chips kann bis zu 80 000 Muster speichern [6] und die nächste Version des Chips soll eine Kapazität von bis zu 128 000 Mustern haben [7]. Somit werden pro *Trigger Tower* bis zu 16, beziehungsweise 25 AM-Chips (je nach Generation) benötigt, um alle Muster abzudecken.

3.3 Invasive Mikroarchitektur

Der Sonderforschungsbereich *Invasive Computing* (InvasIC)[1] ist ein transregionales Verbundprojekt der drei Universitäten *Friedrich-Alexander-Universität Erlangen-Nürnberg* (FAU), *Technische Universität München* (TUM) und *Karlsruher Institut für Technologie* (KIT) und wird von der *Deutschen Fördergemeinschaft* (DFG) mittlerweile in der dritten Phase (2018-2022) finanziert. In diesem Projekt wird ein neues Paradigma für das Design und die ressourcenschonende Programmierung zukünftiger paralleler Computersysteme untersucht. Besonders bei Systemen mit 1000 und mehr Kernen auf einem Chip ist eine ressourcenbewusste Programmierung von elementarer Bedeutung, um eine hohe Auslastung mit zeitgleicher hoher Energieeffizienz und Rechenleistung zu erhalten. Hierfür wurde das

[1] http://www.invasic.de

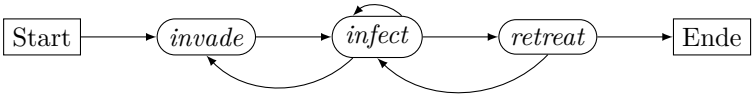


Abbildung 3.9: Zustandsdiagramm des Paradigma *invasive Computing* [102].

Paradigma *invasive Computing* eingeführt, welches dem Programmierer die Möglichkeit gibt, die Ressourcenanforderungen der Anwendung für die verschiedenen Phasen der Ausführung explizit zu definieren [103, 102]. In Abbildung 3.9 sind diese verschiedenen Phasen, welche bei der Ausführung eines invasiven Programmes auftreten, abgebildet. Zu Beginn werden die wird die Anzahl und Art der benötigten Ressourcen für die Ausführung einer Anwendung angefordert (*invade*) und in einem weiteren Schritt wird diese Anwendung auf die beanspruchten Ressourcen geladen (*infect*). Sobald die Anwendung auf allen zugewiesenen Ressourcen ausgeführt wurde, können die Ressourcen entweder wieder freigegeben werden (*retreat*) oder weitere Ressourcen eingefordert werden (*invade*). Zudem ist es auch möglich, dass eine weitere Anwendung direkt diese Ressourcen übernimmt (*infect*). Sobald alle Ressourcen wieder freigegeben worden sind und es keine weiteren Anfragen mehr gibt, terminiert der Programmablauf. In Abbildung 3.10 ist eine Übersicht der verfügbaren Ressourcen der entwickelten invasiven Architektur gegeben, welche eine heterogene Multi-Core-Architektur darstellt. Über ein *Network on Chip* (NoC) sind verschiedene Kacheln (genannt *Tiles*) miteinander verbunden, wobei jede *Tile* über unterschiedliche Ressourcen verfügen kann. Die Standard Rechen-Kachel verfügt über vier Prozessoren (engl. *Central Processing Unit*s (CPUs)) und einem lokalem Speicher (*Tile-local Memory* (TLM)). Neben Kacheln mit Speichereinheiten und Ein-/Ausgabe-Schnittstellen (englisch *Input/Output* (I/O)), gibt es zudem massiv parallele *Tightly-coupled Processor Array*s (TCPAs), die ähnlich einem *General-Purpose* Prozessor (GPP) aufgebaut sind [71]. Des Weiteren gibt es eine Rechen-Kachel mit drei CPUs, lokalem Speicher und einem *Application-Specific Instruction-Set* Prozessor (ASIP), dem sogenannten *i*-Core [66]. Der *i*-Core ist eine adaptive applikations-spezifische Mikroarchitektur, die über ein rekonfigurierbares FPGA-Gefüge verfügt und von dem Teilbereich B des Sonderforschungsbereiches InvasIC entwickelt wird. Dieser Teilbereich

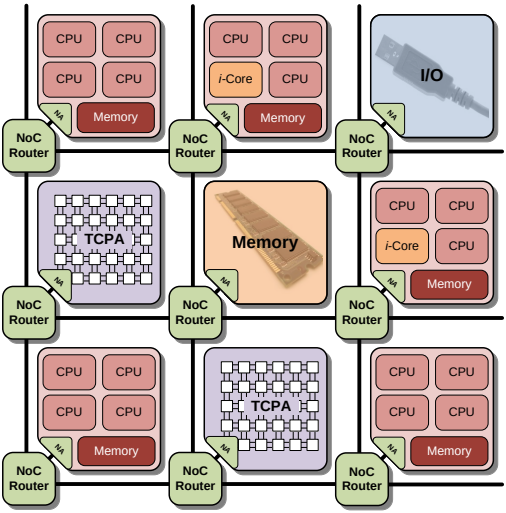


Abbildung 3.10: *Invasive Computing* Architektur mit verschiedenen Kacheln [65].

beschäftigt sich im Allgemeinen mit invasiven Mehrkern Architekturen [67] und untersucht Mechanismen, die eine Laufzeitadaptivität der Mikroarchitekturen unterstützen. Hierbei wurden sowohl zur Laufzeit adaptive L1-Cache-Konfigurationen untersucht [107], als auch Spezialinstruktionen, die bei Bedarf zur Laufzeit bereitgestellt werden [66].

3.3.1 Der invasive Prozessor *i*-Core

Der *i*-Core ist ein adaptiver Prozessor, der die *Rotating Instruction Set Processing* Plattform (RISPP) zu Grunde liegt [14]. In Abbildung 3.11 ist der schematische Aufbau des *i*-Cores mit Anbindung an die *Invasive Computing* Architektur skizziert. Als Basis dient die Implementierung des LEON 3 von Gaisler (siehe Unterabschnitt 2.3.3). Die übernommenen Komponenten sind in Abbildung 3.11 grau markiert. Der *i*-Core verfügt über eine adaptive Sprungvorhersage, einen adaptiven Cache [107], sowie über eine rekonfigurierbare Hardware-Einheit. Die siebenstufige Pipeline

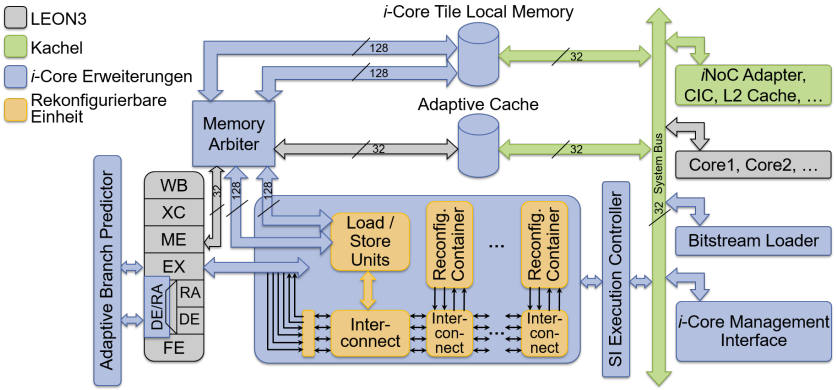


Abbildung 3.11: Adaptive Mikroarchitektur des *i*-Cores mit rekonfigurierbarer Einheit [108].

kann über sogenannte Spezialinstruktionen (SIs) auf die in dieser rekonfigurierbaren Hardware-Einheit geladenen Beschleuniger zugreifen. Diese Spezialinstruktionen sind modular aufgebaut und zeichnen sich dadurch aus, dass sie mehrere Implementierungen (genannt *Moleküle*) besitzen können. Diese Moleküle bestehen wiederum aus mehreren *Atomen*, die die kleinste rekonfigurierbare Einheit darstellen und den eigentlichen Datenpfad beinhalten. Abbildung 3.12 stellt den Aufbau mehrerer modulare Spezialinstruktionen (A bis C) dar, bei der je SI verschiedene *Moleküle* zur Verfügung stehen, welche sowohl Software-Lösungen (A_{SW}, B_{SW} und C_{SW}) als auch Hardware-Implementierungen (A_1 bis C_2) darstellen können. Eine solche Hardware-Implementierung stellt eine Kombination aus mehreren *Atoms* dar, wobei auch mehrere Instanzen eines *Atoms* verwendet werden können. Die einzelnen *Moleküle* erfüllen somit verschiedene Anforderungen, die an die Ausführung der Spezialinstruktion gestellt werden, und je nach Bedarf kann zur Laufzeit der Fokus auf maximale Parallelisierung (ein *Molekül* bestehend aus vielen *Atom*-Instanzen) oder minimalen Rekonfigurierungs-Aufwand (Software-Lösung oder minimale Anzahl von *Atoms*) gelegt werden. Ein *Molekül* einer Spezialinstruktion steht zur Verfügung, wenn alle nötigen *Atoms* dieses *Moleküls* in der rekonfigurierbaren Einheit des *i*-Cores geladen sind. Ein *Atom* an sich ist unabhängig von den *Molekülen*, in welchen es vorkommt, und kann somit auch in verschiedenen

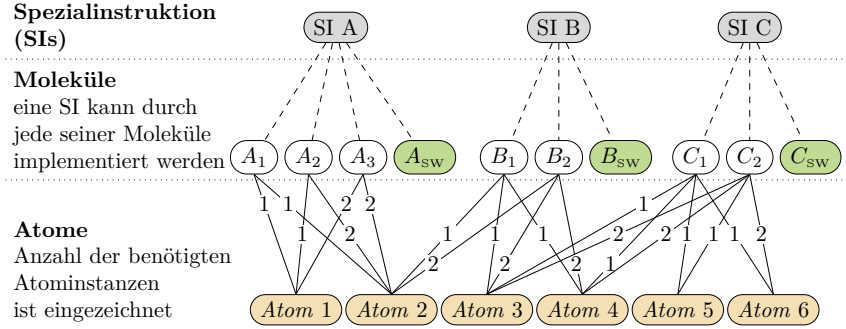


Abbildung 3.12: Hierarchische Zusammensetzung von Spezialinstruktionen (SIs) [14].

Molekülen vorkommen und von verschiedenen Spezialinstruktionen eingesetzt werden. Der *i*-Core verfügt standardmäßig über sechs sogenannte *Atom Container*, in denen je ein *Atom* geladen werden kann. Die *Atom Container* sind über ein *Interconnect* und einer *Load/Store-Unit* (LSU) direkt mit dem *Tile-local Memory* der Kachel verbunden, um eine schnelle Anbindung an den Speicher zu gewährleisten. Die *Atom Container* verfügen über zwei 32 bit Eingänge, zwei 32 bit Ausgänge und mehreren Kontroll- und Zustands-Signalen sowie einem Takteingang. Ein *Atom* wird in einen solchen Container eingebettet und kann Operationen mit zwei 32 bit großen Eingabedaten realisieren und 32 bit Ausgabewerte liefern. In Abbildung 3.13 ist ein Teil der rekonfigurierbaren Einheit mit ihren *Atom Containern*, *Interconnect*, *Load/Store-Unit* und der dazu gehörenden *Address Generation Unit* (AGU) schematisch dargestellt. Die Busstruktur zwischen den *Atom Containern* und die Schnittstelle zur *Integer Unit* (IU) des LEON3 sind fest definiert und können nicht zur Laufzeit geändert werden. Die rekonfigurierbare Einheit ist über vier Eingangs- und zwei Ausgangs-Registern mit je 32 bit Größe an die *Integer Unit* angebunden und verfügt zusätzlich über Kontrollsignale zur IU. Die geladenen *Atoms* werden über ein 54 bit langes *Very Long Control Word* (VLCW) gesteuert, welches zur Laufzeit generiert wird.

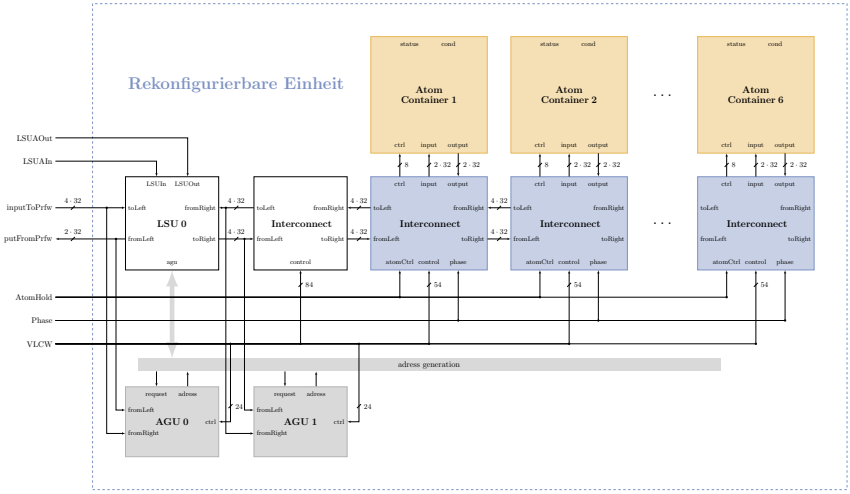


Abbildung 3.13: Teilübersicht der rekonfigurierbaren Einheit des *i*-Cores.

SI Infrastruktur

Zusätzlich zur rekonfigurierbaren Einheit, in der die Spezialinstruktion ausgeführt werden, gibt es noch einige steuernde und unterstützende Module, welche separat von dieser Einheit direkt an die *Integer Unit* angebunden sind. Dazu gehören der *SI State Memory* (SSM), die SI Ausführungseinheit und der Speicher für die jeweiligen *Very Long Control Word*s (VLCWs) [92]:

SI State Memory Wird eine SI in der Pipeline erkannt, wird die zugehörige *SI Identifier* (SI ID) an den *SI State Memory* weitergeleitet. Jede Spezialinstruktion verfügt über eine individuelle ID, welche sich aus einem *Opcode* und einer *SI*-Gruppe mit je 5 bits zusammensetzt. Somit sind bis zu 1 024 verschiedene SIs realisierbar. Der zur SI ID gehörende Eintrag im SSM kodiert den eigentlichen Spezialbefehl und besteht aus Adresse und Anzahl der VLCWs und dem Zustand der aktuellen Implementierung der SI.

SI Ausführungseinheit Erhält vom *SI State Memory* Adresse und Anzahl der benötigten VLCWs und fragt diese beim VLCW Speicher über

den AHB an. Sobald die Pipeline die SI zur Ausführung freigibt, startet die Ausführungseinheit die Verarbeitung der SI durch Anlegen der VLCWs an die rekonfigurierbare Einheit.

VLCW Speicher Im VLCW Speicher werden die VLCWs zur Ansteuerung der *Atoms* gespeichert und können von dort ausgelesen werden. Dieser Speicher ist über eine *Slave*-Schnittstelle an den AHB angebunden, um VLCWs in den Speicher schreiben zu können.

Neben diesen Einheiten für die Ausführung einer Spezialinstruktion, steht zudem noch der sogenannte *Bitstream Loader* zur Verfügung, mit dessen Hilfe ein *Bitfile* eines Beschleunigers in die *Atom-Container* der rekonfigurierbaren Einheit geladen werden kann. Die gegebene Struktur des *i*-Cores und die vorhandenen SI-Module dienen als Basis für die Realisierung des Konzeptes einer transparenten und dynamischen Hardwarebeschleunigung zur Laufzeit, welche in Kapitel 8 entwickelt wird.

4 Adaptive Speicherstruktur mit Mustererkennung

In Abschnitt 3.2 wurde der Einsatz und Aufbau des *Associative Memory* Chips (AM-Chips), welcher für den aktuellen CMS-*Level 1 Track Trigger* zum Einsatz kommt, detailliert beschrieben. Auch wenn in der Regel anwendungsspezifische integrierte Schaltungen (englisch *Application-Specific Integrated Circuit*s (ASICs)) effizienter sind [74], könnte sich eine FPGA Alternative für den Einsatz der Mustererkennung des *Level 1 Track Triggers* als lohnenswert erweisen, da dieser Ansatz sowohl Änderungen im Nachhinein ermöglicht, als auch kürzere Entwicklungszeiten benötigt. Zudem ist dieser Ansatz deutlich flexibler und kann in mehrere der momentan existierenden Konzepte (betrachtet in Unterabschnitt 3.1.1) für den *Level 1 Track Trigger* integriert werden, da keine dedizierte Hardware benötigt wird und der Einsatz von FPGAs für jedes dieser Konzepte vorgesehen ist. Auch können die durch die Flexibilität der FPGAs die Algorithmen der Filterung von Spurbahnen, welches momentan durch den AM-Chip realisiert wird, und Spurrekonstruktion aufeinander abgestimmt werden. Eine Umsetzung dieser Spurfilterung mittels FPGAs schien bisher nicht möglich [79], da die Anzahl der verfügbaren *Configurable Logic* Blöcke der aktuellen und auch zukünftigen FPGAs nicht ausreicht, um die hohe Anzahl von benötigten Mustern zu speichern. In dieser Arbeit wird deswegen kein klassischer *Content Addressable Memory* mit Hilfe eines FPGAs realisiert, sondern eine andere Herangehensweise gewählt. Der Fokus hierbei liegt auf den zu speichernden Daten (in diesem Fall die Muster), diese werden analysiert und für die Speicherung in einem FPGA optimiert, wobei die Funktionalität des *Content Addressable Memories* beibehalten werden soll [HSB+16].

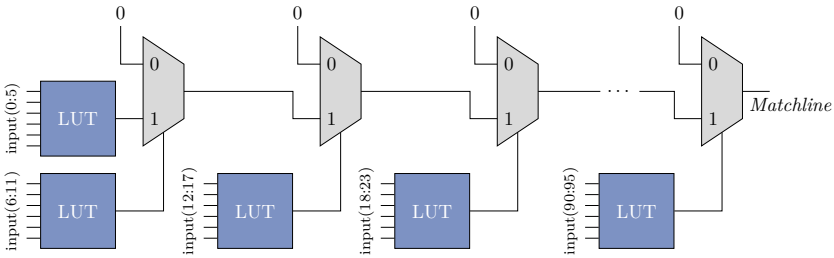


Abbildung 4.1: Vergleichseinheit für ein Muster.

4.1 Konzept der FPGA Speicherstruktur

Im Abschnitt 2.4 wurde der Aufbau eines FPGAs skizziert, welcher sich durch den Aufbau aus *Lookup*-Tabellen nur bedingt für die Realisierung einer Speichereinheit mit Schreib- und Lese-Fähigkeiten eignet. Um eine CAM-Architektur mit einer Wortbreite von n bit zu realisieren, wie sie in Unterabschnitt 2.2.2 beschrieben wird, werden für einen Virtex7 FPGA allein für das Einlesen und Vergleichen der Daten eines Musters mindestens $\lceil \frac{n}{6} \rceil$ *Lookup*-Tabellen benötigt. Eine solche Schaltung für die Wortlänge von 96 bit wird in Abbildung 4.1 dargestellt und besteht aus einer Kaskade von 16 *Lookup*-Tabellen und 15 Multiplexern. Diese Schaltung kann somit ein Muster (bestehend aus 96 bit) speichern, mit einem eingehenden Datenstrom vergleichen und das Ergebnis dieses Vergleiches über die *Matchline* ausgeben. Der im Abschnitt 3.2 beschriebene AM-Chip speichert mehrere zehntausend dieser Muster und jedes dieser gespeicherten Muster benötigt wiederum weitere 16 *Lookup*-Tabellen, was in Summe dann schnell die Anzahl der *Lookup*-Tabellen der heute zur Verfügung stehenden FPGAs sprengt. Die Abbildung eines AM-Chips auf eine FPGA-Architektur ist somit nicht direkt möglich [79]. Um trotzdem die Funktionalität des AM-Chips mit Hilfe einer FPGA-Architektur zu realisieren, wird der Fokus nun auf die zu speichernden Daten verschoben. Dies geschieht, um die Anzahl der benötigten *Lookup*-Tabellen zu reduzieren und basiert auf der Möglichkeit, die zu speichernden Daten logisch zu minimieren. Abbildung 4.2 zeigt eine Schaltung für den Vergleich von zwei verschiedenen Mustern, die sich nur in einem Bit, in diesem Fall dem 15. Bit unterscheiden. Die Anzahl der benötigten *Lookup*-Tabellen und Multiplexer erhöht sich nicht,

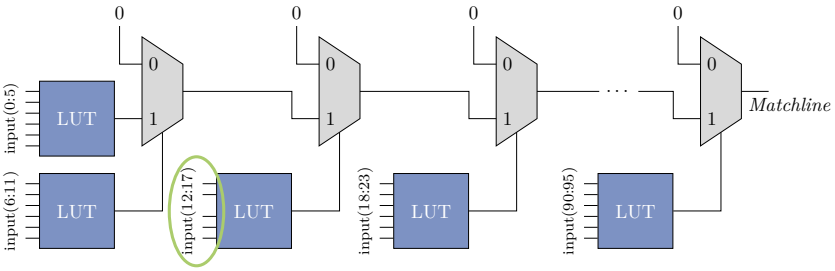


Abbildung 4.2: Vergleichseinheit für zwei Muster.

die Anzahl der benötigten Eingänge wird zudem um eins verringert. Da im Falle des CMS *Level 1 Track Triggers* die zu speichernden Daten (die Muster der Spurbahnen, im Folgenden *Pattern* genannt) vorgeben sind, kann nicht ohne Analyse davon ausgegangen werden, dass eine Minimierung der benötigten Ressourcen per se zu erreichen ist. Aus diesem Grunde wurde ein generierter Datensatz (*Pattern Bank*) verwendet und für verschiedene Anzahlen k von *Pattern* je eine Vergleichseinheit implementiert. Hierbei wird zunächst in *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) ein Design für ein *Pattern* erstellt und durch das Xilinx Tool Vivado für einen FPGA der Virtex 7 Serie synthetisiert. Die *Pattern* werden sukzessive hinzugefügt: ein *Pattern*, welches durch ein Design mit k *Pattern* abgedeckt wird, wird dementsprechend auch durch jedes Design mit mehr als k *Pattern* abgedeckt. Die durchschnittliche Anzahl von benötigten *Lookup*-Tabellen nach der Synthese für Designs mit verschiedener Anzahl von abgedeckten *Pattern* ist in Abbildung 4.3 dargestellt. Im Durchschnitt wird weniger als eine *Lookup*-Tabelle benötigt, falls mehr als tausend *Pattern* in das Design eingebunden werden. Die Anzahl von *Lookup*-Tabellen, die durchschnittlich für ein *Pattern* benötigt werden, kann um bis zu 96 % verringert werden, ist aber abhängig von den gewählten *Pattern*, aber die Ersparnis bei über tausend *Pattern* beträgt konstant um die 95 %. Dies bedeutet, dass eine Einteilung der *Pattern Bank* in mehrere Abschnitte möglich ist, ohne den Effekt der logischen Minimierung zu verlieren, und ermöglicht unter anderem eine parallele Verarbeitung auf mehreren FPGAs. Die ersten Ergebnisse verdeutlichen das Potenzial, welches die logische Minimierung besitzt. Das entworfene Design vergleicht jedoch nur ein Eingangssignal mit den durch die *Lookup*-Tabellen gespeicherten *Pattern*

Anzahl *Pattern*	Anzahl LUT
1	19
2	18
5	29
10	31
20	38
100	78
1000	521
3000	938
5000	2239
10000	4867
15000	7305
200000	83660

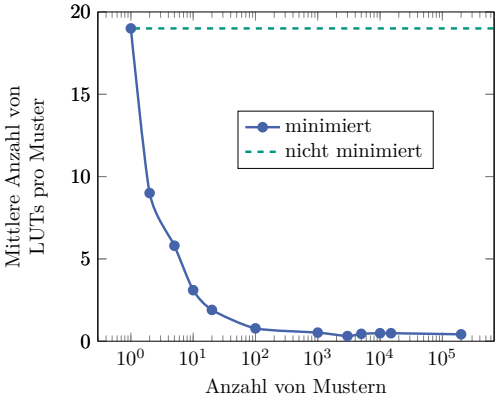


Abbildung 4.3: Reduzierung der benötigten Ressourcen durch logische Minimierung: Ergebnisse der Synthese.

und gibt mit der *Matchline* an, ob das Eingangssignal mit einem dieser *Pattern* übereinstimmt oder nicht. Welches *Pattern* bei einer positiven Überprüfung übereinstimmte, kann durch die Weitergabe des Eingangssignales bestimmt werden. Eine Mehrheitsentscheidung, wie der AM-Chip sie bietet, kann durch diesen Entwurf jedoch nicht effektiv realisiert werden, da hierfür jedes *Pattern* pro betrachteter Lage dupliziert werden müsste und die Bits der entsprechenden Lage bei der Überprüfung übergangen werden müssten. Aus diesem Grunde wird in den folgenden Abschnitten die *Pattern Bank* hinsichtlich der verschiedenen Lagen analysiert und ein System entworfen, welches die eingehenden Daten lagen-weise mit der *Pattern Bank* vergleicht, das Ergebnis zwischenspeichert und dann einer Mehrheitsentscheidung unterzieht.

4.2 Analyse der *Pattern Bank*

In Abschnitt 3.2 wurde die geometrische Einteilung des CMS-Detektors in die verschiedenen *Trigger Tower* erläutert und die Sektortypen *Barrel*, *Endcap* und *Hybrid* vorgestellt. Für jede dieser Sektortypen existiert eine spezifische *Pattern Bank* bestehend aus ein bis drei Millionen verschiedenen

Lage	Verschiedene Sensorwerte	Maximales Vorkommen
0	304	20 461
1	530	10 992
2	828	5 648
3	1 461	3 073
4	2 697	1 813
5	4 223	1 230

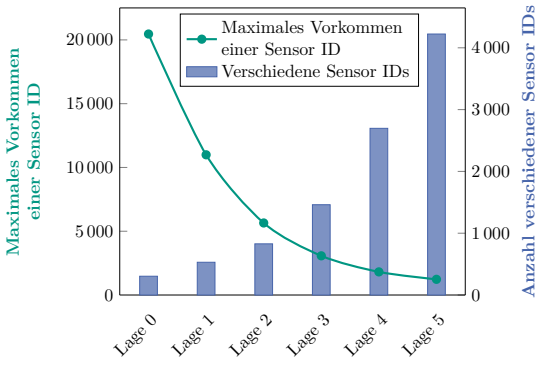


Abbildung 4.4: Analyse der Muster aus denen die *Pattern Bank* für des *Barrel*-Sektors besteht.

Pattern. In Abbildung 4.4 sind die Ergebnisse der Analyse der knapp drei Millionen *Pattern* aus denen die *Pattern Bank* des Barrel-Sektors besteht, illustriert. Pro Lage werden die Anzahl von verschiedenen Sensor IDs und das maximale Vorkommen einer bestimmten Sensor ID in allen *Pattern* dargestellt. In der ersten Detektorlage (Lage 0) existieren beispielsweise 304 verschiedene Sensor IDs und der Sensor mit der ID 2 059 kommt in 20 461 verschiedenen *Pattern* dieser *Pattern Bank* vor. In der fünften Lage hingegen existieren 4 223 Sensor IDs, eine einzelne ID kommt jedoch maximal in 1 230 *Pattern* vor. Dies spiegelt die geometrische Form des *Barrel*-Sektors wider, welche in Abbildung 3.5 auf Seite 53 dargestellt ist. In den inneren Lagen existieren weniger Sensormodule als in den äußeren Lagen und ein *Pattern* besteht immer aus sechs Sensor IDs, wobei jeweils eine ID pro Lage enthalten ist. Dementsprechend werden die Sensor IDs der inneren Lage im Durchschnitt wesentlich häufiger in einem *Pattern* vorkommen als die der äußeren Lagen. In Abbildung 4.5 ist zusätzlich zu dem maximalen Vorkommen einer Sensor IDs in allen *Pattern*, auch das minimale und durchschnittliche Vorkommen einer Sensor ID dargestellt. Im Durchschnitt kommt eine Sensor ID in der inneren Lage in 9 678 einzelnen *Pattern* vor, es gibt jedoch auch eine Sensor ID, der nur in 106 *Pattern* vorkommt. Für die äußeren Lagen gibt es sogar Sensor IDs, die nur in einem einzigen *Pattern* vorkommen. Ausgehend von der inneren

Lage	Anzahl von *Pattern*		
	max	ø	min
0	20 461	9 678	106
1	10 992	5 550	5
2	5 648	3 553	21
3	3 073	2 014	1
4	1 813	1 091	1
5	1 230	679	1

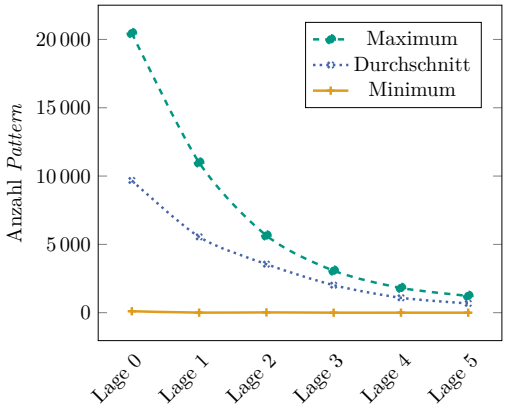


Abbildung 4.5: Anzahl von *Pattern* des *Barrel*-Sektors, die identische Sensor IDs in einer Lage beinhalten.

Lage nimmt der Unterschied zwischen maximalen und durchschnittlichen Vorkommen einer Sensor ID in allen *Pattern* signifikant ab. Ähnlich wie bei dem *Barrel*-Sektor verhält es sich bei der *Pattern Bank* des *Endcap*-Sektors, auch hier existieren weniger Sensormodule in den inneren als in den äußeren Lagen. Jedoch ist der Unterschied geringer, was darauf zurück zu führen ist, dass in diesem Sektor mehr als sechs Sensorlagen liegen und ein *Pattern* nicht Sensor IDs für jede Lage enthalten kann. In diesem Fall entspricht eine Sensorlage des CMS-Detektors somit nicht mit einer Lage innerhalb des *Pattern* überein. Zudem wurden die Daten vor der Analyse von einem Pseudosignal befreit, welches in den äußeren drei Lagen des *Pattern* verwendet wird. Dies geschieht um interessante Teilchenbahnen identifizieren zu können, die bei der geometrischen Anordnung der Sensorlagen in diesem Sektor ansonsten nicht repräsentiert werden könnten, da nicht genügend Sensorlagen von diesem Teilchen durchdrungen werden. Die *Pattern Bank* für diesen Sektor besteht aus knapp zwei Millionen *Pattern*. In Seite 71 ist auf der linken Seite das Ergebnis der Analyse für den *Endcap*-Sektor dargestellt und es ist die gleiche Charakteristik wie in Abbildung 4.4 für den *Barrel*-Sektor zu erkennen. Das Ergebnis der Analyse der *Pattern Bank* des hybriden Sektors, welches auf der rechten Seite in Abbildung 4.6 dargestellt ist, unterscheidet sich jedoch stark von

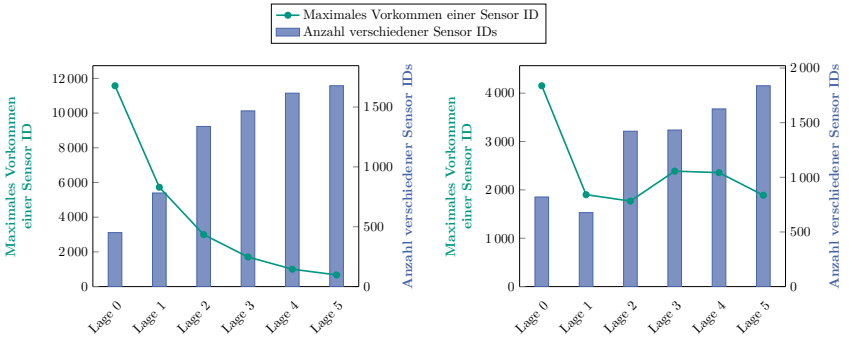


Abbildung 4.6: Analyse einzelner Muster der *Pattern Bank* für den *Endcap*-Sektor (links) und den *Hybrid*-Sektor (rechts).

den Ergebnissen der anderen beiden Sektortypen. Dies ist in Komposition der Sensorlagen begründet, welche sowohl Sensormodule des *Barrels* als auch des *Endcaps* enthält. Dies führt dazu, dass die Anzahl der Sensor IDs vor allem in den äußeren Lagen der *Pattern* sich nicht stark verändert und analog dazu das maximale Vorkommen einer Sensor ID in den *Pattern* dieses Sektors nicht stark variiert. Auch in dem hybriden Sektor wird, vorwiegend in den äußeren Lagen, ein Pseudosignal verwendet, um interessante Teilchenbahnen beschreiben zu können und die *Pattern Bank* für den hybriden Sektor beinhaltet gut eine Millionen verschiedene *Pattern* und enthält somit deutlich weniger *Pattern* als die der anderen Sektortypen. Bei der Analyse des durchschnittlichen Auftretens einer Sensor ID in *Pattern* der *Pattern Bank* deckt sich die Verteilung der Ergebnisse aller drei Sektortypen, wie in Abbildung 4.7 zu sehen ist. Nur die absolute Anzahl der *Pattern* variiert, was durch die Gesamtanzahl der *Pattern* pro Sektor resultiert. Generell zeigt die lagen-weise Analyse der *Pattern* für die einzelnen Sektoren sehr unterschiedliche Ergebnisse, vor allem für die Anzahl der Sensormodule pro Lage, was sich dann in der maximalen Anzahl der *Pattern* widerspiegelt, in denen ein bestimmtes Sensormodul vorkommt. Dieser Effekt wird durch die unterschiedliche Anzahl von *Pattern* pro Sektor von eins bis zu drei Millionen noch verstärkt. So existiert für den hybriden Sektor ein Sensormodul, welches in über 4 000 *Pattern* vorkommt, in der *Pattern Bank* des *Barrel*-Sektors ist diese Anzahl jedoch

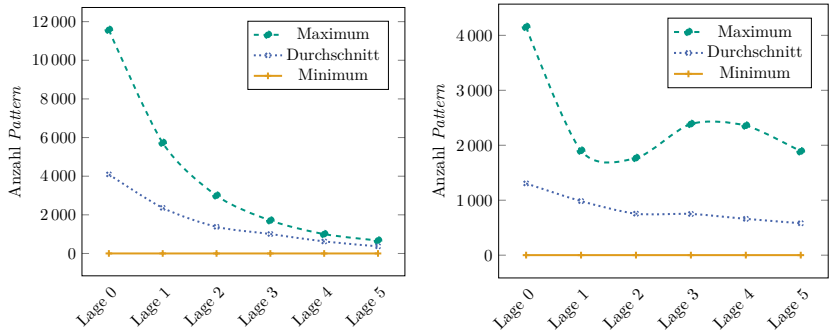


Abbildung 4.7: Anzahl von aktivierten *Pattern*-Nummern des *Endcap*- und des *Hybrid*-Sektors.

um den Faktor fünf größer und es existiert ein Sensormodul was mehr als 20 000 *Pattern* vorkommt. Die *Pattern Bank* für den *Barrel*-Sektor beinhaltet mit Abstand die meisten *Pattern*, variiert bei der Anzahl der Sensormodule pro Lagen am stärksten und besitzt sowohl die größte Anzahl der Sensormodule pro Lage als auch das maximale Vorkommen einer bestimmten Sensor ID in allen *Pattern*. In der vorliegenden Arbeit wird aus diesem Grunde die *Pattern Bank* des *Barrel*-Sektors verwendet, um ein FPGA-Design für den Vergleich der eingehenden *Stubs* mit einer *Pattern Bank* zu entwerfen. In den folgenden Abschnitten wird erläutert, in welcher Weise die in der Analyse verwendeten Parameter die Begrenzungen für das FPGA-Design beeinflussen.

4.3 Aufbau der FPGA Speicherstruktur

Wie in Abschnitt 4.1 erläutert, müssen die eingehend *Stubs* lagen-weise mit der Pattern Bank bestehend aus k *Pattern* verglichen, die Ergebnisse zwischenspeichert und dann einer Mehrheitsentscheidung unterzogen werden. Hierfür werden die enthaltenen *Pattern* durchnummeriert, jedes *Pattern* erhält also eine eindeutige *Pattern*-Nummer $n \in \mathbb{N}$ und alle *Pattern*-Nummern bilden zusammen die Menge N und es gilt $|N| = k$. Die Speicherstruktur liest die eingehenden *Stubs* ein, welche die IDs der

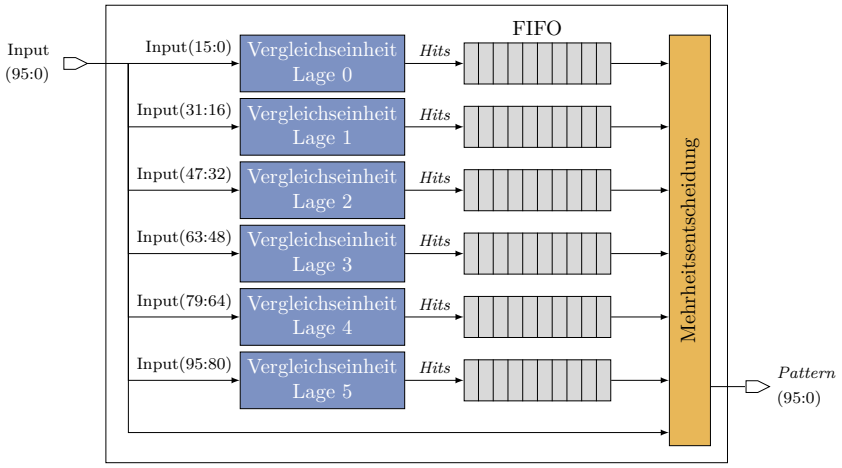


Abbildung 4.8: Aufbau der FPGA Speicherstruktur.

aktivierten Sensoren enthalten. Im folgenden werden diese Sensor IDs *Hits* genannt und ein $h \in H$ ist eine mögliche Sensor ID. Pro Lage wird ein *Hit* $h \in H_L$ eingelesen, mit dem entsprechenden Teil der *Pattern Bank* verglichen und für die übereinstimmenden *Pattern* werden die jeweiligen *Pattern*-Nummern $n \in N$ in die Zwischenspeicher-Einheit geschoben. In einem letzten Schritt wird dann überprüft, ob eine *Pattern*-Nummer in fünf Lagen aktiviert wurde. Falls dies der Fall ist, wird das zu der Nummer gehörende *Pattern* ausgegeben. Wie in Unterabschnitt 3.1.2 beschrieben, ist für den *Level 1 Track Trigger* eine Latenz von 5 μs eingeplant, wobei die Vorfilterung durch die Module und der Transport der Triggerdaten von den Sensoren hin zu der *Level 1 Track Trigger*-Einheit inkludiert ist. Somit soll die zu entwickelnde Speicherarchitektur innerhalb von ungefähr 2 μs alle Nummern der übereinstimmenden *Pattern* ausgegeben haben, damit die Gesamtlatenz von 5 μs eingehalten werden kann. Abbildung 4.8 zeigt den Aufbau des entworfenen Systems mit einer Vergleichs-, Zwischenspeicher- und Entscheidungs-Einheit. In den folgenden Abschnitten werden diese drei verschiedenen Einheiten detailliert beschrieben.

4.3.1 Vergleichseinheit

Die Vergleichseinheit für eine Lage L liest einen *Hit* $h \in H_L$ ein und gibt die dazu passenden *Pattern*-Nummern $n \in \mathbb{N}$ aus. Hierzu wird für jede Lage L eine Abbildung

$$p_L : H_L \rightarrow N_L \tag{4.1}$$

definiert, welche durch den zu der Lage L gehörenden Teil der generierten *Pattern Bank* bestimmt wird. Einem *Hit* $h_L \in H_L$ können mehrere *Pattern*-Nummern zugeordnet werden, $h_l \mapsto \{n_{h0}, n_{h1}, \ldots, n_{hl}\} = N_{L,h_l}$ und die Menge $N_{L,h} \subset N_L$ beinhaltet alle *Pattern*-Nummern, die durch einen bestimmten Hit h in dieser Lage erfüllt werden. $N_{L,max}$ ist die Kardinalität der mächtigsten Menge $N_{L,h}$ für alle $h \in H_L$. Wenn durch die Vergleichseinheit ein *Hit* $h \in H_L$ eingelesen wird, müssen also im schlimmsten Fall $N_{L,max}$ *Pattern*-Nummern weitergegeben werden. Um eine vorgegebene Latenz einzuhalten, werden pro Lage mehrere *Pattern*-Nummern parallel weitergegeben. Die Anzahl der *Pattern*-Nummern, die gleichzeitig weitergegeben werden, wird durch die Taktfrequenz f und die Latenz t bestimmt:

$$l_L = \lceil \frac{N_{L,max}}{t \cdot f} \rceil \tag{4.2}$$

Es werden also l_L verschiedene *Pattern*-Nummern zu einem Paket zusammengefasst und in einem Taktzyklus weitergegeben. Die Vergleichseinheit wird als Zustandsautomat umgesetzt und jeder Zustand s generiert eines dieser Pakete und gibt dieses weiter. Somit werden

$$s_L = \lceil \frac{N_{L,max}}{l_L} \rceil \tag{4.3}$$

Zustände benötigt, um alle $N_{L,max}$ *Pattern*-Nummern zu erzeugen und weiter zu geben, wobei dann $s \leq (t \cdot f)$ gilt. Die *Pattern*-Nummern werden streng aufsteigend erzeugt und weitergeleitet, alle *Pattern*, die zu einem Zeitpunkt t_0 übertragen werden, besitzen eine kleinere *Pattern*-Nummer als die *Pattern*, die zum Zeitpunkt t_1 übertragen werden, wenn $t_0 < t_1$ gilt. Auch innerhalb eines Zeitschrittes sind die *Pattern* streng aufsteigend nach ihrer *Pattern*-Nummer auf die l_L Verbindungen sortiert, so dass die *Pattern*-Nummer, welche über die Verbindung l_1 übertragen wird, kleiner ist als die *Pattern*-Nummer, die über die Verbindung l_2 übertragen wird und so weiter. In Abbildung 4.9 ist der schematische Aufbau der Vergleichsmatrix einer

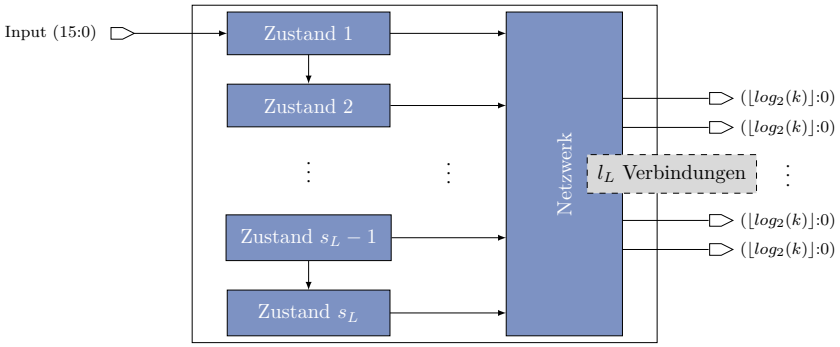


Abbildung 4.9: Struktur der Vergleichseinheit für eine Lage L.

Lage L für die Überprüfung von k gespeicherten *Pattern* dargestellt. Das Eingangssignal von 16 bit wird sukzessive durch jeden Zustand eingelesen und in jedem Schritt werden dann l_L *Pattern*-Nummern zu einem Netzwerk weitergeleitet. Dieses Netzwerk verteilt die empfangenen *Pattern*-Nummern auf die l_L Verbindungen zu der Zwischenspeichereinheit. Pro *Pattern*-Nummer werden $\lceil \log_2(k) \rceil$ Bits benötigt, um die gespeicherten k *Pattern* repräsentieren zu können. Dieser Parameter k ist global für das ganze System fest und variiert nicht für die einzelnen Lagen L. Die Anzahl der benötigten Zustände s_L und Verbindungen l_L hingegen sind abhängig von der Beschaffenheit der *Pattern Bank* und unterscheiden sich für jede Lage L. In Abschnitt 4.2 wurde gezeigt, dass im *Barrel*-Sektor die höchste Anzahl $N_{L,max}$ von aktivierten *Pattern*-Nummern erzeugt wird und somit stellt der *Barrel*-Sektor die höchsten Anforderungen an das zu entwickelte System und benötigt die meisten Zustände und Verbindungen. Zudem gilt für diesen Sektor: $N_{L5,max} < N_{L4,max} < \ldots < N_{L0,max}$, wobei $|H_{L0}| < |H_{L1}| < \ldots < |H_{L5}|$ gilt. Dies bedeutet, dass für die innere Lage $L0$ die meisten Zustände und somit FPGA-Ressourcen benötigt werden.

4.3.2 Zwischenspeichereinheit

Die Zwischenspeichereinheit einer Lage L empfängt die durch die Vergleichseinheit ausgewählten *Pattern*-Nummern und puffert sie für die Mehrheitsentscheidung. Dazu werden $\sum_{L=0}^{5} l_L$ kleine Datenstrukturen

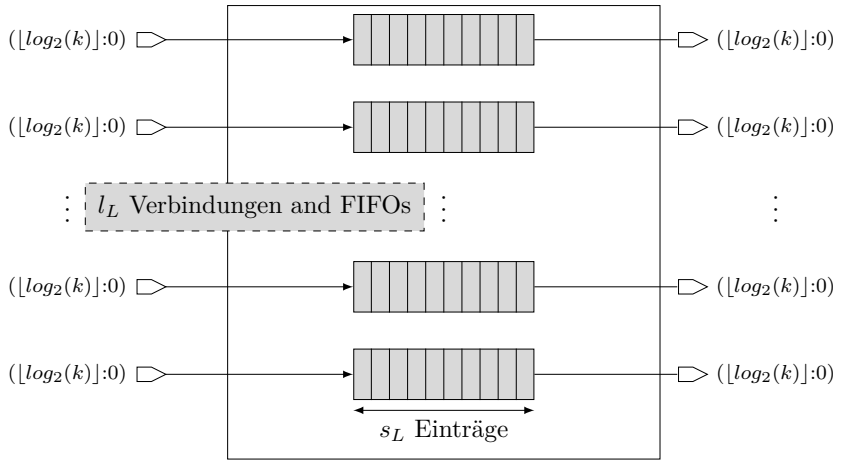


Abbildung 4.10: Struktur der Zwischenspeichereinheit für eine Design mit k *Pattern* für eine Lage L.

zur Verfügung gestellt, welche nach dem FIFO-Prinzip (*First In – First Out*) arbeiten, eine Wortbreite von $\lceil \log_2 k \rceil$ bits besitzen und s_L Einträge beinhalteten. Der Parameter s_L ist für alle Lagen L identisch und bedingt die Anzahl l_L von FIFOs für jede Lage, welche ebenso von der maximalen Anzahl von möglichen *Pattern*-Nummern $N_{L,max}$ beeinflusst wird. In Abbildung 4.10 ist diese Struktur für die Pufferung der *Pattern*-Nummern für eine Lage L dargestellt. Die eingehenden *Pattern*-Nummern werden parallel für alle Lagen des Systems eingelesen und gepuffert. Im Extremfall werden $s_L \cdot l_L$ *Pattern*-Nummern in s_L Takten für die Lage L eingelesen und wieder ausgegeben.

4.3.3 Entscheidungseinheit

Die Entscheidungseinheit empfängt die gepufferten *Pattern*-Nummern und falls eine *Pattern*-Nummer in mindestens fünf der sechs Lagen vorkommt, wird das dazu gehörende *Pattern* ausgegeben. Der Aufbau dieser Einheit ist in Abbildung 4.11 dargestellt und besteht im Wesentlichen aus einer Auslese-Einheit und einer Einheit, die für die eigentliche Entscheidung

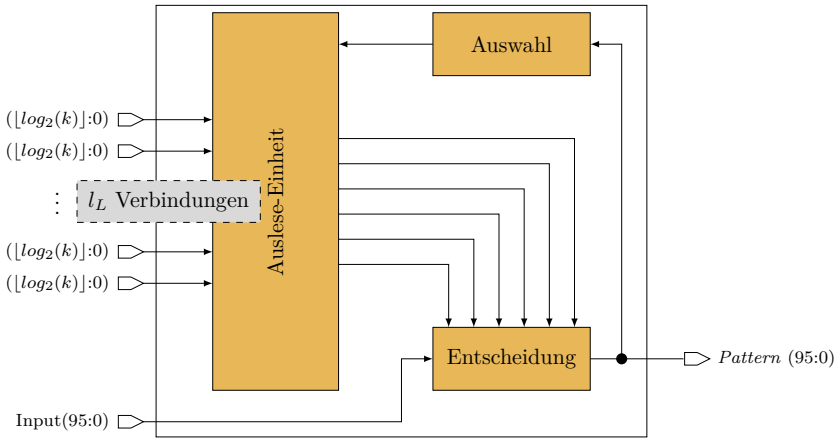


Abbildung 4.11: Struktur der Entscheidungseinheit für ein Design mit k *Pattern*.

verantwortlich ist. Die Ausleseeinheit kann in jedem Taktschritt maximal $\sum_{L=0}^{5} l_L$ *Pattern*-Nummern einlesen und leitet die kleinste *Pattern*-Nummer jeder Lage L weiter, damit entschieden werden kann, ob ein *Pattern* in mehr als vier Lagen vorkommt. Da die kleinsten *Pattern*-Nummern pro Lage betrachtet werden, kann durch die zwei größten Werte dieser *Pattern*-Nummern entschieden werden, welche *Pattern*-Nummer der restlichen Lagen mit kleineren Werten bereits verworfen werden können, da sie nicht mehr in fünf Lagen übereinstimmen können. Diese Berechnung wird durch die Einheit Auswahl der Entscheidungseinheit durchgeführt, die dann der Ausleseeinheit mitteilt, bis zu welcher *Pattern*-Nummer die eingelesenen Daten verworfen und nachgeladen werden können. Dadurch können *Pattern*-Nummern direkt verworfen werden, ohne dass diese explizit auf eine Übereinstimmung untersucht wurden und beschleunigt das Auslesen der Daten und ermöglicht eine schnelle Bewertung der von allen erzeugten *Pattern*-Nummern. Dieser Vorgang wird solange fortgeführt, bis eine *Pattern*-Nummer in mindestens fünf Lagen vorkommt. Falls die *Pattern*-Nummer nicht in allen sechs Lagen übereinstimmt, wird die entsprechende Lage markiert und die folgenden *Pattern*-Nummern dieser Lage bis zu der detektierten *Pattern*-Nummer nachgeladen. Erst dann wird das

k Pattern	Anzahl LUT	Frequenz in MHz	Latenz in μs max	ø
1 023	6 983	238.83	1.43	0.48
4 095	29 432	220.41	1.63	1.03
8 191	59 655	209.12	1.82	1.09
16 383	134 750	146.26	2.67	0.9
32 767	307 195	150.29	2.59	1.4

Tabelle 4.1: Einzelnes System für *k Pattern*: Ergebnisse der Implementierung für die Virtex 7 Plattform.

dazu korrespondierende *Pattern* ausgegeben. Auf diese Weise werden alle Duplikate, die durch die Form der Entscheidung entstehen können, eliminiert und zusätzlich kann die Information, in welcher Lage das *Pattern* nicht vorkam, weitergeben werden. Dieses Vorgehen vereinfacht die weitere Verarbeitung der Daten, bei denen unter anderem die detektierten *Pattern* einem Vertex zugeordnet werden. Nachdem die letzten *Pattern*-Nummern aus der Zwischenspeichereinheit ausgelesen und überprüft wurden, ist der Vergleich der eingehenden Daten mit der *Pattern Bank* abgeschlossen.

4.4 Evaluierung der FPGA Speicherstruktur

Die in dieser Arbeit entwickelte Speicherstruktur wurde vorrangig für einen FPGA des Herstellers Xilinx [116] entworfen, da diese für den Aufbau des CMS *Level 1 Track Triggers* gewählt worden sind. Trotzdem wurde das System auch für eine FPGA-Familie des Herstellers Altera [3] synthetisiert, um zu zeigen, dass die Speicherstruktur nicht nur auf spezifischen FPGAs eines Herstellers effizient realisiert werden kann. Die Ergebnisse der Implementierungen werden in diesem Abschnitt vorgestellt, analysiert und evaluiert. Zudem werden mehrere Varianten des Systems miteinander verglichen, welche unter anderem eine Aufteilung der *Pattern Bank* auf mehrere FPGAs ermöglichen.

k *Pattern*	Anzahl LUT	Frequenz in MHz	Latenz in μs max	ø
1 023	10 109	279	1.22	0.41
4 095	37 778	268.5	1.34	0.84
8 191	59 607	238.5	1.59	0.95
16 383	99 850	212.1	1.84	0.62
32 767	352 313	204.7	1.91	1.03

Tabelle 4.2: Einzelnes System für k *Pattern*: Ergebnisse der Synthese für die Stratix V Plattform.

4.4.1 Einzelnes System

Die in VHDL entworfene Speicherstruktur wurde für die Virtex 7 FPGA Familie mit Hilfe der vom Hersteller Xilinx zur Verfügung gestellten Software Vivado implementiert. Als Alternative hierzu wurde die StratixV Familie des Herstellers Altera gewählt und die Software Synplify Pro verwendet. Für die Ergebnisse in diesem Abschnitt wurden der Xilinx xc7v2000tflg1925 und der Altera 5sgxea4h2, beide mit einem *Speed Grade* von -2 als Zielplattform, verwendet. Alle implementierten und synthetisierten Designs haben eine feste FIFO-Länge von $s_L = 400$ Einträgen und die Latenz variiert somit mit der erzielten Taktfrequenz f. Tabelle 4.1 zeigt die Ergebnisse für ein einzelnes System, welches für die Virtex 7 Plattform implementiert wurde. Die Anzahl k von *Pattern*, welche die Speicherstruktur abdeckt, wurde sukzessive erweitert. Hierbei wurden die *Pattern* so ergänzt, dass ein Design mit mehr *Pattern* alle *Pattern* abdeckt, welche durch ein Design mit weniger *Pattern* abgedeckt werden. Die Anzahl der *Pattern* wurde so gewählt, dass die Wortbreite der FIFOs von $\lceil \log_2 k \rceil$ komplett ausgenutzt werden kann und somit ist k eine Zweierpotenz minus eins ($k = 2^n - 1, n \in \mathbb{N}$). Die Ergebnisse der Synthese für die identischen VHDL-Designs für die Stratix V Plattform sind in Tabelle 4.2 dargestellt. Für beide Plattformen sinkt die Taktfrequenz mit steigender Anzahl von *Pattern*, was zu einer Erhöhung der Latenz führt, da die Anzahl von FIFO-Einträgen pro FIFO s_L konstant bleibt und nur die Anzahl der FIFOs $\sum_{L=0}^{5} l_L$ entsprechend der Anzahl von *Pattern*-Nummer lagen-weise unabhängig voneinander erhöht wird. Für die Xilinx Plattform sinkt die Taktfrequenz beispielsweise um 37%, wenn die Anzahl der *Pattern* von

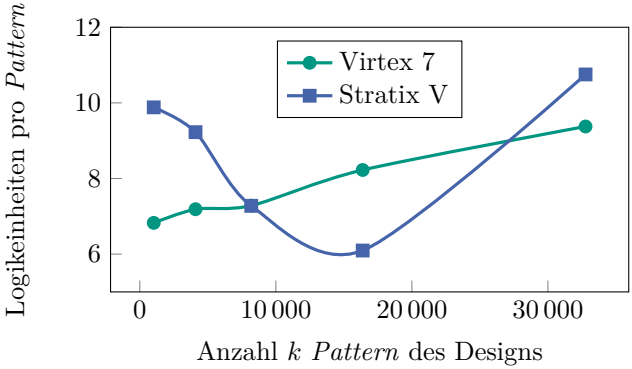


Abbildung 4.12: Benötigte Ressourcen in Zusammenhang mit der Anzahl von berücksichtigten *Pattern*.

$(2^{10} - 1)$ auf $(2^{15} - 1)$ erhöht wird. Ebenso sinkt bei der Altera Plattform die Taktfrequenz um 27%, wobei sie generell höher ausfällt als bei der Plattform von Xilinx, wobei bei der Altera Plattform eine Synthese des Designs vorgenommen wurde und auf der Xilinx das Design implementiert und getestet wurde. Die Latenz wird für beide Plattformen sowohl für *Worst-Case*-Szenario errechnet, wenn die maximale Anzahl an FIFO Einträgen geschrieben, gelesen und miteinander verglichen werden müssen, als auch für den durchschnittlichen und minimalen Fall. Für die Berechnung der Ergebnisse wurde das Design für jede Anzahl k von *Pattern* einzeln lagen-weise analysiert, für ein Design von 1 023 *Pattern* werden beispielsweise in der ersten Lage maximal 1 023 *Pattern* aktiviert und auf drei FIFOs mit der FIFO-Länge 400 verteilt. Dies führt zu maximal 341 Einträgen pro FIFO, die ein- und ausgelesen werden müssen. Die Latenz und die Anzahl der FIFOs ist somit sehr stark an die Beschaffenheit der *Pattern Bank* geknüpft und wird in Unterabschnitt 4.4.3 noch eingehend untersucht. Als obere Grenze für die Latenz kann die maximal definierte Anzahl von 400 FIFO-Einträgen dienen, die jedoch wie für das erläuterte Beispiel des Designs für 1 023 *Pattern*, im Normalfall nicht erreicht werden. Die in Tabelle 4.1 und Tabelle 4.2 dargestellten Latenzen sind durch eine Analyse der möglichen Eingabe-Parameter und der verwendeten *Pattern Bank* bestimmt worden. Die durchschnittliche Latenz wird hierbei durch

k Pattern	Vergleichseinheit	Zwischenspeichereinheit	Entscheidungseinheit
1 023	5 578	362	1 043
4 095	27 296	815	1 321
8 191	55 700	2 313	1 642
16 383	125 922	6 802	2 026
32 767	297 691	7 411	2 093

Tabelle 4.3: Detaillierte Ressourcenverteilung für ein einzelnes System mit k *Pattern* für die Xilinx Plattform.

die durchschnittliche Anzahl von weitergeleiteten *Pattern* errechnet, falls für jede Lage *Pattern* aktiviert werden. Ähnlich wie in der Analyse für die gesamte *Pattern Bank* in Abschnitt 4.2 liegt die durchschnittliche Anzahl von aktivierten *Pattern* deutlich unter der maximalen Anzahl und somit reduziert sich die durchschnittliche Latenz im Vergleich zu der maximalen signifikant um bis zu 66%. In den meisten Fällen kann die angestrebte Latenz von 2 μs auf beiden Plattformen eingehalten werden, nur bei über zehntausend *Pattern* bei der Xilinx-Plattform wird die Latenz im *Worst-Case* verletzt, was jedoch nur durch Hinzunahme weiterer FIFOs abgemildert werden kann. Falls in mehr als einer Lage keine *Pattern* aktiviert werden, kann der Vergleich direkt abgebrochen werden und die Latenz ist dann minimal. Im Gegensatz zur Latenz ist die Anzahl der benötigten Ressourcen des Systems nicht abhängig von den Eingangsdaten, sondern nur von der gespeicherten *Pattern Bank*. In Abbildung 4.12 sind die benötigten Ressourcen für beide FPGA-Plattformen in Relation zu der Anzahl von berücksichtigten *Pattern* illustriert. Im Gegensatz zu den ersten Ergebnissen in Abschnitt 4.1 steigt die Anzahl der im Mittel benötigten *Lookup*-Tabellen pro *Pattern* mit steigender Anzahl von abgedeckten *Pattern*. Trotzdem bleibt auch bei diesem Design die Anzahl der benötigten *Lookup*-Tabellen (LUTs) pro *Pattern* immer unter der Anzahl von 16 *Lookup*-Tabellen (LUTs), die benötigt werden, um die 96 bit aus denen ein *Pattern* besteht, zu vergleichen. Somit erreicht auch das entworfene Design eine Minimierung der zu speichernden Daten und ist dabei auch noch in der Lage, ein *Pattern* zu erkennen, welches nur in fünf der sechs Lagen übereinstimmt. Für den Xilinx FPGA werden 307 195 *Lookup*-Tabellen (LUTs) benötigt, um $k = 32\,767$ *Pattern* mit den eingelesenen *Stubs* vergleichen zu können, dies bedeutet, dass im Schnitt pro *Pattern* $9,4$ *Lookup*-Tabellen (LUTs) verwendet werden. Für den FPGA

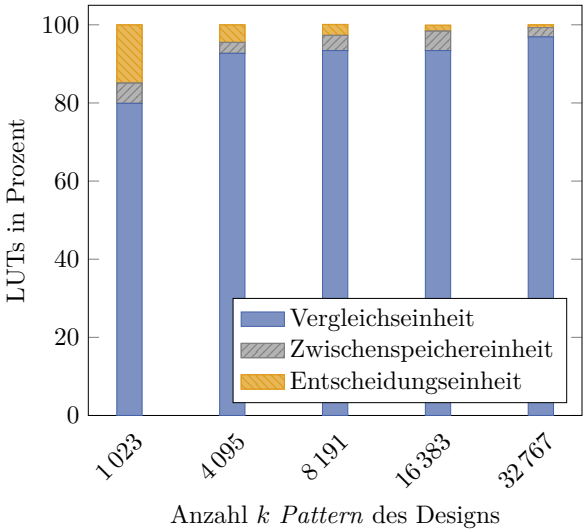


Abbildung 4.13: Detaillierte Ressourcenverteilung für ein einzelnes System mit k *Pattern*.

von Altera variiert die Anzahl der benötigten Logikeinheiten pro *Pattern* deutlich mehr als bei der Virtex 7 Plattform. Zudem gibt es keine direkte Korrelation zwischen den verwendeten Logikeinheiten von Altera FPGAs und den *Lookup*-Tabellen (LUTs) von Xilinx. Im folgenden wird detaillierter auf den Ressourcenverbrauch eingegangen und analysiert, in welchen Teilbereichen des Designs die meisten Ressourcen verwendet werden. Die Ressourcenverteilung auf die drei wesentlichen Bausteine (Vergleichseinheit, Zwischenspeichereinheit und Entscheidungseinheit) ist in Tabelle 4.3 eingetragen. Hierfür wurden die Ergebnisse der Implementierung für die Xilinx Virtex 7 Plattform gewählt, die Ergebnisse für die Altera Plattform liefern eine korrespondierende Verteilung. In Abbildung 4.13 werden die Anteile der einzelnen Einheiten an einem Gesamtsystem für k *Pattern* illustriert. Der prozentuale Anteil der Vergleichseinheit wächst mit steigender Anzahl k von gespeicherten *Pattern* und beträgt zwischen 80 und 97%. Eine höhere Anzahl von *Pattern* steigert meist die Komplexität der Vergleichseinheit. Durch jedes zusätzliche *Pattern* kann die Anzahl

	Einzelnes Design k		Doppeltes Design $k = (k_1 + k_2)$	
k *Pattern*	LEs	Frequenz in MHz	LEs	Frequenz in MHz
8 191	43 916	224.9	63 546	257.5
16 383	72 329	221.8	103 371	216.8
32 767	316 508	209.4	173 967	217.2
64 535			668 183	247.8

Tabelle 4.4: Ergebnisse der Implementierung für einzelne und doppelte Designs für die Altera Plattform.

$N_{L,max}$ von maximal erzeugten *Pattern*-Nummern einer Lage steigen und somit steigt analog dazu die Anzahl der Zustände s_L der Vergleichseinheit für diese Lage. Die Zwischenspeichereinheit wird durch zusätzliche *Pattern* nicht zwangsweise vergrößert, denn erst wenn die Anzahl $N_{L,max}$ die Kapazität der vorhandenen FIFOs übertrifft, wird ein weiterer FIFOs hinzugefügt. Die Entscheidungseinheit wird zudem auch erst durch eine Erweiterung der Vergleichseinheit beeinflusst und nimmt für eine größere Anzahl von k *Pattern* somit einen immer kleineren Prozentsatz der verwendeten Ressourcen in Anspruch. Insgesamt ist zu beobachten, dass die Anzahl der benötigten *Lookup*-Tabellen (LUTs) pro *Pattern* steigt und dass dies vor allem auf die Vergleichseinheit zurückzuführen ist. Sowohl die Zwischenspeichereinheit als auch die Entscheidungseinheit spielen mit steigender Anzahl von *Pattern* eine immer geringere Rolle. Um eine größere Dichte an *Pattern* pro *Lookup*-Tabelle (LUT) zu erreichen, wird deswegen im folgenden Abschnitt ein doppeltes Design evaluiert, welches aus zwei einzelnes Systemen besteht.

4.4.2 Doppeltes System

Ein doppeltes Design besteht aus zwei einzelnen Systemen, die eingehenden *Stubs* werden gleichzeitig an beide Designs weiter geleitet und verarbeiten somit identische Eingangsdaten. Die gespeicherten *Pattern* sind disjunkt und die aktivierten *Pattern* jedes Designs werden zusammengeführt und dann ausgegeben. Ein doppeltes Design, welches die *Pattern* $n \in N$ abdeckt, besteht also aus der Zusammenfassung eines Designs, welches die *Pattern* $n \in N_1$ abdeckt und eines Designs, welches die *Pattern* $n \in N_2$ abdeckt, wobei $N = N_1 \cup N_2$ und $N_1 \cap N_2 = \emptyset$ gilt. Im Folgenden gilt,

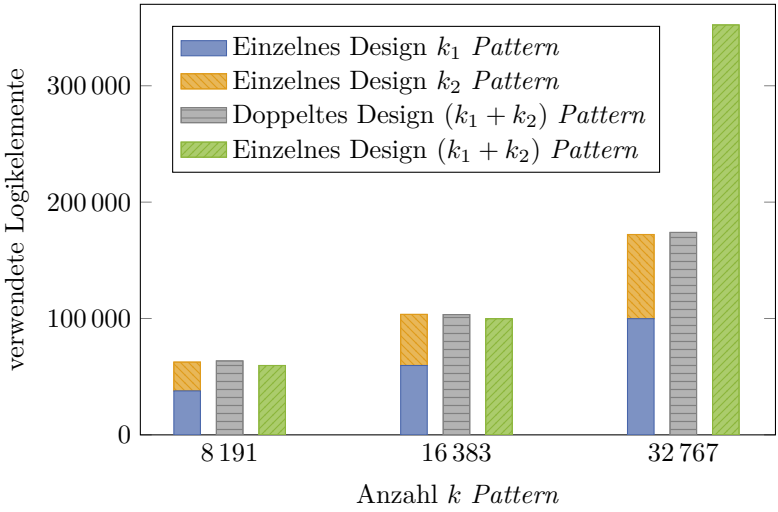


Abbildung 4.14: Vergleich der verwendeten Ressourcen für zusammengesetzte und einzelne Designs.

dass beide zusammengesetzten Designs die gleiche Anzahl von *Pattern* abdeckt, also $|N_1| = |N_2|$ gilt. Zudem gilt $k_1 = |N_1|$, $k_2 = |N_2|$ und somit ist $k = k_1 + k_2 = |N_1| + |N_2| = |N|$ die Anzahl abgedeckter *Pattern* des jeweiligen Designs. Für die Evaluierung der Ergebnisse wurde die Altera Plattform gewählt, da in Unterabschnitt 4.4.1 für diese Plattform ein unverhältnismäßiger Zuwachs der benötigten Ressourcen bei Erhöhung der Anzahl von *Pattern* zu beobachten ist, welches in Abbildung 4.12 deutlich wird. In Tabelle 4.4 sind die Ergebnisse der Implementierung für diese Plattform zu sehen, in der ersten Spalte für ein einzelnes Design, welches k *Pattern* abdeckt, und in der zweiten Spalte für ein System, welches die identischen k *Pattern* abdeckt, aber aus zwei einzelnen Designs besteht, die jeweils die Hälfte der *Pattern* abdeckt. Hierbei wurde die Trennung der k *Pattern* nummerisch vorgenommen und die Reihenfolge der *Pattern* untereinander nicht verändert. Für eine geringe Anzahl von *Pattern* scheint das einzelne Design weniger Ressourcen zu verwenden, was daran liegt, dass die Zwischenspeichereinheit und die Entscheidungseinheit bei geringer Anzahl von *Pattern* noch einen signifikanten Anteil ausmachen. Da dieser

Anteil mit steigender *Pattern*-Anzahl abnimmt, ist für Designs mit großer Anzahl von *Pattern* ein deutlicher Rückgang der verwendeten Ressourcen zu beobachten. Zudem spiegelt die Tabelle 4.4 nicht die verwendeten Ressourcen für einzelne Designs für k_1 und k_2 *Pattern* wider, diese sind in Abbildung 4.14 im Detail dargestellt. Hier ist zu erkennen, dass das doppelte Design den verwendeten Ressourcen nach zu urteilen, tatsächlich einer parallelen Platzierung der beiden Einzelsysteme entspricht. Die steigende Komplexität der Vergleichseinheit, die teilweise zu einer signifikant steigenden Anzahl durchschnittlicher verwendeter Logikelemente pro *Pattern* führt, kann durch die Teilung in zwei kleinere Designs verhindert werden. Hierdurch wird ein System realisierbar, welches eine hohe Anzahl von *Pattern* speichert und trotzdem noch die harten Anforderungen an die Latenz erfüllt. Wie in Tabelle 4.4 ersichtlich, bleibt die Taktfrequenz für verschiedene Anzahlen von abgedeckten *Pattern* nahezu konstant. Auch konnte durch die Teilung ein Design mit $k = 64\,535$ *Pattern* implementiert werden, was zuvor an den bereitgestellten Ressourcen des FPGAs scheiterte. Dies führt zu dem Schluss, dass eine Aufteilung der *Pattern Bank* in mehrere Teilmengen nicht nur möglich, sondern auch sinnvoll ist. In den folgenden Abschnitt wird die Einteilung der *Pattern Bank* in mehrere Untermengen detailliert untersucht.

4.4.3 Zerlegung der Pattern Bank

Bereits bei der Evaluierung des doppelten Designs zeigte sich eine sehr unterschiedliche Anzahl von verwendeten Logikelementen für verschiedene *Pattern*, auch wenn die Anzahl der *Pattern* konstant bleibt. Um dies näher zu untersuchen, sind für die Altera Plattform zwanzig verschiedene Designs implementiert worden, die jeweils 8 191 *Pattern* abdecken. Die in einem Design gespeicherten *Pattern* bilden ein Paket von *Pattern* und diese Pakete sind disjunkt zueinander und zwei aufeinander folgende *Pattern* Pakete decken *Pattern* ab, die auch in der ursprünglichen *Pattern Bank* aufeinander folgen. In Abbildung 4.15 ist der Ressourcenverbrauch und die erzielte Taktfrequenz für jedes Design, welches je ein *Pattern* Paket beinhaltet, illustriert. Auch hier wurde eine fixe FIFO-Länge von 400 Einträgen gewählt, um die einzelnen Designs miteinander vergleichen zu können, die Anzahl der FIFOs variiert jedoch je nach $N_{L,max}$ des Designs zwischen 22 und 30. Zu beobachten ist, dass die Designs der ersten zwei

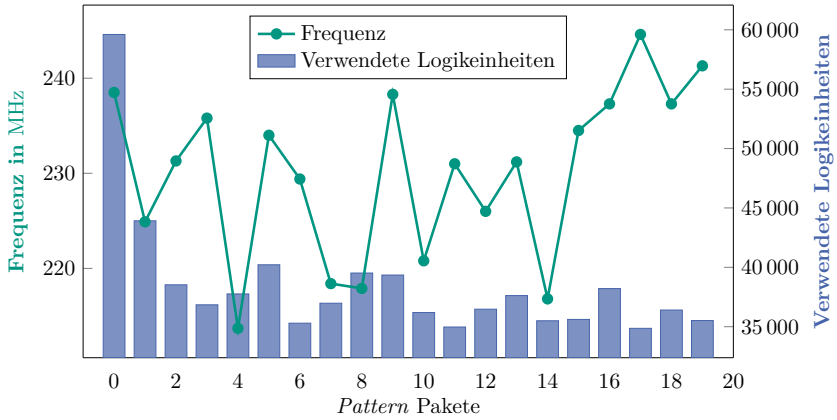


Abbildung 4.15: Ressourcenverbrauch und Taktfrequenz für verschiedene Designs mit gleicher Anzahl von *Pattern*.

Pattern Pakete die meisten Ressourcen in Anspruch nehmen. Dies liegt an der hohen Anzahl $N_{L0,max}$ von *Pattern* die einem bestimmten Hit h zugeordnet sind, und der dazu korrespondierenden niedrigen Kardinalität von H_0, welche zwei beziehungsweise drei beträgt. Dies bedeutet, dass bei einem bestimmten eingehenden *Stub* sehr viele *Pattern*-Nummern durch die Vergleichseinheit generiert und an die Zwischenspeichereinheit weiter geleitet werden müssen. Zudem steigt durch die hohe Anzahl von parallel weitergeleiteten *Pattern*-Nummern gleichzeitig auch die Komplexität der Entscheidungseinheit. Abgesehen von den Designs der ersten zwei *Pattern* Pakete schwankt die Anzahl der verwendeten Logikelemente zwischen 35 000 und 40 000, was einer Abweichung von $12,5\%$ entspricht. Ebenso weichen die bei der Implementierung erreichten Taktfrequenzen der verschiedenen Designs maximal um $12,7\%$ voneinander ab und befinden sich zwischen 213 MHz und 244 MHz. In allen Fällen wird die gesetzte Latenz von 2 μs auf für das Worst-Case-Szenario von maximal generierter Anzahl von *Pattern*-Nummern eingehalten. Vor allem der hohe Ressourcenbedarf der ersten zwei *Pattern* Pakete legt nahe, dass eine geeignete Zerlegung der *Pattern Bank* nötig ist, um eine große Menge an *Pattern* effizient speichern und vergleichen zu können.

5 Effiziente Adaptierung von Speicherinhalten

Die Evaluierung der entworfenen FPGA Speicherstruktur hat gezeigt, dass die Beschaffenheit der *Pattern Bank* einen großen Einfluss auf die Anzahl der benötigten Ressourcen hat. In diesem Kapitel wird die *Pattern Bank* so modifiziert, dass sie möglichst ressourcensparend gespeichert werden kann, ohne dass der Inhalt verfälscht wird. Die Grundidee hierbei ist, dass ähnliche *Pattern* gruppiert werden und somit als Gruppe mit den Eingangsdaten verglichen werden können. Zu diesem Zweck wird eine Umsortierung der *Pattern Bank* durchgeführt und ähnliche *Pattern* werden zu einem Intervall gebündelt [HBWB18].

5.1 Intervallbildung

Um die Anzahl von Schreib- und Lesezyklen der Zwischenspeichereinheit deutlich zu reduzieren, werden *Pattern*, welche in der *Pattern Bank* direkt aufeinander folgen und durch einen bestimmten Hit $h \in H_L$ aktiviert werden, zu einem Intervall zusammengefasst. Ein Intervall wird definiert als:

$$I_{i,j} = [i, j] = \{n \in N_L : i \leq n \leq j\} \tag{5.1}$$

wobei $|I_{i,j}| = j - i + 1$ gilt und es kein Intervall $I_{a,b}$ existiert, für welches $|I_{i,j}| < |I_{a,b}|$ gilt und $i \in I_{a,b}$ oder $j \in I_{a,b}$ ist. Dies bedeutet, dass in dem Intervall $I_{i,j}$ ausnahmslos alle *Pattern* mit der Nummer i bis j liegen, ohne dass ein größeres Intervall gebildet werden könnte. Zusätzlich gilt

$$N_L = \bigcup I_{i,j} \tag{5.2}$$

für alle $I_{i,j}$, jedes *Pattern* $n \in N_L$ liegt also genau in einem Intervall $I_{i,j}$ und $|I_L|$ repräsentiert die Anzahl an benötigten Intervallen für eine Lage L:

$$I_L = \bigcup \{I_{i,j}\} \tag{5.3}$$

und generell gilt $|I_L| \leq |N_L|$. Im Extremfall gilt $|I_L| = |N_L|$, falls für die Lage L keine aufeinanderfolgende *Pattern* mit den gleichen Sensor IDs existieren. Ein Intervall kann durch den linken Endpunkt i und den rechten Endpunkt j definiert und beschrieben werden, sobald drei *Pattern* zu einen Intervall gebündelt werden, wird somit die Anzahl der Bits, die benötigt werden um diese drei *Pattern* zu beschreiben, reduziert. Die in Abschnitt 4.2 durchgeführte Analyse der *Pattern Bank* zeigte vor allen für die inneren Lagen eine geringe Anzahl von $h \in H_L$, was dazu führt, dass in diesen Lagen tendenziell viele *Pattern* zu einem Intervall gebündelt werden können. Anstelle der vielen aufeinander folgenden *Pattern*-Nummern können somit wenige, große Intervalle an die Zwischenspeichereinheit übergeben werden, ohne dass Informationen verloren gehen. Die Evaluierung der entworfenen FPGA Speicherstruktur in Abschnitt 4.4 zeigte, dass sowohl der Ressourcenverbrauch als auch die Latenz des gesamten Systems vorrangig durch die inneren Lagen und deren hohen Anzahl von *Pattern* mit gleichen Sensor IDs dominiert wird. Für diese inneren Lagen gilt $|I_L| \ll N_{L,max} \leq |N_L|$ und die Bündelung von *Pattern* zu Intervallen verspricht eine große Reduzierung der zu speichernden und weiter zu verarbeiteten Daten. Für die äußeren Lagen ist auch der Extremfall $|I_L| = |N_L|$ möglich, in diesem Fall wird die Anzahl der Bits, die benötigt werden um ein *Pattern* zu repräsentieren, durch die Transformation zu einem Intervall mit rechtem und linkem Endpunkt verdoppelt. Jedoch die Anzahl von *Pattern*, die für einen bestimmten Hit h verarbeitet werden müssen, ist für diese Lagen gering und somit auch die identische Anzahl von Intervallen. Eine Verdopplung der Bits hat in diesem Fall keinen großen Einfluss auf das Gesamtsystem.

5.2 Umsortierung der Pattern Bank

Wie im voran gegangenen Abschnitt erläutert, hat die Reihenfolge der *Pattern* innerhalb der *Pattern Bank* einen großen Einfluss auf die Bündelung von *Pattern* zu einem Intervall. In Tabelle 5.1 sind die ersten zehn *Pattern*

Pattern #	Lage 0	Lage 1	Lage 2	Lage 3	Lage 4	Lage 5
1	0	144	273	2309	2703	2967
2	0	145	271	2305	2695	2954
3	0	146	273	260	650	910
4	0	146	273	261	653	911
5	0	146	274	262	2705	2967
6	0	146	274	264	660	924
7	0	146	274	2311	2705	2965
8	0	146	274	2314	4761	4868
9	0	146	275	262	656	918
10	0	146	275	263	657	917

Tabelle 5.1: Die ersten zehn *Pattern* der *Pattern Bank* für die *Barrel*-Sektor.

der *Pattern Bank* für den *Barrel*-Sektor eingetragen, die *Pattern* sind lagen-weise nach dem Wert ihrer Sensor IDs sortiert. Für diese Teilmenge der *Pattern Bank* können in Summe 39 Intervalle gebildet werden, für die Lage 0 ist das Intervall $I_{1,10}$ ausreichend, um alle *Pattern* nach ihren unterschiedlichen Sensor IDs zu gruppieren. Dies ist möglich, da nur $h = 0$ an möglichen Sensor IDs vorhanden ist und somit nur für diesen Eingabewert *Pattern* weitergegeben werden. Für die Lage 1 existieren drei verschiedene Sensor IDs und die *Pattern*, welche diese ID beinhalten, können jeweils zu einem Intervall zusammen gefasst werden: $I_{1,1}$ für $h = 144$, $I_{2,2}$ für $h = 145$ und $I_{3,10}$ für $h = 146$. Somit entsteht die Menge $I_{L1} = \{I_{1,1}, I_{2,2}, I_{3,10}\}$ aller gebildeten Intervalle und $|I_{L1}| = 3$ beschreibt die Anzahl der Intervalle für die Lage L_1. Für die zweite Lage können hingegen nicht alle *Pattern* mit gleichen Sensor IDs zu einem Intervall gebündelt werden, da für $h = 273$ die Intervalle $I_{1,1}$ und $I_{3,4}$ (in Tabelle 5.1 grün markiert) benötigt werden, da das *Pattern* mit der Nummer 2 eine andere Sensor ID enthält. Die Kardinalität von I_{L2} kann durch eine Umsortierung der ersten beiden *Pattern* um eins verringert werden. In diesem Fall wird auch kein Intervall der anderen Lagen durch diese Umsortierung gespalten, was im Allgemeinen nicht der Fall ist. Die orange markierten Intervalle $I_{1,1}$ und $I_{5,5}$ der fünften Lage mit $h = 2967$ können beispielsweise nicht miteinander vereint werden, ohne dass ein Intervall einer anderen Lage dadurch geteilt

Pattern #	Lage 0	Lage 1	Lage 2	Lage 3	Lage 4	Lage 5
2	0	145	271	2305	2695	2954
1	0	144	273	2309	2703	2967
3	0	146	273	260	650	910
4	0	146	273	261	653	911
10	0	146	275	263	657	917
9	0	146	275	262	656	918
5	0	146	274	262	2705	2967
7	0	146	274	2311	2705	2965
6	0	146	274	264	660	924
8	0	146	274	2314	4761	4868
	$I_{L0} = 1$	$I_{L1} = 3$	$I_{L2} = 4$	$I_{L3} = 9$	$I_{L4} = 9$	$I_{L5} = 10$

Tabelle 5.2: Geeignete Umsortierung der *Pattern Bank* mit minimaler Anzahl von Intervallen $I_{total} = 36$.

wird. Durch eine geeignete Umsortierung kann aber die Gesamtanzahl von Intervallen für alle sechs Lagen

$$I_{total} = \sum_{L=0}^{5} |I_L| \tag{5.4}$$

verringert werden. Für die in Tabelle 5.1 dargestellte ursprüngliche Reihenfolge der *Pattern Bank* (1,2,3,4,5,6,7,8,9,10) können insgesamt $I_{totoal} = 39$ Intervalle gebildet werden, durch eine Änderung der Reihenfolge in (2,1,3,4,10,9,5,7,6,8) wie in Tabelle 5.2 dargestellt, um drei Intervalle auf $I_{total} = 36$ Intervalle reduziert werden. Dies entspricht einer Reduzierung um $7,7\%$, wenn nur zehn *Pattern* der *Pattern Bank* in Betracht gezogen worden sind. In der gesamten *Pattern Bank* des *Barrel*-Sektors befinden sich knapp drei Millionen *Pattern* und das Einsparungspotential von Intervallen steigt mit der Anzahl von *Pattern*. In Abbildung 5.1 ist dies für eine Anzahl von $k = 10\,000$ *Pattern* dargestellt. Für die innere Lage $L = 0$ ist die Gesamt-Anzahl der Intervalle I_{total} und der verschiedenen Sensor IDs gleich, was dem Idealfall entspricht, bei dem alle *Pattern* mit der selben Sensor ID zu einem Intervall gebündelt werden können. Dieser Fall tritt ein, da die *Pattern Bank*, wie bereits erwähnt, lagen-weise aufsteigend nach den Sensor IDs sortiert ist. Für die äußeren Lagen bedeutet dies jedoch,

Lage	Sensor IDs	I_{total}
0	3	3
1	36	75
2	114	747
3	323	3 226
4	721	6 831
5	1 240	9 478

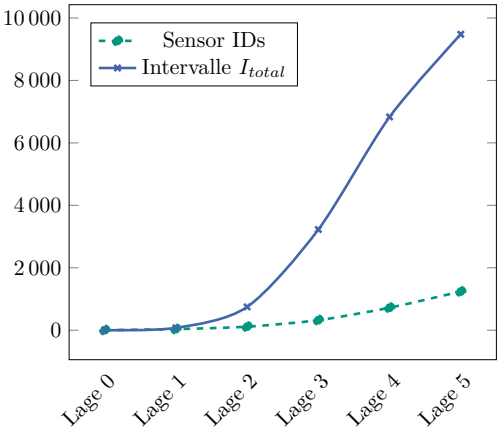


Abbildung 5.1: Analyse der möglichen minimalen und tatsächlichen Gesamtanzahl von Intervallen für jede Lage.

dass deutlich weniger Sensor IDs zu einem Intervall gebündelt werden können. Vor allen für die Lage $L = 5$, welche 1 240 verschiedene Sensor IDs in den ersten 10 000 besitzt, können im Schnitt nur $1,05$ *Pattern* zu einem Intervall gebündelt werden. Im Idealfall könnten um die acht *Pattern* zu einem Intervall gebündelt werden, was aber wiederum Intervalle der anderen Lagen auseinander reißen würde, da ein *Pattern* nur am Stück (für alle Lagen identisch) verschoben werden kann. Eine einfache Permutation der Prioritäten der einzelnen Lagen bei der Sortierung der *Pattern* stellt eine Möglichkeit dar, den Idealfall für je eine Lage zu erreichen. Bei sechs Lagen ergeben sich $6! = 720$ verschiedene Permutationen der Lagen, eine Auswahl von verschiedenen Permutationen und die damit verbundene Gesamtanzahl von Intervallen I_{total} ist in Tabelle 5.3 dargestellt. Das Ergebnis der Analyse aller 720 Permutationen zeigt, dass die ursprüngliche *Pattern Bank* mit der Permutation (0,1,2,3,4,5) in diesem Fall die geringste Anzahl $I_{total} = 20\,360$ von Intervallen für alle Lagen erzielt. Wie jedoch in dem Beispiel der ersten zehn *Pattern* gezeigt werden konnte, ist diese Sortierung nicht optimal und kann signifikant gesenkt werden. Die hohe Anzahl von *Pattern* schließt jedoch einen sogenannten *BruteForce*-Ansatz, bei dem alle möglichen Kombinationen von Reihenfolgen der *Pattern* untereinander

Lagen-weise Permutation	Anzahl an Intervallen						
	Lage 0	Lage 1	Lage 2	Lage 3	Lage 4	Lage 5	I_{total}
012345	3	75	747	3 226	6 831	9 478	20 360
123450	2 791	36	445	2 465	6 156	9 113	21 006
234501	3 347	6 052	114	1 143	4 249	7 728	22 633
345012	3 344	6 453	6 871	323	2 092	5 394	24 477
450123	3 145	6 038	7 123	6 097	721	3 267	26 391
501234	2 076	4 580	6 136	5 983	6 132	1 240	26 147
543210	3 958	6 981	6 741	4 770	3 174	1 240	26 864
432105	3 121	5 109	3 920	2 035	721	8 587	23 493
321054	1 868	2 246	1 125	323	7 881	8 612	22 055
210543	549	428	114	6 268	7 581	7 270	22 210
105432	65	36	6 337	6 002	6 169	5 031	23 640
054321	3	6 430	6 750	5 295	4 322	2 490	25 290

Tabelle 5.3: Anzahl I_{total} an Intervallen für die ersten 10 000 *Pattern* der *Pattern Bank* für eine lagen-weise Permutation der *Pattern*.

ausgerechnet werden, aus. Aus diesem Grund wird das Sortierungsproblem der *Pattern Bank* in den folgenden Abschnitten auf andere kombinatorische Optimierungsprobleme transferiert, die mit Hilfe von Heuristiken gelöst werden können.

5.2.1 Transformation auf Optimierungsprobleme

In Abschnitt 2.5 wurden verschiedene kombinatorische Optimierungsprobleme und deren algorithmische Lösungsansätze besprochen. In den folgenden Abschnitten wird das Sortierungsproblem auf das *Traveling Salesman* Problem transformiert und mit Hilfe der vorgestellten Algorithmen heuristisch gelöst. Anstelle der n Städte, die mit der kürzesten Reisestrecke besucht werden sollen, existieren nun n *Pattern*, welche so sortiert werden sollen, dass die Gesamtanzahl I_{total} an Intervallen minimal ist. Als Distanz zwischen zwei *Pattern* wird die Hamming-Distanz [63] gewählt, welche ein Maß für die Unterschiedlichkeit von Zeichenketten definiert. Die Hamming-Distanz $D_H(n_i, n_j)$ zweier *Pattern* n_i und n_j mit je einer festen Länge von 96 Bits ist hierbei die Anzahl der Lagen, in dem unterschiedlichen Sensor IDs vorkommen. Zwei identische *Pattern* haben somit eine Hamming-Distanz von null und zwei beliebige *Pattern* können maximal

Algorithmus	Anzahl von Intervallen						
	Lage 0	Lage 1	Lage 2	Lage 3	Lage 4	Lage 5	I_{total}
unsortiert	3	75	747	3 226	6 831	9 478	20 360
NN	706	1 661	2 058	2 244	3 630	5 816	16 115
NN 2-Opt	698	1 639	2 015	2 151	3 504	5 695	15 702
MST	1 478	3 548	3 498	2 944	4 109	6 014	21 591
MST 2-Opt	945	2 319	2 334	2 195	3 214	5 118	16 125

Tabelle 5.4: Anzahl I_{total} an Intervallen für die ersten 10 000 *Pattern* der *Pattern Bank* nach einer Umsortierung.

eine Hamming-Distanz von sechs besitzen, wenn sie in allen Lagen unterschiedliche Sensor IDs beinhalten. Immer, wenn zwei in der *Pattern Bank* aufeinanderfolgende *Pattern* sich für eine Lage L unterscheiden, wird für diese Lage ein neues Intervall benötigt. Die Gesamtanzahl der Intervalle I_{total} für eine *Pattern Bank* mit N *Pattern* kann somit wie folgt definiert werden:

$$I_{total} = 6 + \sum_{i=1}^{N-1} D_H(n_i, n_{i+1}) \tag{5.5}$$

Wobei n_i ein beliebiges *Pattern* darstellt und n_{i+1} dessen direkten Nachfolger in der *Pattern Bank*. Die Lösung des *Traveling Salesman* Problems liefert eine Tour $T = (V, E'', c)$ und der Reisende kehrt zum Ende seiner Tour wieder in die erste Stadt zurück. Diese Tour muss für eine Lösung des Sortierungsproblems noch in einen Weg $W = (V, E', c)$ umgewandelt werden, was durch die Wegnahme der Kante $e \in E''$ mit den höchsten Kosten $c(e)$ realisiert werden kann. Mit dieser Transformation von Städten zu *Pattern* und Abstand zu Hamming-Distanz kann das Sortierungsproblem mit Hilfe der in Abschnitt 2.5 vorgestellten Algorithmen für das *Traveling Salesman* Problem gelöst werden.

5.2.2 Sortieren mittels Heuristiken

Für eine Umsortierung der *Pattern* werden die ersten 10 000 Einträge der *Pattern Bank* des *Barrel*-Sektors gewählt, da in Abschnitt 4.4 für diese Anzahl von *Pattern* ein überdurchschnittlich gutes Verhältnis von *Lookup*-Tabellen zu *Pattern* für ein einzelnes Design erzielt wurde. Wie

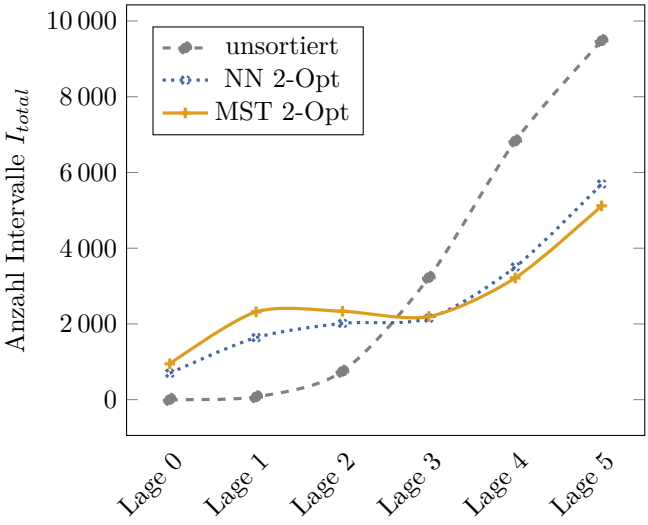


Abbildung 5.2: Anzahl der erreichten Intervalle pro Lage mittels verschiedener Algorithmen.

bereits in vorherigen Abschnitten erläutert und in Tabelle 5.3 dargestellt, werden für diese 10 000 *Pattern* in der ursprünglichen *Pattern Bank* insgesamt $I_{total} = 20\,360$ Intervalle gebildet und diese Anzahl kann durch eine einfache Sortierung mittels Permutation der Priorität der einzelnen Lagen nicht verringert werden. Da das Sortierungsproblem auf das *Traveling Salesman* Problem transferiert werden kann, können nun die in Abschnitt 2.5 vorgestellten Algorithmen mit minimalen Spannbäumen (MST) und mit Nächster-Nachbar-Suche NN eingesetzt werden. In Tabelle 5.4 sind die Ergebnisse dieser zwei Verfahren für jede Lage der sortierten *Pattern Bank* eingetragen. Zudem werden die Ergebnisse zusätzlich noch mit der 2-Opt-Heuristik optimiert. Eine Reduktion von bis zu $22,9\%$ der Intervalle I_{total} wird durch den Einsatz der Nächste-Nachbar-Suche mit 2-Opt-Heuristik (NN 2-Opt) erzielt. Auch die Sortierung mittels minimaler Spannbäume mit der 2-Opt-Heuristik reduziert die Gesamtanzahl der benötigten Intervalle signifikant.Vor allem die Verteilung der Intervalle über die Lagen hinweg wird deutlich ausgeglichen, dies wird durch Abbildung 5.2 visualisiert. In

Pattern Bank	Maximale Anzahl von Intervallen						
	Lage 0	Lage 1	Lage 2	Lage 3	Lage 4	Lage 5	gesamt
unsortiert	1	3	16	33	38	33	124
NN 2-Opt	260	237	84	25	17	17	640

Tabelle 5.5: Maximale Anzahl von erzeugten Intervalls pro Lage für die ersten 10 000 *Pattern* der *Pattern Bank*.

der ursprünglichen *Pattern Bank* werden für die innere Lage 0 nur drei Intervalle benötigt, im Gegensatz dazu jedoch 9 478 für die äußere Lage 5. Dies entspricht einer Standardabweichung von 3 959 Intervallen und führt zu einer deutlich unterschiedlichen Charakteristik der Vergleichseinheiten der unterschiedlichen Lagen. Eine Umsortierung der *Pattern* mittels minimaler Spannbäume und 2-Opt-Heuristik (MST 2-Opt) reduziert die Standardabweichung nahezu um den Faktor drei, auch alle anderen angewendeten Verfahren liefern eine deutliche ausgeglichenere Verteilung der Intervalle auf die einzelnen Lagen. In Tabelle 5.5 ist die maximale Anzahl von erzeugten Intervallen pro Lage dargestellt und die im schlimmsten Fall zu verarbeitende Anzahl von Intervallen für alle Lagen. Hier fallen für die sortiere *Pattern Bank* deutlich mehr Intervalle an, was zu einer größeren Latenz des Gesamtsystems führen kann, welche in Abschnitt 5.4 näher untersucht wird.

5.3 Modifizierung der Speicherstruktur

Die Grundstruktur der Speichereinheit mit ihren drei Blöcken Vergleichseinheit, Zwischenspeichereinheit und Entscheidungseinheit kann auch für die Verarbeitung mit Intervallen unverändert genutzt werden. In den folgenden Abschnitten werden vor allem die Vergleichseinheit und die Entscheidungseinheit modifiziert. Die Zwischeneinheit kann ohne große Veränderung übernommen werden, nur die Anzahl der benötigten FIFOs und die Wortbreite verändern sich.

5.3.1 Vergleichseinheit

Die Vergleichseinheit gibt anstelle von *Pattern*-Nummern die entsprechenden Intervalle weiter:

$$p_L : H_L \rightarrow I_L \tag{5.6}$$

Die Vergleichseinheit für eine Lage L liest einen *Hit* $h \in H_L$ ein und gibt die dazu passenden Intervalle $I_{(a,b)} \in I_L$ aus. Einem *Hit* $h_L \in H_L$ können , analog zu Gleichung 4.1, mehrere Intervalle zugeordnet werden, $h_l \mapsto \{I_{(a_1,b_1)}, I_{(a_2,b_2)}, \ldots, I_{(a_{hl},b_{hl})}\} = I_{L,hl}$ und die Menge $I_{L,h} \subset I_L$ beinhaltet alle Intervalle, die durch ein bestimmten Hit h in dieser Lage erfüllt werden. $I_{L,max}$ ist die Kardinalität der mächtigsten Menge $I_{L,h}$ für alle $h \in H_L$. Dies verringert die generierte und zu speichernde Datenmenge, da mehrere *Pattern*-Nummern zu einem Intervall gebündelt werden. Die Anzahl der benötigten FIFOs für eine Lage L wird analog zu Gleichung 4.2 ermittelt und ist abhängig von der Taktfrequenz f, der Latenz t und der Anzahl von möglichen Intervallen:

$$l_L = \lceil \frac{I_{L,max}}{t \cdot f} \rceil \tag{5.7}$$

Es werden also l_L verschiedene Intervalle zu einem Paket zusammengefasst und in einem Taktzyklus weitergegeben. Auch bei den Intervallen wird wie bei den einzelnen *Pattern*-Nummern eine strikte Ordnung eingehalten und es existiert keine *Pattern*-Nummer, welche über die Verbindung l_1 übertragen wird, die kleiner ist als die *Pattern*-Nummer, die über die Verbindung l_2 übertragen wird und so weiter. Die Vergleichseinheit wird hier unverändert als Zustandsautomat umgesetzt und jeder Zustand s generiert in diesem Fall eine Ansammlung von Intervallen. Somit werden

$$s_L = \lceil \frac{I_{L,max}}{l_L} \rceil \tag{5.8}$$

Zustände benötigt, um alle $I_{L,max}$ Intervalle zu erzeugen und weiterzugeben, wobei wieder $s \leq (t \cdot f)$ gilt. Dabei gilt $I_{L,max} \leq N_{L,max}$ und die Anzahl der benötigten Zustände s_L wird im Normalfall deutlich reduziert. Auch wenn die Anzahl der zu übertragenden Datensätze somit verringert wird, erhöht sich jedoch die Anzahl der Bits, die pro Eintrag benötigt werden, da ein Intervall nicht nur durch eine *Pattern*-Nummer beschrieben

werden kann. Ein Intervall $I_{a,b}$ kann sowohl durch sein Anfangs-*Pattern* a und End *Pattern* b beschrieben werden, als auch durch sein Anfangs-*Pattern* a und die Anzahl der im Intervall befindlichen *Pattern*. Für den ersten Fall verdoppelt sich die Anzahl der benötigten Bits, da anstelle von einer *Pattern*-Nummer, zwei *Pattern*-Nummern weitergegeben werden müssen. Für ein Design, welches k *Pattern* abdeckt, werden für die Repräsentation durch Anfangs- und End *Pattern* somit immer $(2 \cdot \lceil \log_2(k) \rceil)$ Bits als Wortbreite benötigt. Für eine Repräsentation durch das Anfangs-*Pattern* und Länge des Intervall muss das Intervall mit maximaler Länge bestimmt werden und diese Größe variiert je nach Beschaffenheit der *Pattern Bank* und kann sich auch für jede Lage L unterscheiden. Das Intervall $\hat{I}_L$ mit der maximalen Länge von i_L *Pattern* für eine Lage wird bei der Intervallbildung bestimmt:

$$\hat{I}_L = I_{a,b} :\Leftrightarrow \forall I_{x,y} \in I_L : ((a+b) \leq (x+y) \Rightarrow (x+y) \leq (a+b)) \quad (5.9)$$

Dieses Intervall $\hat{I}_L$ besitzt $i_L = |\hat{I}_L|$ *Pattern* und es werden maximal $p_{k,i_L} = (\lceil \log_2(k) \rceil + \lceil \log_2(i_L) \rceil)$ Bits benötigt, um ein Intervall zu repräsentieren. Im schlimmsten Fall ist $i_L = k$, falls es ein Intervall gibt, in dem alle *Pattern* liegen, dann werden für diese Lage $(2 \cdot \lceil \log_2(k) \rceil)$ benötigt, um ein Intervall zu speichern, wie es auch bei der Repräsentation mit Anfangs- und End *Pattern* der Fall ist. Im Normalfall liegt dieser Wert aber auch schon bei der ursprünglichen *Pattern Bank*, wie in Abbildung 5.1 dargestellt, deutlich unterhalb dieses Wertes. Nur für die innere Lage mit drei verschiedenen Sensor IDs und $i_L = 6\,368$ sukzessiv identischen Sensor IDs innerhalb der ersten 10 000 *Pattern* wird die Anzahl der benötigten Bits nur um eins reduziert. Durch die Umsortierung der *Pattern Bank* durch Heuristiken im vorherigen Abschnitt wurde die Anzahl der Intervalle zudem über die Lagen hinweg ausgeglichen und somit die durchschnittliche Länge eines Intervalls reduziert, was zu einer Einsparung der benötigten Bits für ein Intervall führt.

5.3.2 Entscheidungseinheit

Die Entscheidungseinheit hat im Wesentlichen zwei Aufgaben: Sie liest die in der Zwischenspeichereinheit gepufferten Intervalle aus und trifft anhand dieser die Entscheidung, ob ein *Pattern* in mindestens fünf der sechs Lagen

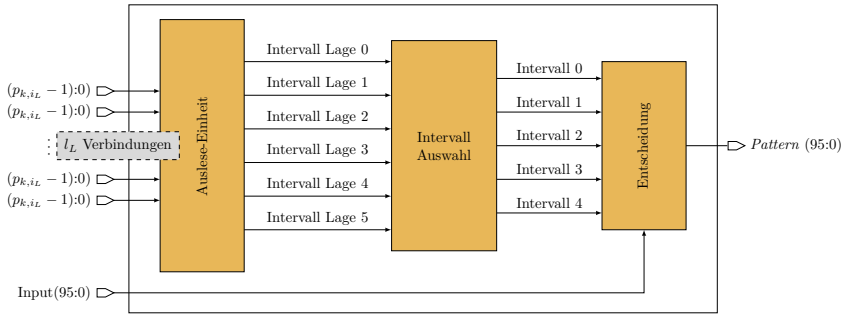


Abbildung 5.3: Struktur der Entscheidungseinheit für die Verarbeitung von Intervallen für ein Design mit k *Pattern*.

vorkommt. Diese Einheit besteht wiederum, wie in Abbildung 5.3 dargestellt, aus drei Blöcken. Die Ausleseeinheit gibt sechs relevante Intervalle an die Intervall-Auswahl weiter und entscheidet, welche Intervalle verworfen werden können. Von den sechs Intervallen werden fünf an die eigentliche Entscheidungseinheit weitergegeben und anhand derer werden dann die korrespondierenden *Pattern* ausgegeben. In den folgenden Abschnitten werden die einzelnen Einheiten und deren Aufgaben noch detailliert erläutert. Für die Verarbeitung der Intervalle durch die Entscheidungseinheit werden zunächst noch die Begriffe Übereinstimmung, Überlappung und Mehrheits-Überlappung definiert. Zwei Intervalle $I_{(a_0,b_0)}$ und $I_{(a_1,b_1)}$ stimmen in einem Intervall $I_{(a_1,b_0)}$ überein, wenn mindestens eine *Pattern*-Nummer in beiden Intervalle vorkommt, also $(a_0 \leq a_1 \leq b_0 \leq b_1)$ gilt. Analog dazu gibt es ein Intervall $I_{(a_0,b_1)}$ an übereinstimmenden *Pattern*-Nummern, wenn $(a_1 \leq a_0 \leq b_1 \leq b_0)$ gilt. In Abbildung 5.4 ist die Konstellation für ein übereinstimmendes Intervall $I_{(a_1,b_0)}$ dargestellt und es entsteht eine Übereinstimmung für das Intervall $I_{(15,45)}$. Im folgenden wird eine Übereinstimmung von mindestens fünf Intervallen in verschiedenen Lagen Überlappung genannt. Hierbei wird zwischen zwei Überlappungen unterschieden: Eine Mehrheits-Überlappung, wenn fünf Intervalle der sechs Lagen eine Übereinstimmung erzeugen, und eine normale Überlappung, wenn in allen sechs Lagen Intervalle übereinstimmen. In Abbildung 5.5 sind die parallel eingelesenen Intervalle aller sechs Lagen dargestellt, die noch folgenden Intervalle sind ausgegraut und werden im ersten Schritt

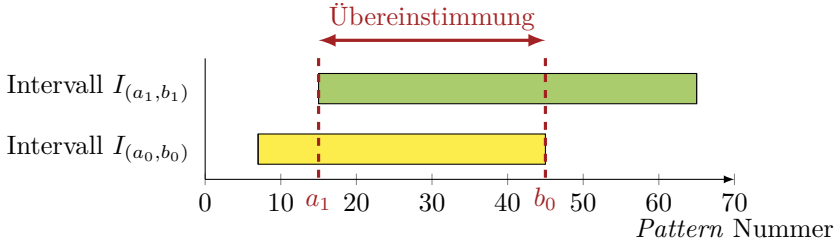


Abbildung 5.4: Übereinstimmung von Intervallen.

nicht weiter betrachtet. Es sind beide Arten von Überlappungen illustriert, es gibt eine Überlappung für das Intervall $I_{(10,20)}$ und eine Mehrheits-Überlappung für das $I_{(10,30)}$. In diesem Fall stimmen alle *Pattern* mit den Nummer 20 bis 30 mit den eingehenden *Stubs* überein und werden von der Entscheidungseinheit ausgegeben.

Auslese-Einheit

Diese Einheit ist direkt mit den FIFOs der Zwischenspeichereinheit verbunden und triggert das Einlesen neuer Intervalle. Für die Einhaltung der gesetzten Latenz ist es wichtig, irrelevante Intervalle so schnell wie möglich zu verwerfen und neue Intervalle zu laden. Ein Intervall ist nicht mehr relevant, wenn mit diesem Intervall keine Mehrheits-Überlappung mehr entstehen kann. Für jede Lage L wird parallel ein Intervall $I_{(a_L,b_L)}$ eingelesen. Für ein schnelles Aussortieren von Intervallen sind unter anderem die zwei Intervalle beliebiger Lagen X und Y mit den größten linken Endpunkten $I_{(a_{max},b_X)}$ und $I_{(a_{max2},b_Y)}$ von Bedeutung. Es gilt $(a_{max2} \leq a_{max})$ und $(a_L \leq a_{max2})$ für alle momentan geladenen Intervalle der übrigen Lagen. Um irrelevante Intervalle direkt verwerfen zu können, werden zwei Regeln aufgestellt:

(A) Ein Intervall $I_{(a_L,b_L)}$ wird verworfen, wenn sein rechter Endpunkt b_L kleiner oder gleich groß als der rechte Endpunkt aller anderen Intervalle ist $(b_L \leq b_{min})$.

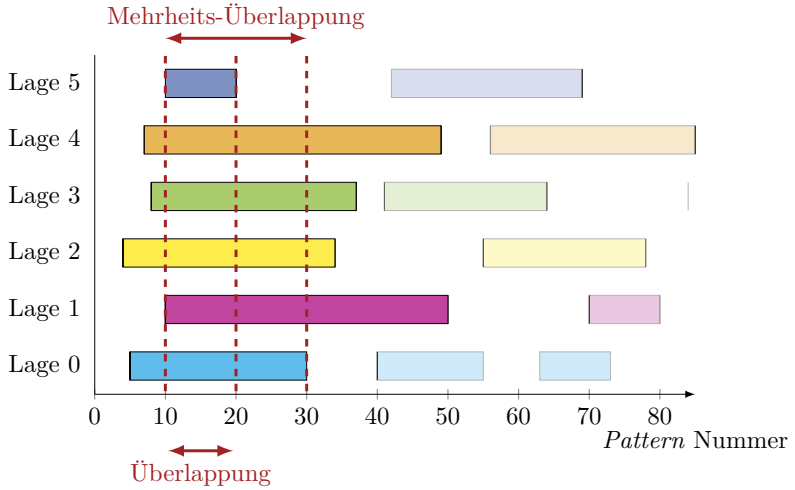


Abbildung 5.5: Eingelesene Intervalle und ihre möglichen Überlappungen.

(B) Ein Intervall $I_{(a_L,b_L)}$ wird verworfen, wenn es mindestens zwei andere Intervalle anderer Lagen gibt, bei denen der linke Endpunkt a größer ist als der rechte Endpunkt b_L ($b_L < a_{max2}$).

Diese Regeln werden nacheinander angewendet, nachdem die Intervalle an die Intervall Auswahl weitergeleitet wurden, welche dann überprüft, ob eine Mehrheits-Überlappung vorhanden ist. Durch die Regel *(A)* wird in jedem Schritt mindestens ein Intervall verworfen. In Abbildung 5.6 ist ein Szenario von eingelesenen Intervallen dargestellt, aufgrund der Regel *(A)* wird zunächst das Intervall I_{a_5,b_5} verworfen, da es den kleinsten rechten Endpunkt aller betrachteten Intervalle besitzt. Dieses Intervall wurde bereits an die Intervall Auswahl weiter geleitet und somit wurden alle Intervall-Überlappungen, die dieses Intervall I_{a_5,b_5} beinhalten, schon berechnet. Nach Regel *(B)* werden in diesem Zeitschritt keine Intervalle verworfen, da zu dieser Zeit kein Intervall einen rechten Endpunkt b_L besitzt, der kleiner ist als 10. Im nächsten Schritt wird das nächste Intervall der Lage 5 eingelesen, welches mit einer Schraffur markiert ist und alle Intervalle werden wieder an die Intervall Auswahl weiter geleitet. Nach Regel *(A)* wird nun das Intervall I_{a_2,b_2} verworfen und nach der Anwendung

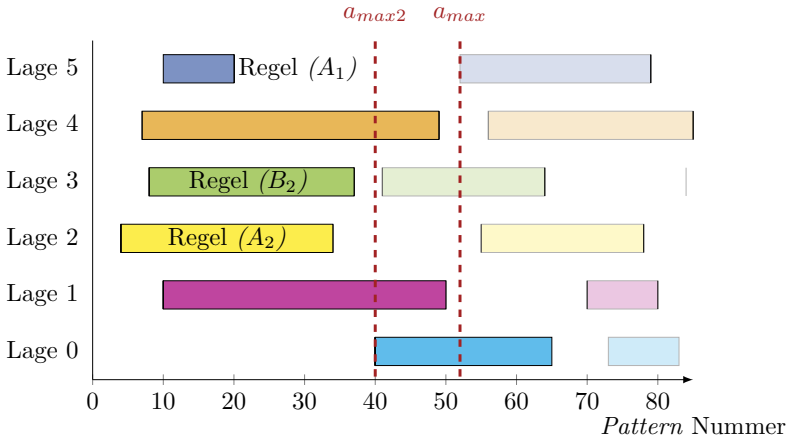


Abbildung 5.6: Angewendete Regeln zur schnellen Verwerfung von Intervallen.

von Regel *(B)* kann zusätzlich das Intervall I_{a_3,b_3} verworfen werden, da der rechten Endpunkte b_3 kleiner ist als die zwei linken Endpunkte a_0 und a_5. So können in einem Schritt mehrere Intervalle verworfen werden, ohne überlappende Intervalle zu übergehen. Maximal können alle Intervalle bis auf zwei verworfen werden, dann werden nur die beiden Intervalle mit den größten linken Endpunkten a_{max} und a_{max2} für die nächste Prüfung einer Überlappung übernommen und es werden neue Intervalle für die restlichen Lagen geladen.

Intervall Auswahl

Die Intervall Auswahl empfängt in jedem Schritt die aktuellen Intervalle jeder Lage und sortiert diese aufsteigend nach ihren linken Endpunkten a_L. Das Intervall I_{a_{max},b_L} mit dem maximalen linken Endpunkt a_{max} wird verworfen und nicht an die eigentliche Entscheidungseinheit weitergegeben. Im Normalfall wird dieses Intervall durch die Ausleseeinheit nicht verworfen, da nach Regel *(B)* nur Intervalle mit einem rechten Endpunkt $b_L < a_{max2}$ verworfen werden darf. Dieses Intervall wird somit im nächsten Zeitschritt erneut an die Intervall Auswahl weitergeleitet, falls ein weiteres

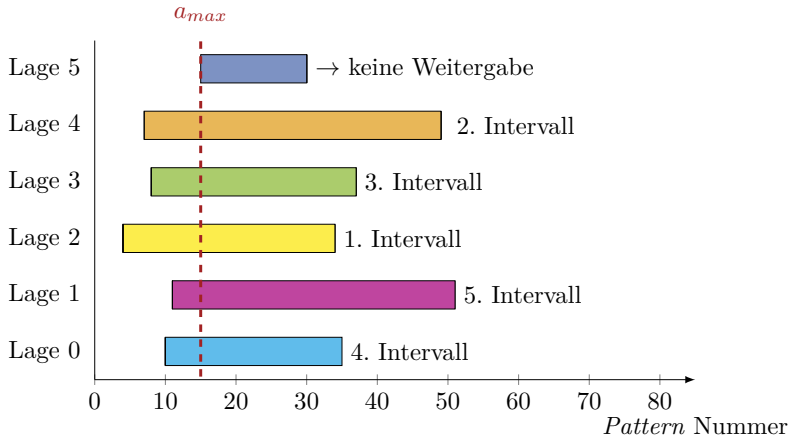


Abbildung 5.7: Ausgewählte Intervalle zur Weitergabe für ein Entscheidungsfindung.

Überlappungs-Intervall mit Hilfe dieses Intervalls gebildet werden kann. Nur für das Intervall $I_{(a_{max},b_{min})}$ kann der Fall eintreten, dass dieses Intervall nicht an die eigentliche Entscheidungseinheit weitergeleitet und es durch Regel *(A)* gelöscht wird. Dieses Szenario ist in Abbildung 5.7 dargestellt, in diesem Fall gilt jedoch für alle anderen Intervalle $I_{(a_L,b_L)}$ $(b_{min} \leq b_L)$ und $(a_L \leq a_{max})$. Dann gibt es ein Überlappungs-Intervall $[a_{max} : b_{min}]$, welches durch eine Mehrheits-Überlappung umschlossen wird und somit auch durch die Erkennung der Mehrheits-Überlappung durch die Entscheidungseinheit abgedeckt wird. In diesem Fall wird durch die Auslese-Einheit auch nur das Intervall $I_{(a_{max},b_{min})}$ verworfen und alle anderen Intervalle im nächsten Zeitschritt wieder der Intervall Auswahl weitergeleitet. Somit ist eine Reduzierung auf fünf Intervalle legitim und vereinfacht die Komplexität der Entscheidungsfindung.

Entscheidungs-Logik

Die Entscheidungseinheit bekommt fünf nach ihrem linken Endpunkt sortierte Intervalle $I_0 = [a_0 : b_0]$ bis $I_4 = [a_4 : b_4]$ weitergeleitet und überprüft, ob eine Übereinstimmung aller Intervalle für ein Intervall $I = [a, b]$

Implementierung	Anzahl LUT	Frequenz in MHz
FPGA Design ohne Intervalle	68 804	195.7
FPGA Design mit Intervallen - unsortiert	30 020	104.17
FPGA Design mit Intervallen - NN 2-Opt	30 939	84.03

Tabelle 5.6: Ergebnisse der Implementierung für verschiedene Ausführungen der Speicherarchitektur.

existiert. Das Intervall $I_4 = [a_4 : b_4]$ mit dem größten linken Endpunkt $a_4 = a_{max}$ und das Intervall $I_L = [a_L : b_L]$ mit dem kleinsten rechten Endpunkt $b_L = b_{min}$ bilden das Überlappungs-Intervall $I = [a_{max}, b_{min}]$, falls $(a_{max} \leq b_{min})$ gilt. Für das in Abbildung 5.7 dargestellte Szenario wird dementsprechend das Intervall $I = [11 : 34]$ gefunden und ausgegeben. Im nächsten Schritt werden in diesem Fall wieder die gleichen Intervalle durch die Anwendung der Regeln *(A)* und *(B)* durch die Intervall Auswahl an die Entscheidungseinheit weitergeleitet und somit kann selbst die Überlappung detektiert werden, wenn zweimal in Folge eine identische Mehrheits-Überlappung gefunden wird. Zudem werden auch sich überschneidende Überlappungs-Intervalle durch die Zwischenspeicherung des letzten ausgegebenen Intervalls ausfindig gemacht und somit verhindert, dass es *Pattern*-Nummern gibt, die mehrmals ausgegeben werden.

5.4 Evaluierung der modifizierten Speicherstruktur

Das modifizierte Design wurde für die Virtex 7 FPGA Familie des Herstellers Xilinx mit Hilfe des Tools Vivado 2016.4 implementiert. In Tabelle 5.6 sind die Ergebnisse der Implementierung für je 10 000 *Pattern* für ein Design ohne Intervallbildung, ein Design mit Intervallbildung und ein Design mit sortierter *Pattern Bank* mit Intervallbildung dargestellt. Die Einführung und Verarbeitung von Intervallen verringert die Anzahl benötigter *Lookup*-Tabelle drastisch um bis zu 55,03 %, die erreichte Frequenz wird durch die komplexere Vergleichseinheit um mehr als die Hälfte reduziert. Dies führt jedoch nicht zu einer Verdopplung der Latenz, sondern muss

Implementierung	Anzahl LUT	Anzahl Intervalle	Anzahl FIFOs	Länge FIFO	Intervalle maximal
unsortiert	30 020	20 360	9	20	124
unsortiert	29 110	20 360	6	38	124
NN 2-Opt	30 939	15 702	35	20	640
NN 2-Opt	26 768	15 702	6	260	640

Tabelle 5.7: Ergebnisse der Implementierung für verschiedene Ausführungen der Speicherarchitektur.

im Detail betrachtet werden, um überhaupt eine Aussage über die Latenz treffen zu können. Bei dem ursprünglichen Design konnte schnell ein *Worst-Case* Szenario durch die Anzahl der FIFO-Einträge erstellt werden, da im schlimmsten Fall alle FIFOs voll beschrieben, ausgelesen und die Einträge miteinander verglichen werden müssen. Bei der Verarbeitung von Intervallen ist dies nicht der Fall, hier hängt die Latenz des Systems stark mit der Beschaffenheit der Intervalle pro Lagen und deren Kombination zusammen. Wie in Abschnitt 5.2 erwähnt, gibt es bereits für die ersten 10 000 *Pattern* der *Pattern Bank* für die äußere Lagen 1 240 verschiedene Sensor IDs. Multipliziert man jede Anzahl von möglichen Sensor IDs der Lagen miteinander, erhält man über drei Billionen verschiedener Kombinationen von Eingabewerten, für die das System die passenden *Pattern*-Nummern ausgeben kann. Diese hohe Anzahl von Kombinationen verhindert ein komplettes Profiling des Systems, daher wird eine Näherung an ein *Worst-Case*-Szenario angestrebt. Das System kann den Abgleich der Eingabewerte mit der gespeicherten *Pattern Bank* beenden, sobald es in zwei Lagen keine zu berücksichtigen Intervalle mehr nachladen kann. Die maximale Anzahl von Intervallen, die erzeugt werden können, wurde in Abschnitt 5.2 auf Seite 95 berechnet und kann als Indiz für die Latenz des Systems dienen. Generell kann davon ausgegangen werden, dass die Anzahl der generierten und zu verarbeitenden Intervalle mit der benötigten Verarbeitungszeit korreliert, da pro Zeitschritt nur maximal in vier Lagen Intervalle verworfen werden können. Zusätzlich ist die Länge der FIFOs beziehungsweise die Anzahl der benötigten FIFOs ein weiterer wichtiger Faktor für die Bearbeitungszeit. In Tabelle 5.7 sind die Ergebnisse der Implementierung eines Designs mit der FIFO Länge von 20 Einträgen für eine unsortierte und eine sortiere *Pattern Bank* angegeben. Zudem ist jeweils

Implementierung	FIFOs	Vergleichseinheit	Zwischenspeichereinheit	Entscheidungseinheit
unsortiert	9	27 318	1 826	814
unsortiert	6	26 392	1 873	814
NN 2-Opt	35	24 532	5 583	824
NN 2-Opt	6	24 000	1 883	787

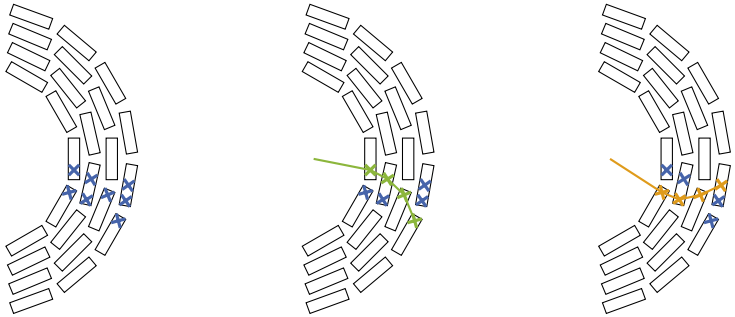
Tabelle 5.8: Detaillierte Ressourcenverteilung für ein System mit Intervallbildung für $k = 10\,000$ *Pattern*.

auch eine Variante implementiert worden, bei der die Länge der FIFOs auf die maximale Anzahl von möglichen Intervallen in einer Lage gesetzt wurde und somit die Anzahl der FIFOs pro Lage auf eins reduziert wird. In dem ersten Fall ist die Latenz geringer, da mehrere Intervalle pro Lage gleichzeitig verarbeitet werden können, was aber zu Lasten der benötigten *Lookup*-Tabellen geht, deren Bedarf im zweiten Fall geringer ist. Für alle Designs wurden als Eingabewerte, diejenigen Sensor IDs verwendet, die jeweils die meisten Intervalle pro Lage erzeugen. Für das originale Design mit der FIFO-Länge von 20 werden 15 Taktzyklen benötigt, um diese Eingabewerte zu bearbeiten und die entsprechenden *Pattern* ausgeben zu können. Das gleich aufgebaute Design mit der sortierten *Pattern Bank* benötigt hingegen 37 Taktzyklen, da im schlimmsten Fall deutlich mehr Intervalle miteinander verglichen werden müssen. Auch für das Design mit der jeweils maximal benötigten Anzahl von FIFO Einträgen ist die Verarbeitungszeit der sortierten *Pattern Bank* mit 19 Taktzyklen deutlich kürzer als für die sortierte *Pattern Bank* mit 61 Taktzyklen. Jedoch selbst bei 61 Taktzyklen und einer Taktfrequenz von 83,03 MHz beträgt die Latenz nur 0,7 μs, somit wird in allen Fällen die gesetzte Latenz von 2 μs ohne Probleme eingehalten. Im Mittel liegt die Anzahl der zu verarbeitenden Intervalle aber deutlich unterhalb der maximal zu erwartenden Anzahl und somit sinkt auch die zu erwartende Bearbeitungszeit. In Tabelle 5.8 ist die detaillierte Ressourcenverteilung der implementierten Systeme für eine Abdeckung von 10 000 *Pattern* eingetragen, auch hier werden mit Abstand die meisten *Lookup*-Tabellen für die Vergleichseinheit benötigt. Interessant hierbei ist, dass die Vergleichseinheit für die sortierte *Patten Bank* sich bei steigender Anzahl von gleichzeitig generierten Intervallen kaum verändert und die Zwischenspeichereinheit jedoch deutlich komplexer wird. Dies ist bei der sortierten *Patten Bank* nicht zu erkennen, wobei hier die Anzahl der FIFOs jedoch zwischen den zwei Systemen kaum

variiert. Zusammengefasst bietet die Bündelung von *Pattern* zu Intervallen vor allem eine enorme Einsparung an benötigten Logikeinheiten und verkürzt zusätzlich noch die Bearbeitungszeit, die benötigt wird, um die eingegebenen Sensor IDs mit der *Pattern Bank* abzugleichen.

6 Speicherstrukturen mit paralleler Mustererkennung

In den zwei voran gegangenen Kapiteln wurde eine FPGA Speicherstruktur entworfen, die eine grob-granulare Teilchenbahn bestehend aus einem *Stub* pro Lage empfängt und diese mit der gespeicherten *Pattern Bank* lagenweise abgleicht. Falls *Pattern* dieser *Pattern Bank* in fünf von sechs Lagen mit den eingelesenen *Stubs* übereinstimmen, werden diese ausgegeben. Wie in Unterabschnitt 3.1.2 beschrieben, werden für den *Level 1 Track Trigger*_ aber nicht nur sechs *Stubs* pro zu berechnenden Vergleich erwartet, sondern in der Spitze bis zu 300. In Abbildung 6.1a ist schematisch ein Teil des Detektors und mehrere *Stubs* pro Lage dargestellt. Diese *Stubs* sind von unterschiedlichen Spurbahnen von verschiedenen Teilchen erzeugt worden und nur anhand dieser *Stubs* kann nicht mehr auf die einzelnen Spurbahnen geschlossen werden. Somit kann jeder einzelne *Stub* einer Lage mit je einem *Stub* aus jeder anderen Lage zu einer möglichen Teilchenbahn kombiniert werden. In Abbildung 6.1b und Abbildung 6.1c sind beispielsweise zwei Spurbahnen eingezeichnet, die einen Teil der *Stubs* erzeugt haben könnten. Wenn nur die Information der *Stubs* zur Verfügung steht, gibt es jedoch keine Möglichkeit im Voraus die für die Spurbahnen A und B relevanten *Stubs* zu ermitteln und nur diese zwei Kombination mit der *Pattern Bank* zu vergleichen. Die einzuhaltende Latenz von $2\,\mu$s und diese hohe Anzahl von Kombinationen von *Stubs* schließt eine sequentielle Abarbeitung aller möglichen Kombinationen von *Stubs* durch das entworfene Design aus. Auch eine parallele Abarbeitung der Kombinationen durch mehrere Speicherstrukturen, die nebeneinander platziert sind, schließt sich durch den hohen Ressourcenbedarf aus. Eine Möglichkeit, die Logikeinheiten dennoch parallel nutzen zu können, stellt das Pipelining dar. In diesem Kapitel wird das in Kapitel 4 entworfene und in Kapitel 5 modifizierte Design durch eine Pipeline-Struktur erweitert.



(a) *Stubs*. **(b)** Teilchenspur A. **(c)** Teilchenspur B.

Abbildung 6.1: Mehrere *Stubs* pro Lage bilden in Kombination verschiedene Teilchenbahnen.

6.1 Erweiterung durch eine Pipeline-Struktur

Für ein Pipelining kann ausgenutzt werden, dass die Vergleichseinheit als Zustandsautomat realisiert wurde und jeder Zustand sequentiell die Nummern oder Intervalle der übereinstimmenden *Pattern* ausgibt (siehe Unterabschnitt 4.3.1). Die einzelnen Zustände können nun einen *Stub* einlesen, verarbeiten, im nächsten Schritt an den folgenden Zustand übergeben und dabei den nächsten *Stub* einlesen. Vor allem die Zwischenspeichereinheit muss jedoch für die Erweiterung einer Pipeline-Struktur modifiziert werden. Auch die Entscheidungseinheit muss eine größeren Anzahl von FIFOs und somit mehrere *Pattern*/Intervallen pro Lage parallel verarbeiten können. Für die Erweiterung dient das implementierte Design aus Kapitel 5 mit Intervallbildung der unsortierten *Pattern Bank* als Grundlage. Hierbei wird die FIFO Länge so gewählt, dass in der Grundstruktur nur ein FIFO pro Lage benötigt wird. Dieses Design bietet für die Erweiterung mittels Pipeline-Struktur die besten Voraussetzungen, da die Latenz von 19 Taktzyklen sehr gering ist, obwohl die Anzahl benötigter *Lookup*-Tabellen moderat ist. Der Aufbau der Speicherstruktur für die Verarbeitung mehrerer Stubs pro Lage ist in Abbildung 6.2 dargestellt und für jeden *Stub* wird ein FIFO in der entsprechenden Lage reserviert.

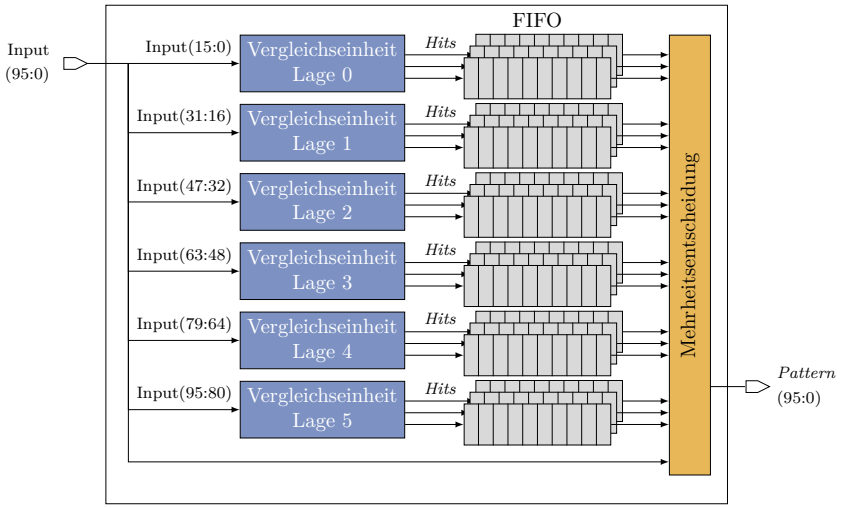


Abbildung 6.2: Aufbau der FPGA Speicherstruktur für die Verarbeitung mehrere *Stubs* pro Lage.

6.1.1 Vergleichseinheit

Die Vergleichseinheit liest pro Takt einen *Stub* ein und dieser passiert in jedem Taktschritt einen der insgesamt s_L Zustände. Die Anzahl s_L an Zuständen entspricht nach Gleichung 5.7 der maximalen Anzahl von verschiedenen Intervallen $I_{L,max}$ für eine Lage, da $l_L = 1$ gesetzt wurde. Dies bedeutet, dass in jedem Zustand genau ein Intervall an die Zwischenspeichereinheit übergeben wird. Prinzipiell können beliebig viele *Stubs* nacheinander eingelesen und bearbeitet werden, die maximale Anzahl von E_L *Stubs* ist nur durch die Anzahl von FIFOs der Zwischenspeichereinheit beschränkt. Jedes Intervall, welches durch den n-ten *Stub*, der die Vergleichseinheit passiert, ausgelöst wird, wird immer in den n-ten FIFO dieser Lage gespeichert. So werden nach und nach alle FIFOs der Lage innerhalb von $(E_L + s_L)$ Takten mit den entsprechenden Intervallen gefüllt. Die Wortbreite der FIFOs beträgt, analog zu der Speicherstruktur mit Intervallbildung, soviele Bits, wie benötigt werden, um ein Intervall zu repräsentieren. Für ein Design mit k *Pattern* und einer maximalen

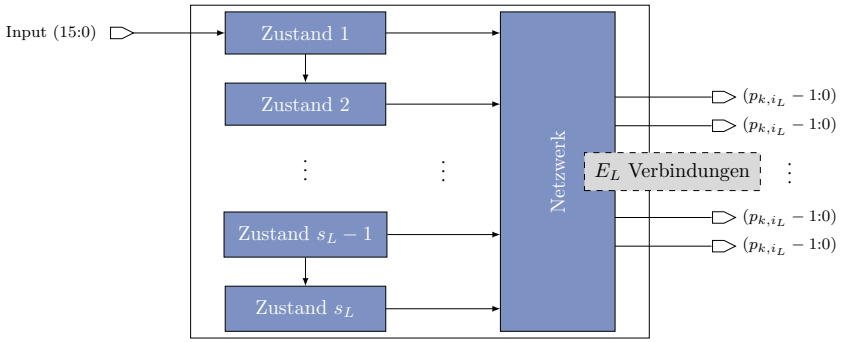


Abbildung 6.3: Struktur der Vergleichseinheit für eine Lage L mit Pipeline-Struktur für E zu erwartenden *Stubs*.

Intervalllänge von i_L *Pattern* sind dies $p_{k,i_L} = (\lceil \log_2(k) \rceil + \lceil \log_2(i_L) \rceil)$ Bits.

6.1.2 Zwischenspeichereinheit

Bei der Einführung einer Pipeline-Struktur erfährt die Zwischenspeichereinheit die größte Modifizierung. Es werden deutlich mehr FIFO-Einheiten benötigt, um die generierten *Pattern* aller eingelesenen *Stubs* speichern zu können. Zudem sollen die empfangenen Intervalle einer Lage direkt miteinander abgeglichen und gegebenenfalls durch die Zipper-Einheit vereinigt werden, damit die Entscheidungseinheit diese schneller auswerten kann. In Abbildung 6.4 ist die erweiterte Struktur der Zwischenspeichereinheit dargestellt. Die Anzahl der ersten FIFO-Instanz ist gleich der Anzahl von maximalen Stubs, die in dieser Lage eingelesen werden können. Wie bereits erwähnt, werden über alle Lagen verteilt bis zu 300 *Stubs* erwartet, diese verteilen sich aber nicht gleichmäßig auf die Lagen. Wie in Abschnitt 4.2 erläutert, befinden sich in den äußeren Lagen mehr Sensormodule als in den inneren. Auch wenn jedes Teilchen durch alle Lagen fliegt, werden in den äußeren Lagen mehr Sensormodule aktiviert, da es keinen Unterschied bei der Auswertung eines Moduls macht, ob es von mehr als einem Teilchen durchdrungen wurde. Zudem wird die Analyse der *Pattern Bank* verwendet, um die maximale Anzahl von zu erwartenden *Stubs* pro Lage abzuschätzen.

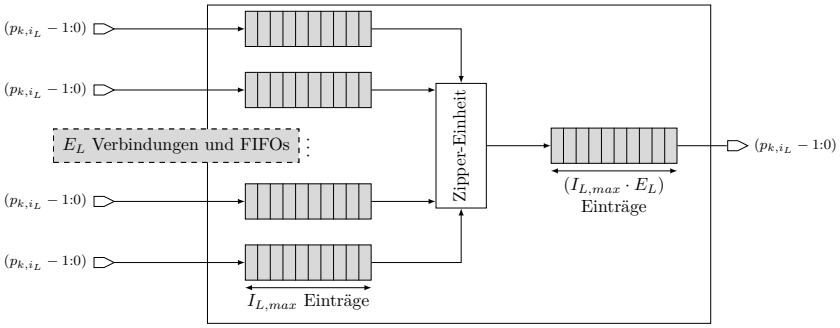


Abbildung 6.4: Struktur der Zwischenspeichereinheit für eine Lage L mit Pipeline-Struktur für E zu erwartenden *Stubs*.

In Abbildung 6.5 ist die Analyse der vorhandenen Sensorwerte und die maximale Anzahl von Intervallen $I_{L,max}$ für die ersten 10 000 *Pattern* der unsortierten *Pattern Bank* angegeben. Da in der inneren Lage nur drei verschiedene Sensor IDs vorkommen, können auch nicht mehr als drei *Stubs* für diese Lage, in diesem Teil der *Pattern Bank*, überhaupt *Pattern*-Nummern generieren. Es ist also ausreichend drei FIFOs für diese Lage vorzusehen, da $E_L = 3$ gilt. Da in dieser Lage jeder der drei Sensor IDs genau ein Intervall erzeugt, ist es zudem ausreichend, dass die nachfolgende FIFO Instanz eine Länge $I_{L,max} \cdot E_L = 3$ Einträgen besitzt. Dies stellt eine obere Schranke dar und durch eine weitere Analyse der *Pattern Bank* kann die Anzahl der benötigten FIFO-Einträge noch nach unten korrigiert werden. Beispielsweise kann für die Konstellation der möglichen Intervalle dieser Lage die Anzahl von drei Einträgen auf zwei reduziert werden. In dieser Lage existieren die Intervalle $I_{0,2161}$, $I_{2162,3632}$ und $I_{3633,99999}$ und wenn zwei direkt aufeinander folgende Intervalle generiert werden, können diese durch die Zipper-Einheit zu einem Intervall zusammengelegt werden. Im schlimmsten Fall werden in dieser Lage nur die beiden Intervalle $I_{0,2161}$ und $I_{3633,99999}$ generiert und müssen an die Entscheidungseinheit weitergegeben werden. Für diese Lage ist es nicht ressourcenaufwendig, den *Worst-Case* von drei *Stubs* abzudecken, auch für die nächste Lage mit maximal 36 *Stubs* und drei Intervallen pro *Stub* ist der Bedarf an Ressourcen noch ohne Probleme realisierbar. Für die äußeren Lagen ist die Anzahl von möglichen Sensor IDs jedoch deutlich höher und der nötige Ressourcenbedarf um alle

Lage	Verschiedene Sensorwerte	$I_{L,max}$ Intervalle
0	3	1
1	36	3
2	114	16
3	323	33
4	721	38
5	1 240	33

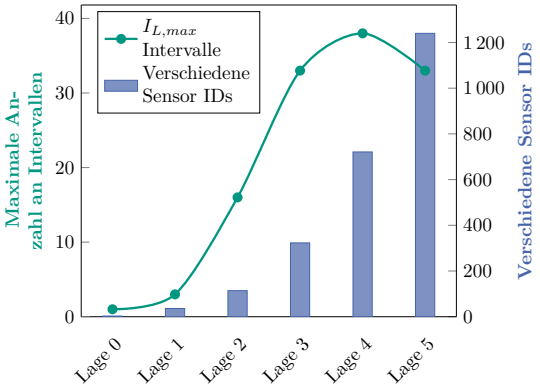


Abbildung 6.5: Analyse der ersten 10 000 *Pattern*: Abschätzung zu erwartende *Stubs*.

möglichen Sensorwerte gleichzeitig verarbeiten zu können, steht im keinen Verhältnis mehr zu der Wahrscheinlichkeit, dass dieser Fall überhaupt eintritt. Wenn insgesamt über alle Lagen verteilt bis zu 300 *Stubs* erwartet werden, würden in der äußeren Lage maximal 300 der maximal möglichen 1 240 Sensor IDs in den *Stubs* vorkommen. Da die *Stubs* aber über alle Lagen verteilt sind, ist die zu erwartende Anzahl von *Stubs* in dieser Lage noch weit unterhalb von 300. Als Abschätzung wurde hier die Verteilung der Sensor IDs über die Lagen herangezogen. Insgesamt existieren in den ersten 10 000 *Pattern* 2 437 verschiedene Sensor IDs verteilt auf alle Lagen. Ungefähr die Hälfte der Sensor IDs kommen hier in der äußeren Lage vor, auf die Anzahl der *Stubs* übertragen, ergibt dies eine zu erwartende Anzahl von 150 *Stubs*. Analog dazu werden in Lage 4 knapp ein Drittel der *Stubs* erwartet, was zu einer Anzahl von 90 *Stubs* führt. Diese Abschätzung reduziert die Anzahl der in der ersten FIFO-Instanz benötigten FIFOs enorm von 2437 auf $300 - 400$ und reduziert zudem die Komplexität der Zipper-Einheit. Diese Einheit liest in jedem Schritt die ersten Intervalle jeder Lage aus, verarbeitet diese nach einem Reißverschluss-Verfahren und verbindet sie entweder oder gibt sie nacheinander an die zweite FIFO Instanz weiter. Hierbei können in der zweiten Instanz auch mehrere FIFOs zur Verfügung gestellt werden, um die Entscheidungseinheit parallel mit

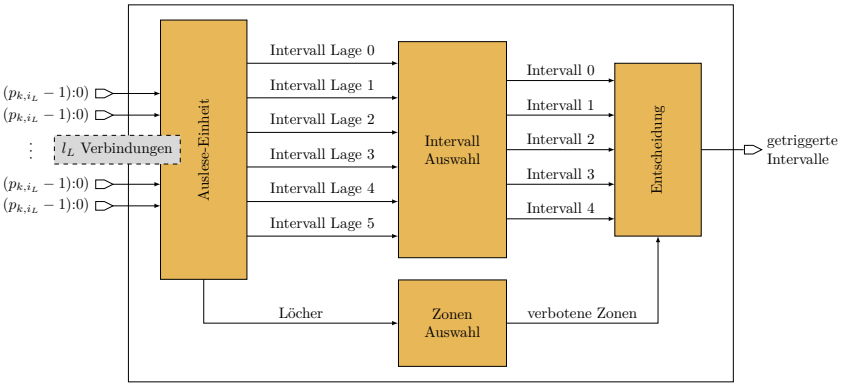


Abbildung 6.6: Struktur der Entscheidungseinheit für die Verarbeitung von mehreren Intervallen pro Lage für ein Design mit k *Pattern*.

mehreren Intervallen pro Lage versorgen zu können. Dies führt zu einer schnelleren Verarbeitungsmöglichkeit in dem letzten Vergleichsschrittes der entworfenem Speicherstruktur. Die Zipper-Einheit fängt an die Intervalle weiterzugeben, sobald für den letzten einzulesenden *Stub* der Lage das erste Intervall generiert und gespeichert wurde.

6.1.3 Entscheidungseinheit

Für die Entscheidungseinheit für das erweiterte Design mit der Pipeline-Struktur dient die in Unterabschnitt 5.3.2 beschriebene Einheit für die Intervall-Verarbeitung als Basis. Durch die gestiegene Anzahl von zu erwartenden Intervallen muss diese Einheit umstrukturiert werden, um die geforderte Latenz auch weiterhin einzuhalten. Durch den Einsatz der Zipper-Einheit werden die parallel durch mehrere *Stubs* erzeugten Intervalle zu einem oder mehreren Intervall-Strömen pro Lage gebündelt. In Abbildung 6.6 ist der strukturelle Aufbau der Entscheidungsmatrix abgebildet. Pro Lage sollen nun mehrere Intervalle gleichzeitig durch die Auslese-Einheit geladen und abgearbeitet werden. Weiterhin sind die Intervalle streng nach ihrem linken Endpunkt a sortiert. In Abbildung 6.7 ist ein Szenario von eingelesenen Intervallen dargestellt, wobei in jeder Lage zwei Intervalle parallel verarbeitet werden. Zusätzlich zu den Intervallen jeder

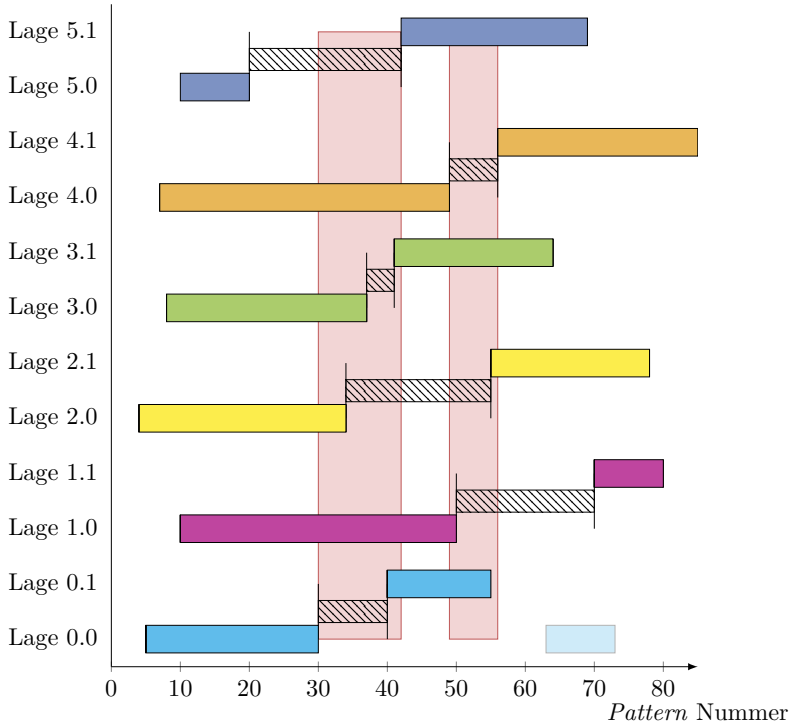


Abbildung 6.7: Eingelesene Intervalle bei zwei Intervallen pro Lage und die entstehenden Löcher (schraffiert) und verbotenen Zonen (rot).

Lage sind schraffiert die Lücken zwischen zwei eingelesenen Intervallen einer Lage eingezeichnet. Diese werden als Löcher bezeichnet und von der Auslese-Einheit an die Zonen-Auswahl weiter geleitet. Anhand dieser Löcher werden sogenannte verbotene Zonen definiert. Verbotene Zonen beschreiben einen Wertebereich, in dem es keine Mehrheits-Überlappung geben kann, da mindestens zwei Löcher überlappen. Diese Zonen sind in Abbildung 6.7 rot gekennzeichnet. Bei sechs Löchern können maximal fünf verbotene Zonen entstehen, die dann an die eigentliche Entscheidungseinheit weitergegeben werden. Die beiden durch die Auslese-Einheit eingelesenen Intervalle jeder Lage werden zu einem Intervall zusammengelegt und an die

Intervall-Auswahl gegeben. Diese ist identisch mit der Intervall-Auswahl der Entscheidungseinheit für die Intervall-Verarbeitung in Unterabschnitt 5.3.2 und liefert eine Mehrheits-Überlappung an die Entscheidungseinheit. Die Mehrheits-Überlappung der zusammengelegten Intervalle stellt nun eine mögliche Übereinstimmung der Sensor IDs mit der in der *Pattern Bank* gespeicherten *Pattern*. Die Berechnung der Mehrheitsüberlappung und der verbotenen Zonen geschieht parallel und die Ergebnisse werden dann in Kombination verwendet, um die tatsächlich in mindestens fünf der sechs Lagen aktivierten *Pattern* zu filtern. Hierzu werden die verbotenen Zonen von der Mehrheits-Überlappung abgezogen und es entstehen maximal sechs Intervalle von übereinstimmenden *Pattern*, die dann sukzessive ausgegeben werden.

6.2 Evaluierung

Auch bei diesem System werden die benötigten Ressourcen und die Latenz abgeschätzt, um eine Evaluierung durchführen zu können. Der Ressourcenverbrauch der Vergleichs- und Entscheidungseinheit wird sich im Vergleich zu dem System mit Intervallbildung kaum unterscheiden, da diese Einheiten teilweise identisch übernommen werden können. Nur bei der Entscheidungseinheit kommen mehr Signale und eine komplexere Netzwerk-Struktur aufgrund der höheren Anzahl von FIFOs in der Zwischenspeichereinheit hinzu. Zusätzlich muss diese Netzwerkstruktur jeden Zustand mit jedem der FIFOs verbinden können und je nach Bedarf die Intervalle zu den entsprechenden FIFOs leiten. Die Zwischenspeichereinheit wird durch die Pipeline-Struktur deutlich mehr Ressource benötigen, ein Indiz hierfür ist die steigende Anzahl von benötigten *Lookup*-Tabellen der Zwischenspeichereinheit für die Verarbeitung der sortierten *Patten Bank* in Tabelle 5.8 auf Seite 105. Für das Design mit 35 FIFOs werden dreimal so viele Ressourcen benötigt als für das Design mit sechs FIFOs, dementsprechend ist für die knapp 400 benötigten FIFOs für die Pipeline-Struktur ein enormer Anstieg der benötigten Ressourcen zu erwarten. Bisher wurden die FIFOs aufgrund ihrer Größe durch *Lookup*-Tabellen realisiert, dies ist für die große Anzahl von FIFOs nicht mehr praktikabel. Die FIFOs müssen daher auf den verfügbaren BRAM oder UltraRAM, welche in Unterabschnitt 2.4.1 beschrieben sind, platziert werden. Die neueren Virtex Ultrascale FPGAs

des Herstellers Xilinx bieten genug integrierten Speicherplatz, um auch diese hohe Anzahl von FIFOs im BRAM, beziehungsweise im UltraRAM, zu realisieren. Die Einführung der Pipeline-Struktur erhöht die Anzahl der benötigten Logikeinheiten insgesamt nicht kritisch, falls für die Zwischenspeichereinheit auf den intern verbauten Speicher zurück gegriffen werden kann. Auch die zu erwartende Taktfrequenz wird nicht signifikant unter der erreichten Taktfrequenz für das Design mit Intervallbildung liegen und kann für eine Latenzabschätzung als Grundlage dienen. Da die *Stubs* nacheinander eingelesen werden und dann die Intervalle generieren, kann die Zipper-Einheit der Zwischenspeichereinheit erst anfangen, wenn der letzte *Stubs* sein erstes Intervall gespeichert hat. Da in einer Lage bis zu E_L *Stubs* eingelesen werden, kann nach E_L Takten die Zipper-Einheit starten und gegebenenfalls Intervalle miteinander bündeln oder hintereinander ordnen. Frühestens, nachdem der letzte *Stub* das letzte Intervall an die Zwischenspeichereinheit weitergegeben hat, können die letzten Intervalle die Zipper-Einheit passieren, dies ist nach spätestens $(S_L + E_L)$ Takten der Fall. Als obere Schranke werden $I_{L,max}$ Einträge pro FIFO herangezogen, daraus resultieren maximal $(E_L \cdot I_{L,max})$ Intervalle, die zusammengeführt und sortiert an die Entscheidungsmehrheit weitergegeben werden. Im *Worst-Case* werden hierzu noch einmal $(E_L \cdot I_{L,max})$ Taktschritte benötigt. Somit werden maximal $(S_L + E_L + E_L \cdot I_{L,max})$ Takte benötigt, um alle *Stubs* einzulesen und die generierten Intervalle zu bündeln. Da für das System, wie in Unterabschnitt 6.1.1 beschrieben, $s_L = I_{L,max}$ gilt, ist somit die obere Schranke der Bearbeitungszeit bestimmt durch die maximale Anzahl von Intervallen $I_{L,max}$ in einer Lage L und durch die maximale Anzahl E_L von *Stubs*, die in dieser Lage verarbeitet werden können. Die Entscheidungseinheit kann frühestens anfangen die Intervalle aller Lagen miteinander zu vergleichen, wenn die Zipper-Einheit jeder Lage das erste Intervall weitergegeben hat. Nachdem die ersten Intervalle die Zipper-Einheit verlassen haben, kann die Entscheidungseinheit parallel zum restlichen System fortfahren und kontinuierlich alle Intervalle miteinander abgleichen. Ebenso wie bei dem Design mit Intervallbildung ohne die Pipeline-Struktur ist die Latenz sehr abhängig von der Beschaffenheit der zu vergleichenden Intervalle. Da im *Wort-Case* nach E_L Takten die Zipper-Einheit frühestens das erste Intervall liefern kann und erst nach $(I_{L,max} + E_L + E_L \cdot I_{L,max})$ Takten die Verarbeitung abgeschlossen hat, kann hierbei davon ausgegangen werden, dass alle Intervalle kontinuierlich direkt verglichen werden konnten. Bei einer großen Anzahl von *Stubs*, die

für die äußere Lage auf 150 begrenzt wurde, kann frühestens nach 150 Takten mit dem Vergleich begonnen werden, was bei einer Taktfrequenz von 80 MHz nahezu schon der gesamt zulässigen Latenz entspricht. Das System mit der Pipeline-Struktur kann die Latenz somit nur einhalten, wenn die Anzahl der einzulesenden *Stubs* weiter reduziert wird oder mehrere FIFOs pro *Stub* und pro Lage in Betracht gezogen werden. Wenn die Anzahl der *Stubs* in der äußeren Lage beispielsweise auf die Hälfte reduziert wird und pro Lage und *Stus* bis zu 20 FIFOs zur Verfügung gestellt werden, kann auch die herausfordernde Latenz von 2 μs eingehalten werden. Dafür werden jedoch wiederum deutlich mehr Logikeinheiten benötigt und somit entsteht ein Trade-off zwischen Latenz und der zur Verfügung stehenden Ressourcen.

7 Laufzeitadaptive Erweiterung von Mikroarchitekturen

In diesem Kapitel wird ein Konzept vorgestellt, welches ermöglicht den in Abschnitt 3.3 vorgestellten adaptiven Prozessor *i*-Core automatisch, transparent und dynamisch zur Laufzeit Hardwarebeschleuniger einzusetzen.

7.1 Konzept einer transparenten und dynamischen Hardwarebeschleunigung

Das Konzept des *i*-Cores basiert darauf, dass dem Entwickler Spezialinstruktionen zur Verfügung gestellt werden, welche dann durch Beschleuniger ausgeführt werden, die in der rekonfigurierbaren Einheit geladen sind. Falls für die Anwendung ein Beschleuniger zwar vorhanden ist, der Entwickler diese Spezialinstruktion aber nicht explizit nutzt, kann der Beschleuniger nicht verwendet werden. Im Rahmen dieser Arbeit wird ein Konzept entworfen, was dem *i*-Core ermöglicht eigenständig zu erkennen, ob eine Anwendung durch einen Hardwarebeschleuniger unterstützt werden kann [HSD+17]. In Abbildung 7.1 ist das Prinzip der automatisch generierten Spezialinstruktionen (Auto-SIs) und deren Ablauf dargestellt. Der linke Zweig beschreibt die normale Ausführung einer Anwendung mittels Software auf einem *General-Purpose* Prozessor und der mittlere stellt das Prinzip des *i*-Cores dar, der einen Beschleuniger zum Ausführen der Anwendung zur Verfügung hat. Durch den Einsatz des Auto-SI Ansatzes (rechter Zweig) soll nun die Anwendung während ihrer Durchführung beobachtet werden und, wenn möglich, ein Beschleuniger zur Verfügung gestellt und genutzt werden. Um eine transparente und dynamische Beschleunigung einer Anwendung zur Laufzeit realisieren zu können, sind folgende vier Schritte nötig:

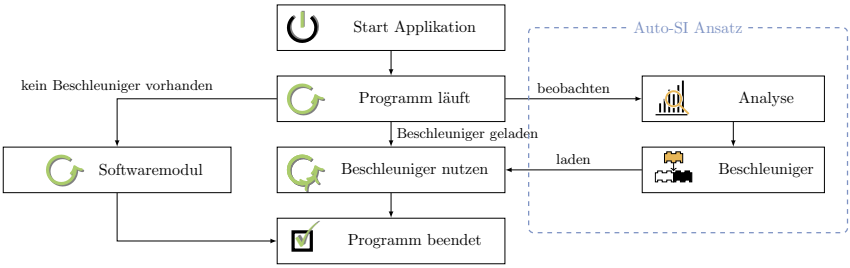


Abbildung 7.1: Prinzipieller Ablauf einer adaptiven Beschleunigung.

(A) Beobachten der laufenden Instruktionen

(B) Vorbereiten der Konfiguration

(C) Rekonfigurierbare Einheit programmieren

(D) Einsatz des Beschleunigers

Da die Analyse des gerade laufenden Programms und vor allem die Bereitstellung eines Beschleunigers einige Zeit in Anspruch nehmen, werden für den Einsatz eines Beschleunigers nur Schleifenkörper berücksichtigt. Für Instruktionen oder Programmabschnitte, die nur einmal ausgeführt werden, steht der Aufwand in keiner sinnvollen Relation zum möglichen Nutzen. Es besteht ein Trade-off zwischen der Anzahl von wiederholten Ausführungen der beschleunigten Instruktionen und dem Mehraufwand, der durch den Einsatz des Auto-SI Ansatzes entsteht. In den folgenden Abschnitten werden die vier einzelnen Schritte, die für die transparente und dynamische Beschleunigung zur Laufzeit nötig sind, detailliert beschrieben.

7.1.1 Beobachtung der laufenden Instruktionen

Für die Analyse der aktuell durchgeführten Instruktionen wird ein sogenannter Tracker in das Framework des *i*-Cores hinzugefügt. Dieser Tracker soll Schleifenkörper erkennen, die rechenintensive Instruktion beinhalten, welche durch den Einsatz eines Beschleunigers effizienter ausgeführt werden können. Diese Instruktionen werden im Folgenden Kernel genannt und dieser beinhaltet alle Instruktionen der Schleife, abgesehen von der für

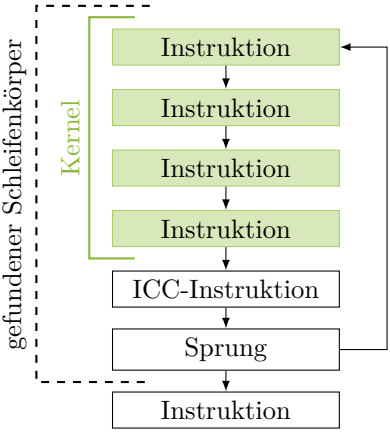


Abbildung 7.2: Schleifenkörper mit einem rechenintensivem Kernel, welcher sich für eine Beschleunigung eignet.

den Sprung verantwortliche *Integer Condition Code* (ICC) Instruktion und den Sprungbefehl. In Abbildung 7.2 ist ein Schleifenkörper mit einem solchen Kernel dargestellt. Wird ein Sprungbefehl erkannt und dessen Bedingungen erfüllt, speichert der Tracker den aktuellen *Program Counter* (PC) und zeichnet die folgenden Instruktionen auf. Bei einem erneuten Sprung wird der dann aktuelle PC mit dem letzten gespeicherten PC verglichen. Falls sie übereinstimmen, wurde ein Schleifenkörper gefunden. Bei einem negativen Vergleich wird der gespeicherte PC verworfen und der neue gespeichert und es wird abgewartet, bis der nächste Sprungbefehl erkannt wird. Wenn ein Schleifenkörper detektiert wurde und somit ein Kernel gefunden wurde, wird überprüft, ob sich dieser durch die rekonfigurierbare Einheit des *i*-Cores beschleunigen lässt. Es gibt diverse Instruktionen, die nicht unterstützt werden können, beispielsweise weil nicht auf alle Pipeline-Stufen oder Co-Prozessoren zugegriffen werden darf. In Abschnitt 7.3 werden die unterstützten Instruktionen detailliert betrachtet. Falls der Kernel Instruktionen beinhaltet, die nicht unterstützt werden, kann dieser Kernel nicht durch den Auto-SI Ansatz beschleunigt werden und wird verworfen. Ist jedoch ein Kernel erkannt worden, der auf der rekonfigurierbaren Einheit realisierbar ist, wird die mit der Vorbereitung der Konfiguration fortgefahren.

7.1.2 Vorbereiten der Konfiguration

Bevor ein Beschleuniger erstellt werden kann, muss der erkannte Kernel weiter analysiert werden, um sicherzustellen, dass alle Bedingungen erfüllt sind. Die wesentliche Einschränkung besteht hierbei in der Schnittstelle zwischen der CPU-Pipeline und der rekonfigurierbaren Einheit, in die der Beschleuniger geladen werden soll. Über diese Schnittstelle empfängt der Beschleuniger seine Eingangsdaten und übermittelt auch seine Ausgangsdaten. Ein Kernel, der mit dem Auto-SI-Ansatz beschleunigt wird, kann somit nicht mehr Ein- und Ausgänge beinhalten, als die Struktur des *i*-Cores zur Verfügung stellt. Zudem muss der Kernel durch einen Beschleuniger realisiert werden können, der mit den ihm zur Verfügung gestellten Ressourcen des FPGAs auskommt. Wenn der detektierte Kernel alle Voraussetzungen erfüllt, kann er durch einen Beschleuniger in der rekonfigurierbaren Einheit ausgeführt werden. Um dies zu überprüfen, wird für den erkannten Kernel ein Datenflussgraph (DFG) erstellt und die einzelnen Knoten (Instruktionen) dieses Graphen werden nach einer Reihenfolgeplanung Zeitintervallen zugeordnet. Durch den Einsatz einer *As Soon as possible* (ASAP) Strategie für die Reihenfolgeplanung kann der kritische Pfad für den Kernel bestimmt werden. Auf diese Weise können mehrere Knoten einem einzelnen Zeitfenster zugewiesen und eine Parallelverarbeitung auf Anweisungsebene (englisch *Instruction Level Parallelism* (ILP)) durchgeführt werden. Mit Hilfe des erstellen DFG kann abschließend der zu dem Kernel korrespondierende Beschleuniger gefunden werden. Es wird eine Bibliothek an verschiedenen Beschleunigern bereit gestellt, deren *Bitfile* und DFG in dem verfügbaren Speicher liegen. Falls ein passender DFG in der Bibliothek vorhanden ist, existiert ein Beschleuniger, der die Funktionalität des erkannten Kernels besitzt und in die rekonfigurierbare Einheit geladen werden kann.

7.1.3 Rekonfigurierbare Einheit programmieren

Sobald eine passende *Bitfile* für den aktuellen Kernel gefunden wurde, wird der Beschleuniger durch eine Rekonfigurierung des FPGAs geladen. Diese Rekonfiguration geschieht parallel zur laufenden Anwendung und stellt den zeitaufwendigsten Einzelschritt des Auto-SI-Ansatzes dar. Dieser Prozess kann mehrere tausend Taktzyklen (abhängig von der Taktfrequenz, Größe

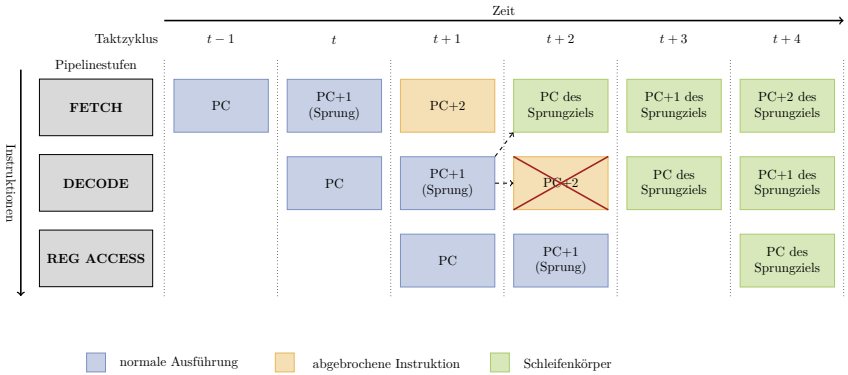


Abbildung 7.3: Instruktionsfluss durch die IU während ein Sprung auftritt.

des *Bitfiles* und der Rekonfigurationsbandbreite) in Anspruch nehmen und dieser Zeitaufwand wird in Abschnitt 7.4 im Detail betrachtet.

7.1.4 Einsatz des Beschleunigers

Sobald das *Bitfile* vollständig in die rekonfigurierbare Einheit des *i*-Cores geladen wurde und somit einsatzbereit ist, wird bei einem erneuten Auftreten des erkannten Kernels die Ausführung des Kernels an die rekonfigurierbare Einheit abgegeben. Dies erfordert eine Manipulation des Befehlsstroms, den die CPU-Pipeline verarbeitet, damit der nun verfügbare Hardwarebeschleuniger verwendet wird und nicht weiterhin die ursprünglichen Instruktionen des Kernels in Software abgearbeitet werden. In Abbildung 7.3 ist ein beispielhafter Instruktionsfluss dargestellt, wie er bei einem Sprungbefehl in der Pipeline verarbeitet wird. Ob ein konditionaler Sprung genommen wird, entscheidet sich in der *Decode*-Stufe, in der die ICC entsprechend des decodierten Sprungbefehls ausgewertet wird. Wenn nun ein erneutes Auftreten des bereits detektierten Kernels erkannt wird, wird in der *Decode*-Stufe nach dem erkannten Sprung eine Spezialinstruktion eingefügt und der PC wird auf die Sprungadresse gesetzt. Dies führt dazu, dass die Pipeline nicht komplett angehalten wird, sondern lediglich das Laden der folgenden Instruktionen verhindert wird. So werden Instruktionen, die sich bereits in der Pipeline befinden, zu Ende ausgeführt und deren Ablauf wird

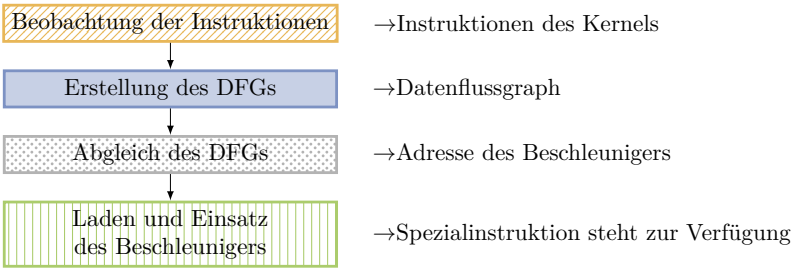


Abbildung 7.4: Auto-SI Ablauf und Zwischenergebnisse.

nicht beeinträchtigt. Die Eingangswerte der eingesetzten Spezialinstruktion werden direkt aus der Pipeline an die rekonfigurierbare Einheit übergeben und die Ausführung der Spezialinstruktion und deren Durchführung durch den Beschleuniger wird gestartet. Sobald die Berechnungen abgeschlossen sind, wird dies der Pipeline signalisiert, welche daraufhin die Ausgabedaten abholt und in die entsprechenden CPU-Register zurückschreibt. Der Programmablauf wird anschließend mit dem Sprung fortgesetzt und bei einer erneuten Ausführung des Kernels wird der geladene Beschleuniger direkt verwendet und die Spezialinstruktion erneut durch eine Manipulation des Befehlsstroms eingefügt. Falls der Kernel nicht erneut ausgeführt wird, wird die Beobachtung der laufenden Instruktionen fortgeführt und nach neuen Schleifenkörpern gesucht, die sich für eine Beschleunigung eignen.

7.2 Umsetzung des Auto-SI Konzeptes

In Abbildung 7.4 werden die wesentlichen Schritte, die für den Einsatz des Auto-SI Konzeptes benötigt werden, zusammengefasst und deren Resultat dargestellt. In diesem Abschnitt wird der Zustandsautomat beschrieben, der diese einzelnen Schritte kontrolliert. Der entworfene Zustandsautomat ist in Abbildung 7.5 dargestellt und die zu den Schritten in Abbildung 7.4 korrespondierenden Zustände sind mit der gleichen Farbe untermalt. Zunächst befindet sich der Auto-SI-Zustandsautomat im Standby-Modus (Zustand `LEERLAUF`) und wartet darauf, dass ein Sprungbefehl in der Pipeline auftritt. Sobald ein Sprungbefehl erkannt wird, wird der aktuelle PC gespeichert und der Zustandsautomat wartet (Zustand `BRANCH`), bis die

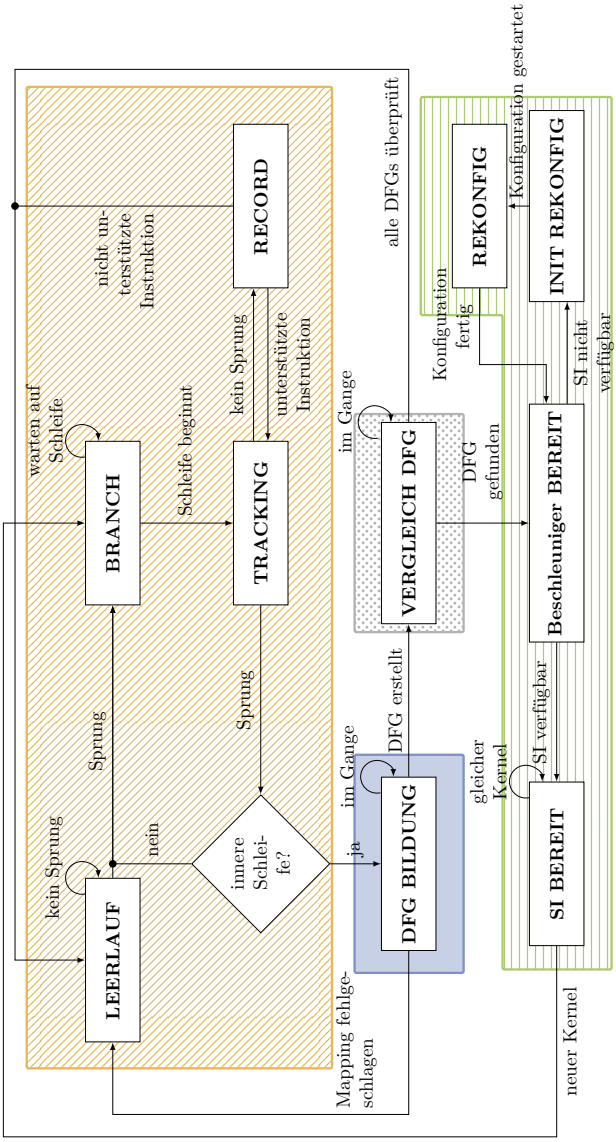


Abbildung 7.5: Prozess des Auto-SI Konzeptes.

Instruktion des Sprungzieles die Pipeline betreten hat. Dies ist notwendig, da es durch das Laden und Dekodieren von Instruktionen eine Verzögerung gibt, wie bereits in der Vorstellung des Konzeptes erwähnt und in Abbildung 7.3 veranschaulicht wurde. Sobald dies der Fall ist und ein möglicher Schleifenkörper beginnt, werden alle unterstützten Instruktionen des potentiellen Kernels verfolgt (Zustand `TRACKING`) und in dem sogenannten *Track-Cache* abgespeichert (Zustand `RECORD`). Falls bei dieser Überprüfung eine nicht unterstützte Instruktion auftritt, wird die Beobachtung des potentiellen Kernels abgebrochen und dieser wird verworfen, der Zustandsautomat kehrt dann in den Standby-Modus zurück und wartet auf den nächsten Sprungbefehl. Falls während der Überprüfung ein weiterer Sprungbefehl auftritt, der nicht dem letzten detektierten Sprung entspricht, wird der aktuelle Kernel ebenfalls verworfen und direkt der nächste potentielle Kernel betrachtet. Falls der neue Sprungbefehl jedoch mit dem letzten Sprungbefehl übereinstimmt, wurde eine innere Schleife gefunden und somit ein Kernel, der sich prinzipiell für eine Beschleunigung durch den Auto-SI Ansatz eignet. In diesem Falle wird angefangen, für diesen Kernel einen Datenflussgraphen zu erstellen (Zustand `DFG BILDUNG`). Dieser Vorgang wird abgebrochen, sobald der Kernel die gegebenen Limitierungen nicht mehr erfüllt. Bei einem Abbruch wird direkt in den Stndby-Modus gewechselt und auf neue Sprungbefehle gewartet. Wurde jedoch ein DFG erstellt, wird dieser dann im nächsten Schritt mit allen in der Bibliothek vorhandenen Datenflussgraphen verglichen (Zustand `VERGLEICH DFG`). Wenn ein passender Beschleuniger in der Bibliothek gefunden werden kann, wird zuerst geprüft, ob dieser bereits in der rekonfigurierbaren Einheit geladen ist und somit zur Verfügung steht (Zustand `Beschleuniger BEREIT`). Falls der Beschleuniger noch geladen werden muss, wird eine Rekonfigurierung angestoßen (Zustand `INIT REKONFIG`) und durchgeführt (Zustand `REKONFIG`). Sobald der Beschleuniger dann zur Verfügung steht (Zustand `SI Bereit`), kann dieser mittels einer Spezialinstruktion (SI) genutzt werden. Der Zustandsautomat kehrt dann wieder in den Standby-Modus zurück und bei erneuten Auftreten des Kernels wird dieser dann direkt durch die rekonfigurierbare Einheit in Hardware beschleunigt. In den folgenden Abschnitten werden die grundlegenden Schritte der einzelnen Zustände sowie die strukturelle Umsetzung und Integration in das Gesamtsystem detailliert erläutert.

Name	Verwendung
NO_FABRIC_IN_REGS	Anzahl der *Atom*-Eingangsregister
NO_FABRIC_OUT_REGS	Anzahl der *Atom*-Ausgangsregister
NO_FABRIC_IN_BITS*	Bitgröße der Anzahl Eingangsregister
NO_FABRIC_OUT_BITS*	Bitgröße der Anzahl Ausgangsregister
AUTO_SI_CACHE_SIZE	Maximale Größe *Track-Cache* und DFG
AUTO_SI_NODE_ID_SIZE*	Anzahl Bits für Kodierung einer Knoten ID
AUTO_SI_CMDQUEUE_ADDRESS	AHB-Addresse des *CmdQueue* Moduls
AUTO_SI_MEM_ADDRESS	Speicheraddresse der DFG-Bibliothek
AUTO_SI_NO_BITFILES	Anzahl *Bitfiles* / DFGs in Bibliothek
AUTO_SI_GROUP	*SI*-Gruppe für Auto-SI
AUTO_SI_OPCODE	*SI* ID für Auto-SI
AUTO_SI_ID*	AUTO_SI_GROUP und _ID
AUTO_SI_SSM_ENTRY	Eintrag im *SI State Memory*
AUTO_SI_MAX_DELAY	Maximale Anzahl Takte Verzögerung
AUTO_SI_NODE_SIZE*	Bitgröße eines DFG-Knotens

*wird aus anderen *Generics* erzeugt

Tabelle 7.1: Definierte *Generics* des Auto-SI Moduls.

7.2.1 Das Auto-SI Modul

Das Konzept der automatisch generierten Spezialinstruktion wird mittels eines optionalen Moduls in die Struktur des *i*-Cores, wie in Abbildung 7.6 dargestellt, integriert. Dieses Modul beinhaltet den im vorhergegangenen Abschnitt erläuterten Zustandsautomaten und zusätzlich eine AHB-Schnittstelle. Das Auto-SI Modul ist in drei Elemente aufgeteilt:

autosi_top Die *Top-Level-Entity* autosi_top wird in der *Integer Unit* (IU) instantiiert und stellt die AHB-Master-Schnittstelle zur Verfügung.

autosi_core Das Kernmodul autosi_core für Auto-SI beinhaltet den in Abbildung 7.5 dargestellten Zustandsautomaten. Es kontrolliert die Kommunikation mit dem *Advanced High-performance* Bus, veranlasst das Laden von *Atoms* und regelt die Benutzung des konfigurierten Beschleunigers.

autosi_lib Alle übergeordneten Konstanten, Typen- und Komponentendeklarationen werden in der *Library* definiert.

Bei der Implementierung wurde die von Jiri Gaisler in [52] vorgeschlagene *Two-process Design Method* verwendet, um kombinatorische von sequentieller Logik zu trennen und den VHDL-Code auf diese Weise sowohl

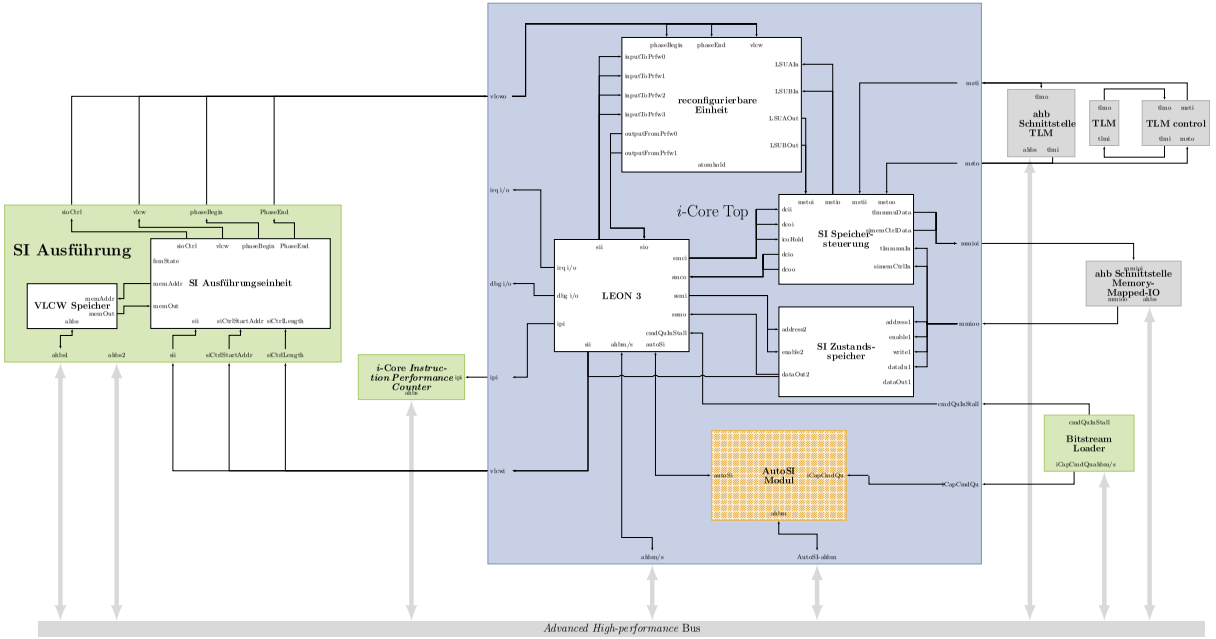


Abbildung 7.6: Struktureller Überblick der Einbindung des Moduls in den *i*-Core.

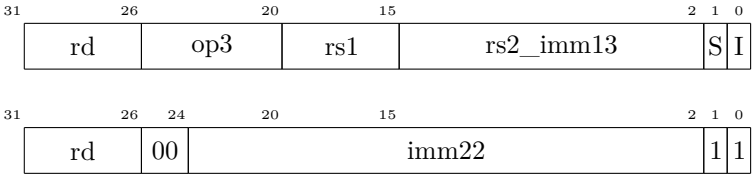


Abbildung 7.7: *Track-Cache* Struktur (S = SETHI, I = imm).

lesbar als auch wartbar zu halten. Bei der Implementierung wurde auf Skalierbarkeit geachtet und die Schnittstelle zur rekonfigurierbaren Einheit wurde aus diesem Grunde generisch gestaltet, ebenso wie die Größe des *Track-Caches*, in welchem die Instruktionen einer möglichen inneren Schleife speichert. Somit kann bei einer späteren Vergrößerung der Schnittstelle zwischen *Integer Unit* und der rekonfigurierbaren Einheit das Auto-SI Modul schnell durch eine Änderung der *Generics* angepasst werden. Die für die Implementierung gewählten *Generics* sind in Tabelle 7.1 aufgelistet.

7.2.2 Durchführung einer Auto-SI Beschleunigung

In diesem Abschnitt werden alle nötigen Schritte für die Umsetzung des Auto-SI Konzeptes detailliert erläutert. Nachdem durch die Beobachtung der aktuell durchgeführten Instruktionen eine innere Schleife gefunden wurde, erhält der Zustand `DFG BILDUNG` die Instruktionen des gefunden Kernels, welche in dem sogenannten *Track-Cache* abgespeichert wurden und gibt, falls möglich, den korrespondierenden Datenflussgraphen aus. Zudem werden die Erstellung der Datenflussgraphen durch den Zustand `DFG BILDUNG` und der Abgleich mit den in der Bibliothek befindlichen Datenflussgraphen durch den Zustand `VERGLEICH DFG` im Detail beschrieben.

Track-Cache

Der *Track-Cache* beinhaltet für jede Instruktion des Kernels die verwendeten Quell- und Zielregister, den Befehlscode und die Information, ob es sich um eine SETHI-Instruktion handelt oder eines der beiden Quellregister ein *Immediate* darstellt. Die Mehrzahl der unterstützten Instruktionen sind im

Format 3 der SPARCv8 Instruktionsformate (siehe Unterabschnitt 2.3.1) kodiert und aus diesem Grunde orientiert sich die Speicherstruktur des *Track-Caches* an diesem Format. Nur für die SETHI Instruktion und für Instruktionen mit *Immediates* wird ein zusätzliches *Flag* benötigt, auf diese Besonderheit wird in Abschnitt 7.3 detailliert eingegangen. In Abbildung 7.7 ist das Format angegeben, welches genutzt wird, um die nötigen Informationen einer Instruktion zu speichern. Mit Hilfe dieser Informationen kann ein korrespondierender Datenflussgraph für den Kernel erstellt werden.

Datenflussgraph und Erstellung

Die Knoten eines Datenflussgraphen stellen die einzelnen Instruktionen dar und die Kanten zwischen den Knoten beschreiben die ermittelten Datenabhängigkeiten dieser Instruktionen. Der DFG ist einfach gerichtet, dies bedeutet, dass in jedem Knoten nur dessen Vorgänger gespeichert werden und die Anzahl der Vorgänger ist durch den verwendeten Instruktionssatz auf zwei limitiert. Ein Knoten des zu generierenden Datenflussgraphen besteht aus 35 bit und beinhaltet neben den beiden Vorgängerknoten die Information, welches Eingabe- oder Ausgaberegister der rekonfigurierbaren Einheit genutzt wird, den Befehlscode der Instruktion und zusätzlich noch die Information, ob es sich um eine SETHI Instruktion handelt oder ein *Immediate* verwendet wird. Zudem wird noch die Information gespeichert, zu welchem Zeitpunkt diese Instruktion anhand eines ASAP Zeitplans platziert wurde. In Tabelle 7.2 sind diese Bestandteile des Knotens und deren Datentypen detailliert aufgeführt. Die Vorgängerknoten werden durch ihre Position im *Track-Cache* repräsentiert und AUTO_SI_CACHE_SIZE beschreibt die Anzahl der Instruktionen, die für einen Kernel gespeichert werden können. In diesem Fall wird die Anzahl der Instruktionen auf 100 reduziert. NO_FABRIC_IN_REGS und NO_FABRIC_OUT_REGS sind die Anzahl der Eingabe- beziehungsweise Ausgaberegister, die der rekonfigurierbaren Einheit zur Verfügung stehen und sind auf je zwei gesetzt. Ein Datenflussgraph besteht aus mehreren solcher Knoten und die Erstellung eines Datenflussgraphen ist schematisch in Abbildung 7.8 dargestellt. Die in dem *Track-Cache* gespeicherten Instruktionen des Kernels werden zu Beginn schrittweise durchgegangen und überprüft, auf welchen Eingangsdaten Instruktionen arbeiten. Dies ist wichtig, um entscheiden

Knoteneintrag	Typ	Bits
Vorgänger 1	natural range 0 to AUTO_SI_CACHE_SIZE	7
Vorgänger 2	natural range 0 to AUTO_SI_CACHE_SIZE	7
Befehlscode	op3_type	6
SETHI	std_logic	1
Eingangsregister 1	natural range 0 to NO_FABRIC_IN_REGS	2
Eingangsregister 2	natural range 0 to NO_FABRIC_IN_REGS	2
Immediate	std_logic	1
Ausgangsregister	natural range 0 to NO_FABRIC_OUT_REGS	2
Zeitpunkt	natural range 0 to AUTO_SI_CACHE_SIZE-1	7

Tabelle 7.2: Daten eines DFG-Knotens.

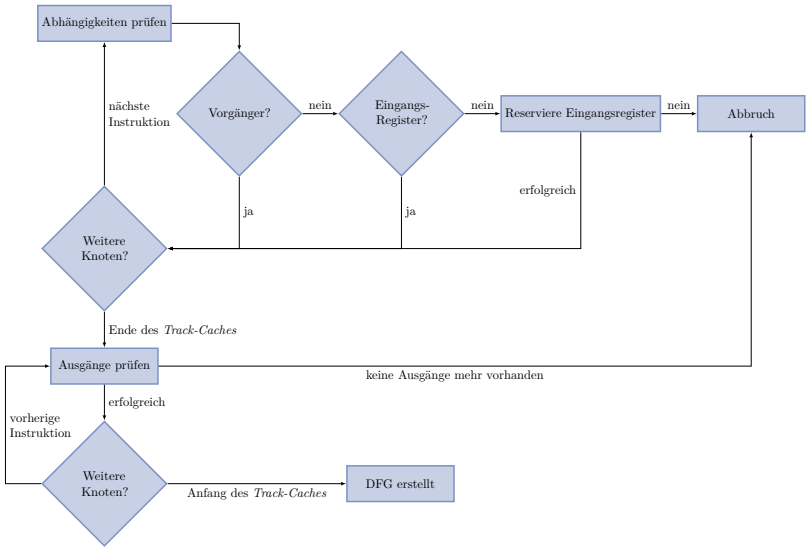


Abbildung 7.8: Ablauf der Erstellung eines Datenflussgraphen (DFGs).

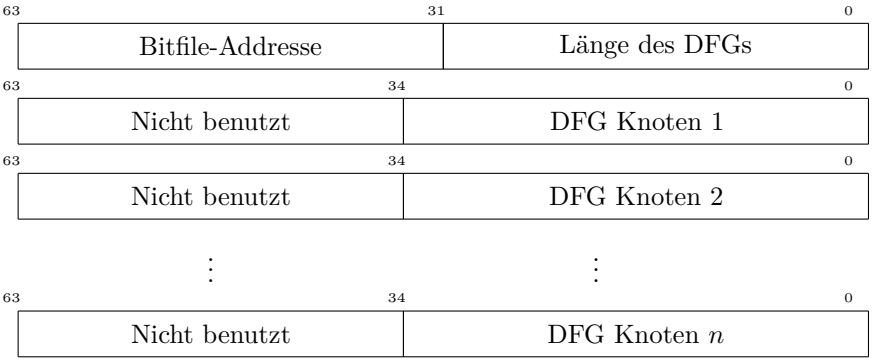


Abbildung 7.9: Datenstruktur eines DFGs in der Bibliothek.

zu können, welche Instruktionen Abhängigkeiten untereinander besitzen und wie viele Eingaberegister für den Kernel beansprucht werden. Für jedes Eingaberegister einer Instruktion des Kernels muss eine der folgenden Bedingungen gelten:

(A) Das Ausgaberegister einer bereits überprüften Instruktion stimmt mit den Eingaberegister überein.

(B) Eines der Eingangsregister der rekonfiguriergaren Einheit stimmt mit dem Eingaberegister überein.

(C) Es existieren noch freie Eingaberegister der rekonfigurierbaren Einheit.

Diese Bedingungen werden nacheinander abgeprüft und, falls keiner der drei Fälle zutrifft, wird die Erstellung des Datenflussgraphen abgebrochen, da der Kernel mehr als zwei Eingabewerte besitzt und somit nicht als ein *Atom* in der rekonfigurierbaren Einheit realisiert werden kann. Da die vorhergehenden Knoten immer von Anfang bis zum aktuellen Knoten durchlaufen werden und die Vorgängerknoten dabei überschrieben werden können, wird immer der zum aktuellen Knoten nächste Vorgänger gefunden. Eingangsregister der rekonfigurierbaren Einheit und auch *Immediates* können mehrfach als Eingaberegister einer Instruktion verwendet werden. Für die Eingangsregister der rekonfigurierbaren Einheit muss hierbei jedoch darauf geachtet werden, ob der Registerwert durch eine andere Instruktion bereits verändert wurde, dies wird durch die Ermittlung des

SI-Parameter	Typ	Verwendung
src_cnt	natural range 0 to 2	Anzahl verwendeter Eingangsregister
imm_cnt	natural range 0 to 2	Anzahl verwendeter Immediates
dest_cnt	natural range 0 to 2	Anzahl verwendeter Ausgangsregister
src	auto_si_in_imm_type	Eingangsregisteradressen und *Immediates*
imm	std_logic_vector(1 downto 0)	Kontrollbit *Immediate*-Verwendung
dest	auto_si_out_regs_type	Ausgangsregisteradressen

Tabelle 7.3: Auto-SI-Register.

nächsten Vorgängers sicher gestellt. Falls alle Instruktionen durchlaufen worden sind und die Eingaberegister anderen Ausgaberegistern oder den Eingaberegistern der rekonfigurierbaren Einheit zugeordnet werden konnten, werden alle Instruktionen noch einmal in umgekehrter Reihenfolge durchgegangen. Bei diesem Durchlauf werden die Ausgaberegister jeder Instruktion untersucht und ermittelt, ob die maximale Anzahl von zwei Ausgaberegistern für den Kernel eingehalten wird. Sobald beide Ausgaberegister der rekonfigurierbaren Einheit blockiert werden und eine weitere Instruktion Ausgabewerte liefert, die mit keinem anderen Eingaberegister mehr übereinstimmt, wird die Erstellung des Datenflussgraphen abgebrochen. Wenn für alle Instruktionen die Ausgaberegister zugeordnet werden konnten, wird der zum Kernel korrespondierende Datenflussgraph, welcher für jede Instruktion des Kernels einen Knoten mit zusätzlichen Informationen beinhaltet, generiert. In Abbildung 7.9 ist die Datenstruktur eines DFGs mit n Knoten gegeben. In den ersten 64 bits werden die Länge des Datenflussgraphen und die Adresse im Speicher, an der der eigentliche Beschleuniger liegt, kodiert. Zudem werden bei erfolgreicher DFG-Erstellung die in Tabelle 4.4 dargestellten Parameter der Spezialinstruktion ausgegeben. Diese werden später der Pipeline für die Konfiguration der Auto-SI zur Verfügung gestellt, falls ein passendes *Bitfile* zum aktuellen Kernel gefunden werden kann.

Vergleich des Datenflussgraphen mit der Bibliothek

Der erstellte Datenflussgraph wird mit einer Bibliothek von verschiedenen Datenflussgraphen, deren Beschleuniger zur Verfügung stehen, abgeglichen. Diese Beschleunigerbibliothek wird über die AHB-Master-Schnittstelle angesprochen und pro Takt können über diesen Bus 32 bits übertragen

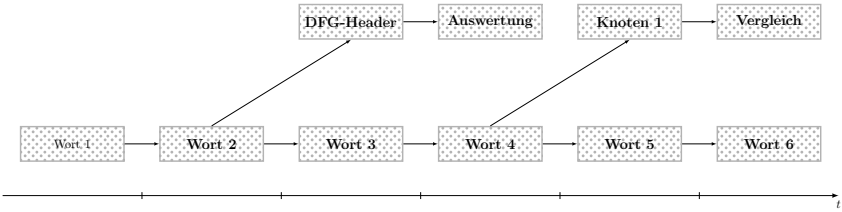


Abbildung 7.10: Laden und Vergleich von Datenflussgraphen (DFGs).

werden. Da ein Knoten des Datenflussgraphen jedoch aus 35 bits besteht, werden für das Laden eines solchen Knotens zwei AHB-Transfers benötigt. Dies verlangsamt den eigentlichen Vergleich jedoch nicht, da während des Wartetaktes auf den zweiten AHB-Transfer der Vergleich des zuletzt empfangenden DFG-Knotens stattfindet. Der Ablauf eines solchen Vergleiches ist in Abbildung 7.10 dargestellt. Die Knoten jedes DFGs werden innerhalb von zwei Takten aus einem Speicher gelesen und einzeln mit den Knoten des erzeugten Datenflussgraphen verglichen. Sobald ein abweichender Knoten gefunden wird, kann der Vergleich abgebrochen werden und der DFG des nächsten zur Verfügung stehenden Beschleunigers geladen und verglichen werden. Wenn der Vergleich bis zum letzten Knoten n erfolgreich durchläuft, wurde ein passender Datenflussgraph in der Bibliothek gefunden und der Kernel kann mittels eines zur Verfügung stehenden Beschleunigers ausgeführt werden.

Laden des Beschleunigers

Die zu den in der Bibliothek gespeicherten Datenflussgraphen gehörenden Beschleuniger *Bitfiles* sind in einem von der Bibliothek getrennten Bereich des Arbeitsspeichers abgelegt. Die Adressen der *Bitfiles* sind in den zugehörigen DFGs hinterlegt und wurden während des Vergleiches der Datenflussgraphen ausgelesen. Falls nun ein passender Datenflussgraph gefunden wurde, wird die Adresse des Speicherortes der *Bitfile* an die für die Rekonfigurierung des i-Cores zuständigem Module über den *Advanced High-performance* Bus weitergeleitet und der Rekonfigurationsvorgang wird gestartet. Nach Abschluss der Rekonfiguration wird dem

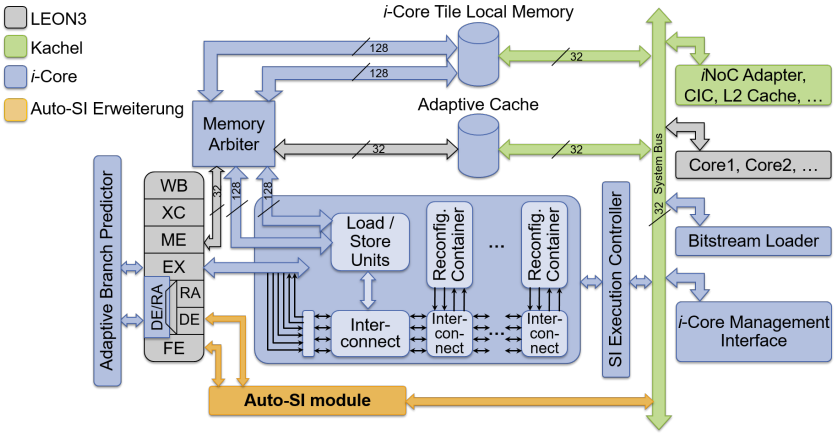


Abbildung 7.11: Der *i*-Core mit eingebetteten Auto-SI-Modul.

i-Core mitgeteilt, dass der Beschleuniger zur Verfügung steht und wie er angesprochen werden kann.

7.2.3 Integration in das Gesamtsystem

Zusätzlich zur Erstellung des Auto-SI Moduls sind weitere Änderungen an der bestehenden Implementierung des *i*-Cores notwendig, um die Funktionalität des Auto-SI-Ansatzes realisieren zu können. Vor allem die *Integer Unit* (IU) ist davon betroffen und das Auto-SI Modul benötigt eine direkte Anbindung an die verschiedenen Pipeline-Stufen. In Abbildung 7.11 ist die Einbettung des Auto-SI-Ansatzes in den *i*-Core und die Anbindung an die Pipeline schematisch dargestellt.

Modifizierung der Pipeline

Bei der Ausführung einer Auto-SI wird in der Fetch-Stufe der *Program Counter* (PC) für die Instruktion, welche dem beschleunigten Kernel nachfolgt, geschrieben. Dies ist in diesem Fall immer die Instruktion, die den *Integer Condition Code* (ICC) verändert, welches das Ende des inneren

Name	Typ
de_branch	std_ulogic
branch	std_ulogic
vfpc	pctype
de_inst	ssm_in_type
ssmo	std_logic_vector(18 downto 0)

Tabelle 7.4: Input-Schnittstelle.

Name	Typ
mapped	std_ulogic
start_si	std_ulogic
loop_end_pc	pctype
ssmi	word
si_regs	auto_si_register_type

Tabelle 7.5: Output-Schnittstelle.

Schleifenkörpers bildet. Bei einem Sprung werden sowohl der *Program Counter* (PC) des Sprungs als auch das Sprungziel und die Information, ob der Sprung ausgeführt wird, ausgegeben. Diese Daten werden für den Zustand `TRACKING` des Auto-SI Moduls benötigt. In der Decode-Stufe finden die wesentlichen Arbeitsschritte der Auto-SI-Realisierung statt. Zunächst wird die eigentliche Auto-SI erstellt und die Zielregister werden hinzugefügt, diese erstellte Instruktion wird dann in die Pipeline gespeist. Die noch fehlenden Instruktions-Informationen, wie beispielsweise Angaben zu Quellregister und der Verwendung von *Immediates*, werden aus den *SI*-Registern ausgelesen und ebenfalls an die Pipeline weitergegeben. Die restlichen Pipeline-Stufen können ohne Modifizierung übernommen werden, da die Auto-SI bereits kodiert ist und alle nötigen Information bereitgestellt wurden. Der weitere Ablauf einer Auto-SI entspricht dem normalen Ablauf einer Spezialinstruktion, nur wird die Ausführung einer Auto-SI durch ein *Flag* im Kontrollregister der Pipeline angezeigt.

Modifizierung weiterer Module

Wie in Abbildung 7.6 dargestellt, wird das Auto-SI Modul über drei Schnittstellen in den *i*-Core eingebunden. Die wichtigste Schnittstelle stellt hierbei die Anbindung an die *Integer Unit* (IU) des LEON3 dar, welche durch Ports definiert werden. Diese Ports werden zu je einem Input- und Output-*Record* zusammengefasst und sind in Tabelle 7.4 beziehungsweise in Tabelle 7.5 dargestellt.

Diese Ports werden durch die verschiedenen ineinander verschachtelten VHDL-*Entities* des *i*-Cores geleitet. Zusätzlich benötigt das Auto-SI Modul den Status des Rekonfigurationsprozesses und die Information, ob eine SI oder Auto-SI verwendet werden kann. Diese Informationen können über den AHB aus dem *CmdQueue* Modul des *i*-Cores geladen werden.

Hierzu muss das entsprechende Register immer wieder gepollt werden, was aber je nach Abfragehäufigkeit entweder den Bus stark belastet oder zusätzlichen Zeitbedarf für die Bereitstellung einer Auto-SI bedeutet. Um dies zu umgehen, werden die relevanten Bits direkt aus dem *ICAPE2* Modul, welches innerhalb des *CmdQueue* Moduls initiiert wird, an das Auto-SI Modul weitergeleitet. Die dritte Anbindung des Auto-SI Moduls, die Schnittstelle an den *Advanced High-performance* Bus, wird durch ein Interface geregelt und bedarf keiner weiteren Modifizierung des Systems.

7.3 Unterstützte Instruktionen

Welche Instruktionen sich durch das vorgestellte Auto-SI-Konzept beschleunigen lassen, ist vor allem beeinflusst, durch die Anbindung der rekonfigurierbaren Einheit des *i*-Cores an die *Integer Unit* des LEON3. Dies führt unter anderem zu Einschränkungen bezüglich der Ein- und Ausgangsregister, die auf zwei beschränkt sind. Wie bereits in Abschnitt 7.1 beschrieben, werden nur die Instruktionen einer inneren Schleife in Betracht gezogen und nicht die für den Sprung verantwortliche ICC Instruktion und den Sprungbefehl. Eine Änderung des ICC muss für die geforderte Transparenz des Auto-SI-Konzeptes an die Pipeline zurückgemeldet werden und würde somit bereits einen der beiden zur Verfügung stehenden Ausgangsregister beanspruchen. Da am Ende einer Schleife (und somit dem zu beschleunigenden Kernel) immer eine Instruktion vorkommt, die den ICC verändert, würde dies die Anzahl der realisierbaren Instruktionen drastisch einschränken. Aus diesem Grunde wird diese Instruktion nicht mit in den Kerlen übernommen und muss von der *Integer Unit* abgearbeitet werden. Dies hat zudem den Vorteil, dass der Schleifenindex nicht als Ausgabewert des Kernels übertragen werden muss. Des Weiteren werden alle Instruktionen nicht unterstützt, die Einfluss auf den *Program Counter* nehmen, da dieser nicht über die Schnittstelle der rekonfigurierbaren Einheit verändert werden kann. Analog dazu werden auch alle Instruktionen, die den *Window Pointer* verändern ausgeschlossen, da dieser nicht aus der rekonfigurierbaren Einheit heraus verändern werden kann. Auch kann über die rekonfigurierbare Einheit nicht auf die internen Register der *Integer Unit* zugegriffen werden und somit Load-/Store-Operationen auf dem *Alternate Space* nicht abgebildet werden, ebenso wie Instruktionen,

die das *Carry*-Bit des *Processor State Register*s (PSRs) verändern. Auch der Zugriff auf das Y-Register ist aus der rekonfigurierbaren Einheit nicht möglich und somit können Dividier- und Multiplizierinstruktionen, die dieses abfragen, nicht realisiert werden. Da kein *Trap*-Händler existiert, können Befehle, die *Traps* auslösen, nicht in die rekonfigurierbare Einheit verschoben werden. Sowohl auf die *Floating-Point Unit* als auch auf optionalen Coprozessoren kann nicht zugegriffen werden und diese stehen somit der rekonfigurierbaren Einheit nicht zur Verfügung. Nach Überprüfung der Voraussetzungen entsteht folgende Liste von unterstützten Instruktionen:

Speicher-Instruktionen: LDSB, LDSH, LDUB, LDUH, LD, LDD, STB, STH, ST, STD.

Logische Operationen: AND, ANDN, OR, ORN, XOR, XNOR.

Schiebe-Operationen: SLL, SRL, SRA.

Arithmetische Operationen: ADD, SUB, UMUL, SMUL, UDIV, SDIV.

Sonstige: SETHI.

7.3.1 Umgang mit Immediates

Immediates müssen nicht unbedingt als Eingabewerte berücksichtigt werden, da diese als Teil des Programmcodes unveränderlich sind. In einem Beschleuniger können sie beispielsweise fest verdrahtet werden, was jedoch unter Umständen dazu führt, dass ein Beschleuniger nicht mehr universell für einen Algorithmus einsetzbar ist. Hierbei entsteht ein Trade-Off zwischen der Allgemeingültigkeit eines Beschleunigers und den Eingangsregistern, die dem Beschleuniger zur Verfügung stehen. Aus diesem Grund wird für die Verarbeitung von *Immediates* ein *Flag* im Auto-SI-Register gesetzt, um auf diese Besonderheit eingehen zu können. Es gibt neben der Abbildung von *Immediates* auf Eingangsregister im Wesentlichen zwei Möglichkeiten, *Immediates* im Kontext von Auto-SI abzubilden:

(A) *Immediates* werden zur Laufzeit partiell in das *Bitfile* eingetragen oder, wenn möglich, über das VLCW kodiert.

(B) Für eine Auswahl von möglichen *Immediates* wird je ein passendes *Bitfile* und der dazu korrespondierende Datenflussgraph zur Verfügung gestellt.

Das realisierte Auto-SI-System kann sowohl *Immediates* als Eingaberegister abbilden, als auch mehrere Beschleuniger für unterschiedliche *Immediates* anbieten.

7.4 Evaluierung

Das vorgestellte Auto-SI-Konzept wurde für einen Virtex 7 FPGA des Herstellers Xilinx implementiert und in die bereits für diese Plattform vorhandene Implementierung des *i*-Cores eingebunden. In diesem Abschnitt wird das erstellte Design hinsichtlich der verwendeten Ressourcen und in Testsituationen erreichten Leistungssteigerungen evaluiert. Zudem wird eine Latenzanalyse durchgeführt, um eine Abschätzung der benötigten Zeit für die Bereitstellung eine Auto-SI zu erhalten.

7.4.1 Ressourcenbedarf

Der bereits existierende *i*-Core wurde, wie im vorhergehenden Abschnitt erläutert, um das Auto-SI Modul erweitert und einige kleinere Modifizierungen wurden an dem bestehenden System *i*-Core vorgenommen. Das Auto-SI Modul benötigt weniger als ein Prozent der verfügbaren *Lookup*-Tabellen des verwendeten Virtex 7 FPGAs. In Abbildung 7.12 ist der Ressourcenbedarf an Logikeinheiten für das Auto-SI Modul und den ursprünglichen *i*-Core dargestellt. Zusätzlich wird der Ressourcenverbrauch für den *i*-Core mit Auto-SI Unterstützung dargestellt. Verglichen mit dem ursprünglichen *i*-Core werden für den *i*-Core mit eingebetteten Auto-SI Modul 8,5 % mehr *Lookup*-Tabellen verwendet. Insgesamt werden 22 % mehr Ressourcen verwendet, um dem *i*-Core eine automatische und transparente Beschleunigung zur Laufzeit zu ermöglichen. Vor allem die Anzahl an benötigten Registern steigt signifikant um knapp 35 % an, da diese für den *Track-Cache* allokiert werden. Dieser kann aber, falls nötig, auch in den BRAM des FPGAs verschoben werden. Der Ressourcenbedarf ist weder an die Anzahl der verfügbaren Beschleuniger noch deren Größe gekoppelt, nur

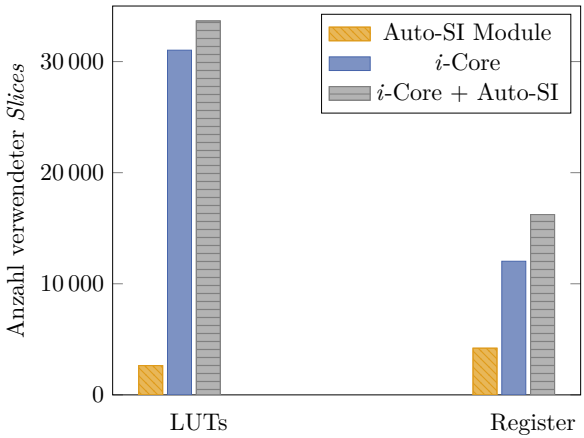


Abbildung 7.12: Ressourcenbedarf des Auto-SI Moduls im *i*-Core-Umfeld.

die maximale Länge eines Kernels ist durch den *Track-Cache* bei dieser Implementierung auf 100 Instruktionen beschränkt. Diese liegen in einem über den AHB angeschlossenen Speicher und dieser ist unabhängig von dem restlichen System. Die erzielte Taktfrequenz des *i*-Cores von etwa 75 MHz wird durch das Hinzufügen der Auto-SI Funktionalitäten nicht verändert.

7.4.2 Latenzanalyse

Im Folgenden wird zunächst die maximale Verzögerung, die durch die Ausführung einer Auto-SI Beschleunigung verursacht wird, analytisch abgeschätzt. Darauf basierend wird dann eine Evaluierung des Auto-SI Ansatzes durchgeführt, welche diese Verzögerung gegen eine mögliche Beschleunigung der Anwendung abwägt. In Tabelle 7.6 sind die verschiedenen Faktoren, welche für die Latenzanalyse berücksichtigt werden, aufgeführt. Die Wartezeit $t_{\text{RAM\_access}}$ ist sowohl von der Auslastung des *Advanced High-performance* Bus als auch von anderen nur zur Laufzeit bestimmbaren Zuständen abhängig und wird daher über durchgeführte Messungen abgeschätzt. Analog zu den in Abschnitt 7.1 herausgearbeiteten einzelnen

Faktor	Bezeichnung
Anzahl der Instruktionen des Kernels	n
Anzahl der *Bitfiles* (bzw. DFGs) in der Bibliothek	k
Länge der einzelnen DFGs der Bibliothek	L_i
Auslastung des AHB	$load_{\mathrm{AHB}}$
Wartetakte auf den Speicher	$t_{\mathrm{RAM\_Access}}$
Wartetakte auf den AHB	$t_{\mathrm{AHB\_wait}}$

Tabelle 7.6: Einflussfaktoren auf die Latenz.

Schritten des Auto-SI Konzeptes werden vier Einzelkomponenten für die Abschätzung der Latenz ermittelt:

(A) Beobachtung der laufenden Instruktionen D_{track}

(B) Erstellung des Datenflussgraph D_{DFG}

(C) Abgleich des DFGs mit der Bibliothek D_{match}

(D) Laden des Beschleunigers D_{reconfig}

Diese vier Ausführungszeiten bilden in Summe die Gesamtlatenz D und diese wird in Taktzyklen angegeben. Die Anzahl von Taktzyklen, die für die Beobachtung des aktuellen potentiellen Kernels benötigt wird, wird durch die Anzahl der in diesem Kernel befindlichen Instruktionen bestimmt:

$$D_{\mathrm{track}} = n \tag{7.1}$$

Ebenso die Ausführungszeit der Datenflussgraph-Erstellung steht in Abhängigkeit zu der Anzahl an Instruktionen des Kernels. Für die Erzeugung eines DFG wird zunächst einmal eine Initialisierung durchgeführt, dann für jede Instruktion die Eingänge überprüft und mit möglichen Vorgängern oder Eingangsregistern der rekonfigurierbaren Einheit verknüpft. Zudem werden die *Immediates* auf die Eingangsregister verteilt und anschließend die Outputs auf die Ausgangsregister der rekonfigurierbaren Einheit gesetzt. Für jede dieser Aktionen müssen alle im Kernel befindlichen Instruktionen

immer wieder durchlaufen werden und dies führt zu einer Abschätzung der benötigten Taktzyklen:

$$\begin{aligned} D_{\mathrm{DFG}} &= 1 + n + \sum_{i=1}^{n-1} i + NO\_FABRIC\_IN\_REGS * n + n + n + 1 \\ &= 2 + 5n + \sum_{k=1}^{n-1} k \\ &= \frac{1}{2} * (n^2 + 9n + 4) \end{aligned} \tag{7.2}$$

Hierbei wird die Anzahl der Eingangsregister der rekonfigurierbaren Einheit $NO\_FABRIC\_IN\_REGS$ auf zwei begrenzt und die Gleichung $\sum_{i=1}^{n-1} i = (n-1)\frac{n}{2}$ eingesetzt. Somit hat der Algorithmus für die Erstellung eines Datenflussgraphen einen quadratischen Laufzeitaufwand von $O(n^2)$ in Abhängigkeit zu der Anzahl von zu bearbeiteten Instruktionen n eines gefundenen Kernels. Die Ausführungszeit des Vergleiches des generierten Datenflussgraphen mit den in der Bibliothek zur Verfügung gestellten DFGs ist sowohl abhängig von der Länge des erstellten Datenflussgraphen als auch von der Anzahl k der zu vergleichenden DFGs. Für die Latenzanalyse wird davon ausgegangen, dass erst der letzte zu vergleichende Datenflussgraph zu einer Übereinstimmung führt und die maximale Anzahl von k DFGs verglichen werden muss. Zudem wird die Zugriffszeit des Speichers einkalkuliert und der DDR3-Arbeitsspeicher des verwendeten Virtex7-*Boards* kann maximal 8 Datenworte von 32 bit Länge in einem konsekutiven *Burst* übertragen, sodass jeweils nach 32 B eine weitere Wartezeit von t_{AHB_wait} hinzugefügt werden muss. Desweiteren können inkrementelle AHB-*Bursts* 1-kB-Adressbereiche nicht überschreiten [8]. Das AHB-Interface benutzt nach Möglichkeit die maximal zulässige inkrementelle *Burst*-Länge von acht Transfers. Somit sind 32 *Bursts* nötig, um 1 kB Daten zu übertragen und hierbei müssen 32 Wartezeiten in Betracht gezogen werden. Experimente liefern eine durchschnittliche Wartezeit $t_{\mathrm{RAM\_wait}}$ von etwa 40 Taktzyklen und somit fällt für eine Übertragung von 1 kB

$$\begin{aligned} t_{\mathrm{RAM\_wait\_KB}} &= 32 \cdot t_{\mathrm{RAM\_wait}} \\ &= 1280 \end{aligned}$$

Takten an Gesamt-Wartezeit an. Das Datenvolumen aller DFGs beträgt $2 \cdot \sum_{i=0}^{k} L_i \cdot 4$ Bytes. Die hierzu korrespondierende Latenz D_{match} für den Vergleich aller DFGs lautet:

$$\begin{aligned} D_{\text{match}} &= t_{\text{RAM\_Access}} + 2 * \sum_{i=0}^{k} L_i \\ &= \frac{size}{32\ Byte} \cdot t_{\text{RAM\_wait}} + 2 \cdot \sum_{i=0}^{k} L_i \\ &= 12 \cdot \sum_{i=0}^{k} L_i \end{aligned} \tag{7.3}$$

Die Latenz für die Rekonfiguration, also für das Laden des Beschleunigers, wird ebenfalls durch die Auslastung des AHB beeinflusst. Zudem ist sie abhängig von der Größe s_{bitfile} des zu ladenden *Bitfiles*, der Taktfrequenz f des laufenden Systems und der Übertragungsrate r:

$$D_{\text{reconfig}} = t_{\text{AHB\_wait}} + t_{\text{reconfig}} \tag{7.4}$$

Die Rekonfiguierung an sich korrespondiert mit $t_{\text{reconfig}} = \frac{s_{\text{bitfile}}}{r} \cdot f$ [40].

7.4.3 Experimentelle Ergebnisse

Um die Latenzanalyse des Auto-SI Konzeptes zu überprüfen, wurde ein Hardware-Prototyp erstellt, deren Bibliothek mit mehreren Kerneln bestehend aus einer variierenden Anzahl von sequentiellen ADD-Anweisungen ausgestattet ist. Um die Ausführungszeiten des Kernels bewerten und einen Geschwindigkeitsgewinn bestimmen zu können, wird ein *Speedup* definiert, der die Ausführungszeit des Kernels in Software in Relation zu der Ausführungszeit des korrespondierenden Beschleunigers setzt:

$$Speedup = \frac{t_{\text{Software}}}{t_{\text{Beschleuniger}}}$$

Der *Speedup* ist somit eine einheitenlose Größe, ist der Wert größer eins, wird durch den Einsatz des Beschleunigers ein Geschwindigkeitsgewinn

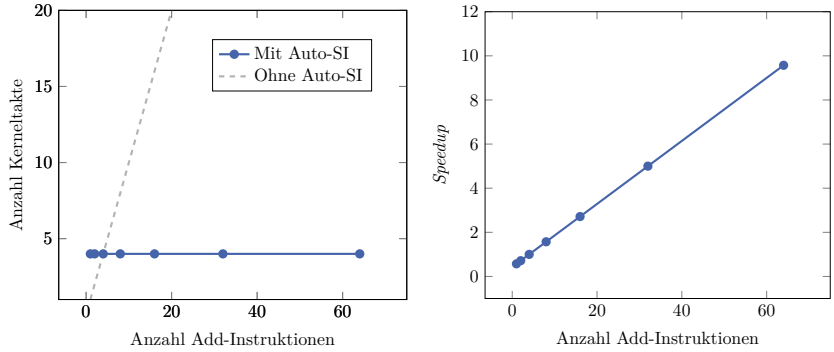


Abbildung 7.13: Beschleunigung von sequentiellen ADD-Instruktionen.

erzielt. In Abbildung 7.13 wird der maximal erzielbare *Speedup* für die Kernels in Abhängigkeit der Anzahl ihrer Instruktionen dargestellt. Je mehr Instruktionen in einem Kernel gebündelt werden können, desto höher fällt der maximal erzielbare *Speedup* aus. Die Anzahl der Instruktionen, die in einem Kernel zusammengefasst werden und in einem Beschleniger realisiert werden können, ist prinzipiell nur durch die Größe der rekonfigurierbaren Einheit des i-Cores beschränkt. Sobald ein Beschleuniger nicht mehr in einem *Atom* der rekonfigurierbaren Einheit realisiert werden kann, kann dieser beispielsweise auf zwei *Atoms* verteilt werden. Je nach Beschaffenheit des Beschleunigers kann dieser auch sequentiell mehrere Male hintereinander ausgeführt werden, dies ist der Fall bei den sequentiellen ADD-Instruktionen aus denen Testkernel erstellt wurden. Sobald durch die Beschränkung der Größe der rekonfigurierbaren Einheit auf die sequentielle Ausführung eines Beschleunigers zurückgegriffen werden muss, erhöht sich die Anzahl der für die Ausführung benötigten Takte stufenweise und der erreichbare *Speedup* wird bei der Einführung eines weiteren Taktes lokal verringert. Dieser Zusammenhang zwischen Anzahl der berücksichtigten Instruktionen und *Speedup* ist in Abbildung 7.14 dargestellt. Sobald nach der n-ten Instruktion die Größe eines *Atoms* überschritten wird, muss ein weiterer Takt für die erneute Benutzung des Beschleunigers eingeführt werden. Um genau diese Anzahl n_1,n_2,... von Instruktionen, bei denen ein erneuter Aufruf des Beschleunigers von Nöten ist, befindet sich ein lokales Minimum des *Speedups*. Generell steigt der zu erwartende *Speedup*

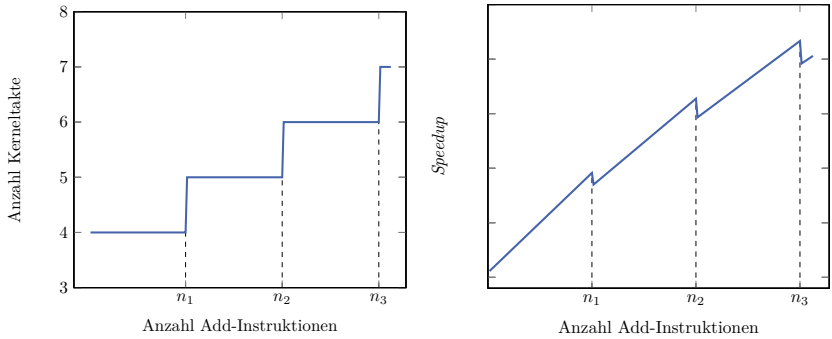


Abbildung 7.14: Einfluss der Anzahl an Instruktionen auf den zu erwartenden *Speedup*.

jedoch mit steigender Anzahl von Instruktionen und erreicht für 64 ADD Instruktionen einen Wert von 9.57. Der tatsächlich erzielbare Leistungsgewinn hängt nicht nur von dem erzielbaren *Speedup* eines Beschleunigers ab, sondern auch von der Anzahl der Iterationen des Kernels. Zudem muss auch der Mehraufwand der Rekonfigurierung in Betracht gezogen werden, falls kein passender Beschleuniger geladen ist. In Abbildung 7.15 sind die erzielten Ergebnisse für einen Kernel, der aus 64 sequentiellen ADD-Instruktionen besteht, dargestellt. Hierbei wurde kein Beschleuniger im Voraus geladen, zunächst wurden die laufenden Instruktionen beobachtet und parallel zur laufenden Anwendung ein Datenflussgraph erstellt. Dieser wurde dann, ebenfalls parallel zur laufenden Anwendung, mit den in der Bibliothek verfügbaren Datenflussgraphen verglichen und im Anschluss wurde der dazu korrespondierende Beschleuniger in die rekonfigurierbare Einheit des i-Cores geladen. Nach insgesamt 1 591 Iterationen des Kernels stand der Beschleuniger dann zur Verfügung und es konnte ein *Speedup* durch den Einsatz des Beschleunigers verzeichnet werden. Durch den Mehraufwand der Rekonfigurierung wird der theoretisch zu erreichende *Speedup* von 9.57 auf ungefähr 8 reduziert. Falls der korrespondierende Beschleuniger bereits geladen ist und der Kernel wieder ausgeführt wird, kann dieser nach Erstellung und Vergleich des Datenflussgraphen direkt genutzt werden. Dieses Szenario ist in Abbildung 7.16 dargestellt und nach 14 Iterationen des Kernels konnte der Beschleuniger genutzt werden. Der

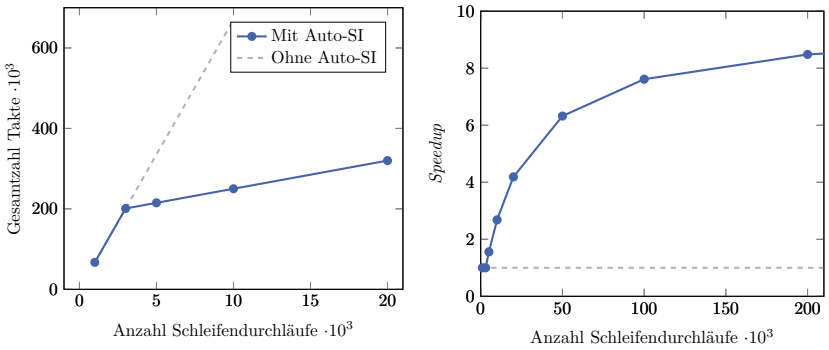


Abbildung 7.15: Erzielte Beschleunigung des Auto-SI Konzeptes für einen ADD64-Kernel ohne zuvor geladenen Beschleuniger.

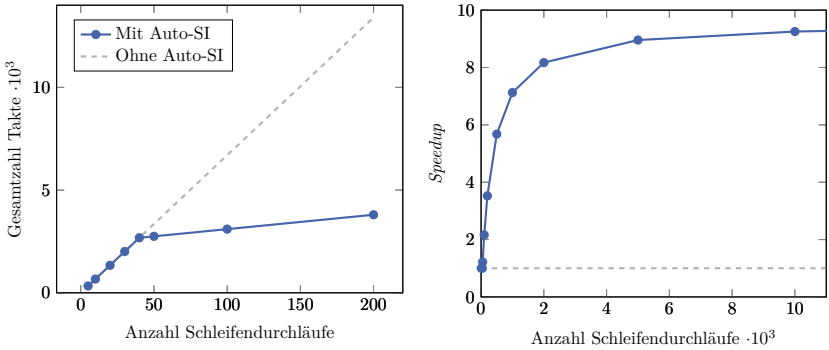


Abbildung 7.16: Erzielte Beschleunigung des Auto-SI Konzeptes für einen ADD64-Kernel mit bereits geladenen Beschleuniger.

theoretische *Speedup* des Kernels konnte in diesem Fall nahezu erreicht werden, da der Mehraufwand des Auto-SI Konzeptes hier gering ist.

7.4.4 Auswertung

In den letzten Abschnitten wurden sowohl die Latenz, der Ressourcenbedarf und der zu erreichende *Speedup* des vorgestellten und implementierten Auto-SI Konzeptes analysiert. Bei der Einbettung des Auto-SI-Moduls in den *i*-Core ist ein Mehraufwand an Ressourcen von 22 % entstanden. Eine Testapplikation erzielte im Experiment einen *Speedup* des Kernels von $9,5$, welcher in erste Linie nur durch die dem *Atom* zur Verfügung gestellten Ressourcen begrenzt ist. Vor allem für eine hohe Anzahl von Iterationen ist der Einsatz des Auto-SI Konzeptes sinnvoll und selbst eine Rekonfigurierung zur Laufzeit der Anwendung gewinnbringend. Auch für Kernel, die nicht oft ausgeführt werden, lohnt sich eine automatische Erkennung zur Laufzeit, da diese die normale Ausführungszeit nicht beeinflusst und bei einem bereits schon geladenen Beschleuniger dieser schnell gefunden und genutzt werden kann. Die Auto-SI Erweiterung beschleunigt die laufende Anwendung automatisch und transparent im besten Fall signifikant, führt jedoch nie zu einer längeren Ausführungszeit. Solange die benötigten Ressourcen zur Verfügung stehen, profitiert die laufende Anwendung von der Auto-SI Erweiterung des *i*-Cores. Der zu erzielende *Speedup* wird in den Experimenten vor allem durch die gegebenen Einschränkungen des *i*-Cores begrenzt, das entworfene Konzept der dynamischen und transparenten Beschleunigung ist jedoch nicht auf den *i*-Core begrenzt und nutzt diesen nur als Testumgebung. Das Konzept kann für jeden Prozessor angewendet werden, solange dieser einen Zugriff auf die Pipelinestufen zulässt und eine rekonfigurierbare Einheit für die Beschleuniger zur Verfügung stellt. Der dann zu erzielende *Speedup* ist im Wesentlichen durch die Anbindung an die Pipeline und die Größe der rekonfigurierbaren beschränkt. Die erzielten Ergebnisse zeigen, dass mittels Auto-SI eine erhebliche Beschleunigung von *Single-Thread*-Anwendungen erreicht werden kann, welche weit über die Grenzen dessen gehen, was durch aktuelle Architekturentwicklungen am Markt möglich ist.

8 Adaptive Beschleunigerstrukturen

Die Evaluierung des in Kapitel 7 entworfenem Auto-SI Konzeptes zeigt das Potential, welches in einer automatischen und dynamischen Beschleunigung zur Laufzeit liegt. Neben den Limitierungen des *i*-Cores wird der zu erzielende *Speedup* vor allem durch die Rekonfigurierung beschränkt. Aufgrund dessen wird in diesem Kapitel ein modularer Beschleuniger vorgestellt, welcher für unterschiedliche Anwendungen genutzt werden kann. Eine Spezialinstruktion des *i*-Cores ist bereits wie in Unterabschnitt 3.3.1 beschrieben, modular aus *Atoms* zusammengesetzt. Diese Struktur soll nun weiter bis in das einzelne *Atom* hineingezogen werden. Diese Erweiterung ist durch die begrenzte Anzahl von *Atoms*, die geleichzeitig zur Verfügung stehen, sinnvoll. Ebenso ist es erstrebenswert die maximal zur Verfügung stehenden Ressourcen eines *Atoms* auszunutzen, da die Größe eines *Atoms*, beziehungsweise eines *Bitfiles*, keinen signifikanten Einfluss auf die Ausführungszeit der Rekonfigurierung hat und die ungenutzten Ressourcen nicht für andere Zwecke verwendeten werden können. In Abbildung 8.1 ist die Grundstruktur eines solchen modularen Beschleunigers dargestellt, der zur Laufzeit durch das *Very Long Control Word* gesteuert werden soll. Hierbei wird die Struktur des modularen Aufbaus einer Spezialinstruktionen bestehend aus *Atoms* aufgegriffen und konsequent bis in die Hardwarestruktur eines einzelnen *Atoms* fortgesetzt. Die modularen Beschleuniger unterstützen somit den Auto-SI-Ansatz und ermöglichen einen flexibleren Einsatz der einzelnen Beschleuniger. In diesem Kapitel wird ein modularer Beschleuniger für die in Abschnitt 2.5 vorgestellten Algorithmen der Bildverarbeitung entworfen, in den *i*-Core mit Auto-SI Funktionalität integriert und abschließend evaluiert.

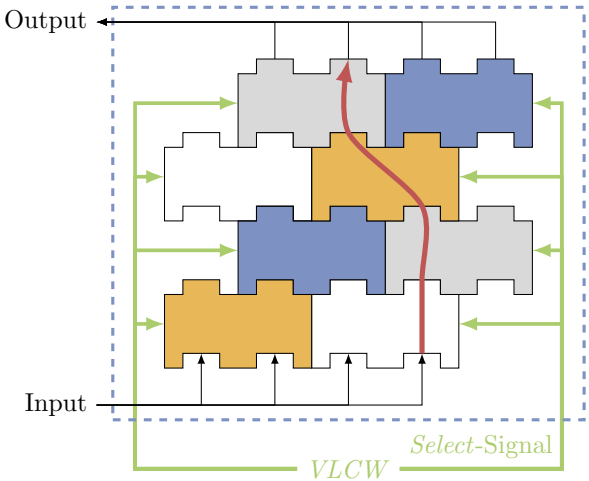


Abbildung 8.1: Modularer Aufbau eines Beschleunigers.

8.1 Modularer Beschleuniger für Algorithmen der Bildverarbeitung

Die in Abschnitt 2.5 vorgestellten Algorithmen des Gauß-, Sobel- und Laplace-Filters führen eine Faltung im Ortsbereich des Bildes durch und unterscheiden sich grundsätzlich nur in der zu verwendenden Faltungsmatrices f_G, S_x, S_y und Δ^2_{xy}. Jeder dieser Algorithmen erhält als Input neun Pixel $(a - i)$ eines Originalbildes A und berechnet mit diesen ein Pixel y des gefalteten Resultates:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \mapsto y \tag{8.1}$$

Das Eingangsbild ist im 8-Bit-Graustufen-Format kodiert und für jedes Pixel werden somit acht Bits zur Verfügung gestellt, mit denen insgesamt 256 verschiedene Grau-Abstufungen realisiert werden können. Da für den i-Core nur zwei 32 bit Eingänge zur Verfügung stehen, wird ein Interface für

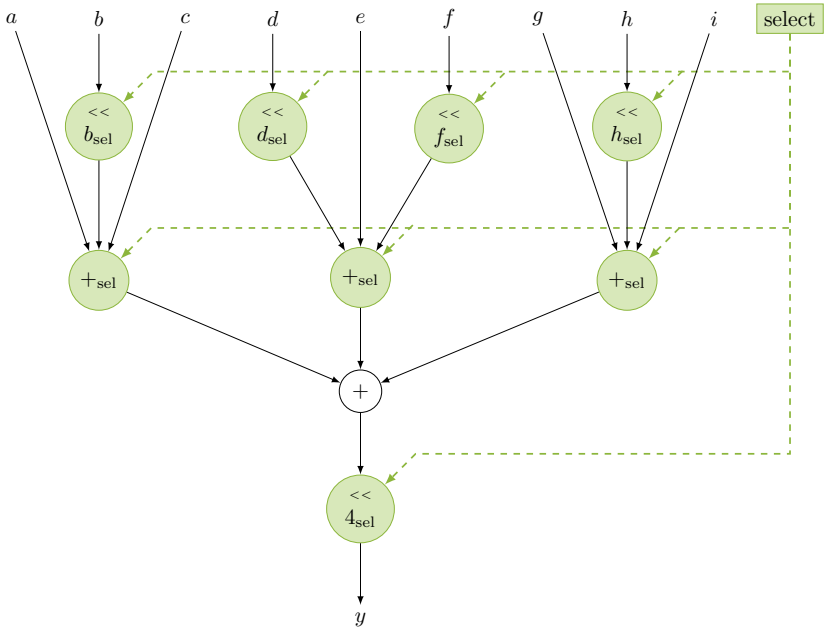


Abbildung 8.2: Datenpfad des modularen Beschleunigers für vier verschiedene Faltungsmatrices.

den modularen Beschleuniger vorgesehen, welcher in jedem Takt acht Pixel einliest und dann schrittweise neun Pixel in den eigentlichen Beschleuniger weiterleitet. In Unterabschnitt 8.1.1 wird dieses Interface beschrieben, in diesem Abschnitt wird der Aufbau des modularen Beschleunigers entworfen und davon ausgegangen, dass im jedem Takt neun Pixel mit je acht Bits empfangen werden können. In Abbildung 8.2 ist der Datenpfad eines modular aufgebauten Beschleunigers dargestellt, der die Faltungs-Operation

$M(\text{sel}) * A$ in Abhängigkeit eines *Select*-Signals sel für die vier verschiedenen Faltungsmatrices durchführen kann:

$$M(\text{sel}) = \begin{cases} f_G & \text{sel} = 00 \\ S_x & \text{sel} = 01 \\ S_y & \text{sel} = 10 \\ \Delta^2_{xy} & \text{sel} = 11 \end{cases} \tag{8.2}$$

Zu Beginn werden die neun Pixel $(a - i)$ eingelesen und, falls nötig, mittels einer *Shift*-Operation mit einer Zweierpotenz multipliziert. Dieser Vorgang wird durch das *Select*-Signal gesteuert und durch eine *Case* Anweisung entsprechend der Gleichung 8.2 implementiert. Beispielsweise wird für den Fall (sel = 00) oder (sel = 10), wenn eine Faltung mit der Gauß- oder der S_y Matrix vorliegt, das Signal b um eins nach links geschoben, sprich mit zwei multipliziert. In allen anderen Fällen wird der Wert des Signals b ohne Veränderung weitergegeben. Im nächsten Schritt werden je drei Signale zusammengefasst, auch dies geschieht wiederum in Abhängigkeit des *Select*-Signals. In diesem Schritt entscheidet jeweils das Vorzeichen der Elemente der verwendeten Faltungsmatrix, in welcher Form die Signale zusammengefasst werden. Für die Gaußmatrix (sel = 00) handelt es sich bei allen Operationen um eine Addition der drei Signalwerte, für die S_x Matrix (sel = 10) wird beispielsweise jeweils der dritte Signalwert vom ersten abgezogen. Nachdem die Zusammenfassung der drei Signale je nach Vorgabe vollzogen wurde, werden nun diese drei ermittelten Werte miteinander addiert, unabhängig von der durchzuführenden Faltung. Abschließend wird, falls eine Faltung mit der Gaußmatrix vorliegt, der Wert noch einmal durch 16 dividiert, indem er durch eine *Shift*-Operation um vier Stellen nach rechts verschoben wird.

8.1.1 Integration in den *i*-Core

Da dem *i*-Core nur zwei 32 bit Eingangsregister zur Verfügung stehen, können pro Takt nur acht anstelle von neun Pixeln übertragen werden, welche als Eingabe-Parameter für die im vorangegangenen Abschnitt entworfenen Beschleuniger benötigt werden. Daher muss ein Interface entworfen werden, dass pro Takt acht Pixel mit je acht Bit einliest, zwischenspeichert und dem Beschleuniger dann pro Takt neun Pixel weiterleitet. Um die empfangenen

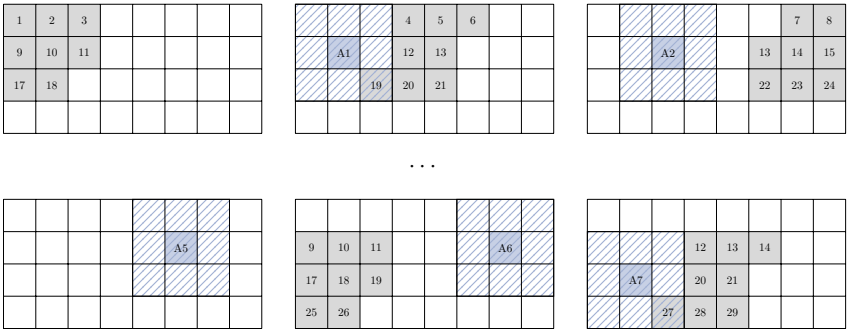


Abbildung 8.3: Lade-Zyklen des Interfaces für ein Bild mit (8×4) Pixeln.

Pixel zwischenzuspeichern wird ein Speicherfeld von (8×3) Einträgen instanziiert, welches innerhalb von drei Takten komplett gefüllt werden kann. In Abbildung 8.3 ist der Ablauf skizziert, in welchem der Speicher gefüllt und gelesen wird und dies exemplarisch für ein (8×4) großes Bild durchgeführt. Für jeden Schritt sind diejenigen Pixel grau unterlegt, die zu diesem Zeitpunkt gelesen werden und die blau schraffierten Pixel, werden zu diesem Zeitpunkt an den Beschleuniger weitergeleitet. In der oberen Zeile der Abbildung sind die ersten drei Speicherzyklen zu sehen, in der die ersten drei Zeilen des Originalbildes eingelesen werden. Nach zwei Takten kann mit der Berechnung des ersten Pixels A1 des gefalteten Resultats begonnen werden, da alle hierfür benötigten Pixel (in der Abbildung schraffiert markiert) geladen wurden. Diese werden an den Beschleuniger weitergeleitet, der die eigentliche Faltung mit einer der Faltmatrices durchführt. Ab diesem Zeitpunkt können je neun Pixel sequentiell weitergeleitet werden, um einen weiteren Pixel des zu resultierenden Bildes zu berechnen. Sobald die geladenen Pixel eines Lade-Zyklus nicht mehr für eine Berechnung eines weiteren Pixels in dieser Zeile benötigt werden, werden diese verworfen und durch die nächsten Pixel des Originalbildes ersetzt. Dieser Ablauf wird bis zum Zeilenende des Originalbildes fortgesetzt und sobald dieses erreicht wird, werden die nächsten drei Zeilen des Originalbildes sukzessive geladen und die Berechnung des ersten Pixels der nächsten Zeile kann ohne Unterbrechung erfolgen. Dieser Ablauf wird fortgeführt, bis alle Pixel des Originalbildes an den Beschleuniger weitergegeben wurden. Diese Vorgehensweise minimiert zwar den Speicherbedarf des Interfaces,

erhöht jedoch den insgesamt nötigen Datentransfer um nahezu den Faktor drei. Abgesehen von den Rändern, wird jedes Pixel des Originalbildes bei einer Faltung mit einer (3×3) großen Faltungsmatrix für drei Zeilen des resultierenden Bildes für die Berechnung von je drei Pixeln benötigt. Anstatt sie zwischenzuspeichern, verwirft das entworfene Interface die Pixel jedoch, sobald sie für die aktuelle Zeile nicht mehr relevant sind. Dies ist der Limitierung der rekonfigurierbaren Einheit und somit der maximalen Anzahl der zur Verfügung stehenden Ressourcen für ein *Atom* geschuldet. Es ist nicht möglich, die Pixel solange zwischenzuspeichern, bis sie nicht mehr benötigt werden, da der hierfür nötige Speicherplatz nicht vorhanden ist. Aus diesem Grunde werden fast alle Pixel dreimal eingelesen, was den Speicherbedarf auf (8×3) Einträge minimiert. Dieser Ressourcengewinn wird mit einer steigenden Anzahl von notwendigen Speicherzugriffen bezahlt, die den Beschleuniger jedoch in der Ausführungszeit nicht beeinflusst. Nach dem zweiten Takt wird durch den Beschleuniger pro Takt ein Pixel des Ausgabebildes ausgegeben. Mit welcher Faltungsmatrix die Berechnung durchgeführt wird, wird in diesem Fall durch das *Very Long Control Word* kodiert. Abgesehen von der Veränderung des VLCWs wird der modulare Beschleuniger wie jeder andere Beschleuniger des *i*-Cores durch eine Spezialinstruktion angesprochen.

8.2 Evaluierung und Ausblick

Für die Evaluierung wurden nur die Ressourcen für die eigentlichen Beschleuniger betrachtet und nicht des Interfaces, da dieses für eine Bewertung des Konzeptes eines modularen Beschleunigers nicht relevant ist. Für jede der in Abschnitt 2.5 vorgestellten Filtermatrix wurde ein Beschleuniger für die Virtex 7 Plattform des Herstellers Xilinx implementiert, der eine Faltung eines Teilbildes mit genau dieser Faltungsmatrix realisiert. Somit wurden vier eigenständige Beschleuniger erstellt, die dann zu einem modularen Beschleuniger vereint wurden. Bei der Vereinigung wurde darauf geachtet, dass die vier Beschleuniger nicht einfach parallel nebeneinander auf dem FPGA platziert wurden, sondern dass diese soweit wir möglich überlappen und ein gemeinsamer Datenpfad entsteht. In Tabelle 8.1 sind die wesentlichen Ergebnisse der unterschiedlichen Beschleuniger angegeben. Die Taktfrequenz der einzelnen Beschleuniger liegt leicht oberhalb der

Implementierung	Anzahl LUT	Taktfrequenz in MHz	Bildrate in fps
Gauß-Filter	50	110.59	141
Laplace-Filter	53	117.73	142
Sobel_X-Filter	26	112.02	142
Sobel_Y-Filter	26	111.83	142
Modularer Beschleuniger	126	108.05	138

Tabelle 8.1: Ergebnisse der Implementierung für verschiedene Beschleuniger und dem modularen Beschleuniger.

Taktfrequenz des modularen Beschleunigers, da zusätzlich zu der Berechnung des gefalteten Bildes noch das *Select*-Signal ausgewertet werden muss. Bei einer Bildgröße von ($1\,024 \times 768$) können die einzelnen Beschleuniger eine Bildrate von bis zu 142 Bildern pro Sekunde erreichen, der modulare Beschleuniger kommt durch die leicht niedrigere Taktfrequenz auf bis zu 138 Bildern pro Sekunde. Wobei dieser unabhängig von der gewählten Filtermatrix immer die gleiche Bildrate aufweist und der Unterschied zwischen der Bearbeitungszeit des modularen Beschleunigers und dem Beschleuniger für den Gaußfilter nur rund 2 % beträgt. Bei Beurteilung des Ressourcenbedarfes müssen die vier einzelnen Beschleuniger in Summe betrachtet und mit dem modularen Beschleuniger verglichen werden, da dieser die Funktionalität aller Beschleuniger miteinander vereint. Für die vier einzelnen Beschleuniger fällt insgesamt ein Ressourcenbedarf von 155 *Lookup*-Tabellen an, der modulare Beschleuniger benötigt für den identischen Funktionsumfang jedoch rund 18 % weniger Logikeinheiten. Da nur ein Beschleuniger zur Zeit in einem *Atom Container* geladen sein kann, spielt der absolute Mehrbedarf an Ressourcen des modularen Beschleunigers kein Hindernis dar, solange der Beschleuniger in den Container platziert werden kann. Der größte Vorteil des modularen Beschleunigers liegt darin, dass er vier Funktionalitäten miteinander vereint und bei einem Wechsel der Anwendung von einem unterstützten Filter auf einen anderen, kann dieser sofort weiter genutzt werden, ohne dass eine Rekonfigurierung durchgeführt werden muss. Eine solche Rekonfiguierung, also das Laden eines neuen Beschleunigers in die rekonfigurierbare Einheit des *i*-Cores, kann wie in Unterabschnitt 7.4.3 erläutert, mehrere tausend Iterationen des Beschleunigers in Anspruch nehmen und den Mehrwert

dieses Beschleunigers deutlich senken. Der modulare Beschleuniger nutzt die einem *Atom* zur Verfügung stehenden Ressourcen effizient aus und hat im Vergleich zu den einzelnen Beschleunigern kaum Performanceeinbußen, zudem erweitert es das Konzept der Spezialinstruktionen konsequent bis in das Innere des *Atoms* hinein. Der in Kapitel 7 erläuterte Ansatz der transparenten und dynamischen Hardwarebeschleunigung kann mithilfe dieser Erweiterung des modularen Aufbaus eines einzelnen Beschleunigers deutlich flexibler angewendet werden. Zudem ermöglicht der modulare Aufbau weitere Möglichkeiten verschiedene Ausführungen eines Algorithmus in einem Beschleuniger zu integrieren. Somit können nicht nur verschiedene Algorithmen mit ähnlichen Datenflüssen miteinander vereint werden, sondern auch verschiedene Ausprägungen eines Beschleunigers. Ein Beispiel für die verschiedenen Ausprägungen kann hierbei die Granularität einer Approximation darstellen, zur Laufzeit würde dies dann einen dynamischen Wechsel der verschiedenen Granularitätsstufen ermöglichen. Das System könnte somit zur Laufzeit flexibel auf die Anforderungen reagieren und bei Bedarf die Genauigkeit der Berechnung beispielsweise auf Kosten der Latenz erhöhen. Das in dieser Arbeit vorgestellte und realisierte Konzept der transparenten und dynamischen Hardwarebeschleunigung soll in Kombination mit den modularen Beschleunigern den Grundstein für weitere Forschungen im Bereich der dynamischen Hardwarebeschleunigung legen.

9 Schlussfolgerung

Im Rahmen dieser Arbeit wurden flexible Mikroarchitekturen entworfen und evaluiert, die applikationsspezifisch Parameter der Architektur optimieren. Hierbei wird im Vorfeld die zu erwartenden Applikationen und Daten analysiert und von mit Hilfe eines entworfenen Frameworks werden die aus der Analyse erhaltenden Ergebnisse zum Entwurf einer applikationsspezifischen Architektur verwendet. Die entworfenen Mikroarchitekturen erfüllen die gestellten Anforderungen durch eine neuartige und effiziente Nutzung der vorhandenen Ressourcen. Diese Mikroarchitekturen werden durch das Framework automatisch generiert und durch die Verwendung der in der Analyse ermittelten Parameter können die zur Verfügung stehenden Ressourcen effizient genutzt werden. Für die inhalts-adaptive Speicherstruktur werden hierfür die zu speichernden Daten im Vorfeld untersucht und eine Architektur generiert, die für eine effiziente Verarbeitung genau dieser Daten ausgelegt ist. Hierbei werden ebenfalls die zur Verfügung stehenden Ressourcen analysiert, um eine effiziente Abbildung der Daten in der verwendeten Speicherarchitektur zu gewährleisten. Durch die im Vorfeld ausgeführten Analysen und die applikationsspezifische Generierung der Architektur wird ein zur Laufzeit hocheffizientes System für diesen Anwendungsfall erstellt. Das entworfene Konzept bleibt durch die automatische Generierung dennoch flexibel einsetzbar, ist adaptierbar und zeigt zudem das Potential auf, welches in einer Verschiebung der Komplexität des Anwendungsfalls zur Laufzeit auf Analysen im Vorfeld liegt. Ein weiterer verfolgter Ansatz ist das Konzept einer transparenten und dynamischen Hardwarebeschleunigung eines adaptiven Prozessors. Hierfür wurde ein Automatismus entworfen und dem Prozessor zur Verfügung gestellt, mit der dieser eigenständig zur Laufzeit rechenintensive Kernel detektieren und beschleunigen kann. Diese transparente Erweiterung des *General-Purpose* Prozessors ermöglicht eine Beschleunigung, ohne dass die Anwendungen dahingehend optimiert werden muss, und bewahrt somit alle Vorteile der ursprünglichen Prozessorarchitektur. Beispielsweise sind keine Änderungen

des *Compilers* notwendig, das Modul der automatischen, transparenten und dynamischen Beschleunigung ist direkt an die Pipeline des Prozessors geknüpft und generiert eigenständig alle notwendigen Instruktionen. Der Mehrbedarf an Ressourcen für die Realisierung der automatisch generierten Spezialinstruktionen ist durch das Potential der Leistungssteigerung mehr als gerechtfertigt und führt in keinem Fall zu einer steigenden Verarbeitungsdauer der laufenden Anwendung. Die erzielten Ergebnisse zeigen, dass mittels der entworfenen Konzepte eine erhebliche Beschleunigung von *Single-Thread*-Anwendungen erreicht werden kann, welche weit über die Grenzen dessen gehen, was durch aktuelle Architekturentwicklungen am Markt möglich ist. Darüber hinaus werden die Ressourcen der rekonfigurierbaren Einheit des adaptiven Prozessors durch den Einsatz der entwickelten modularen Beschleuniger optimal genutzt. Auf diese Weise können eine Vielzahl von Anwendungen beschleunigt werden, ohne dass zusätzliche Ressourcen für dedizierte Hardwarebeschleuniger anfallen. Zusammenfassend kann gezeigt werden, dass mit relativ geringem Entwicklungsaufwand leistungsstarke und flexible Mikroarchitekturen entworfen und realisiert werden können, wenn ein Hauptaugenmerk auf die effiziente Nutzung der vorhandenen Ressourcen gelegt wird. In naher Zukunft wird die schnelle und kostengünstige Entwicklung von flexiblen und anwendungsoptimierte Systemen immer mehr in den Fokus der Anbieter von Mikroprozessoren gelangen und die Zusammenarbeit mit Entwicklern und Herstellern von programmierbaren Logik-ICs forciert werden. In der Modellierung solch neuartiger Mikroarchitekturen liegt ein enormes Potential und die durch den Wegfall des *Dennard Scalings* entstandene Lücke der Performanzsteigerung kann zunächst durch deren Einsatz kurzfristig geschlossen werden. In naher Zukunft wird der Trend vermehrt in Richtung von flexiblen Mikroarchitekturen gehen, welche die zur Verfügung stehenden Ressourcen für verschiedene Anwendungen hocheffizient nutzen. Hierzu ist es notwendig alle Ebenen von Rechnersystemen gemeinsam zu betrachten und diese synergetisch miteinander zu verschmelzen. Neue Forschungsgebiete wie beispielsweise das *Approximate Computing*, erschaffen auf diese Weise neue Entwurfsparadigmen, welche die Leistung der Rechensysteme auch weiterhin steigern lassen.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AGU *Address Generation Unit*
AHB *Advanced High-performance* Bus
AM-Chip *Associative Memory* Chip
ASAP *As Soon as possible*
ASIC *Application-Specific Integrated Circuit*
ASIP *Application-Specific Instruction-Set* Prozessor
ATLAS *A Toroidal LHC ApparatuS*
Auto-SI automatisch generierte Spezialinstruktion

BRAM *Block Random Access Memory*

CAM *Content Addressable Memory*
CBC CMS *Binary Chip*
CERN *Conseil européen pour la recherche nucléaire*
CLB *Configurable Logic* Block
CMS *Compact Muon Solenoid*
CP Co-Prozessor
CPU *Central Processing Unit*
CWP *Current Window Pointer*

DFG Datenflussgraph
DSP *Digital Signal Processor*

ESA *European Space Agency*

FAU *Friedrich-Alexander-Universität Erlangen-Nürnberg*
FF Flipflop
FPGA *Field Programmable Gate Array*
FPU *Floating-Point Unit*

GPL GNU *General Public License*
GPP *General-Purpose* Prozessor
GPU *Graphics Processing Unit*

HL-LHC *High Luminosity-Large Hadron Collider*

ICC *Integer Condition Code*
ILP *Instruction Level Parallelism*
InvasIC *Invasive Computing*
IP-Core *Intellectual Property Core*
ISA *Instruction Set Architecture*
IU *Integer Unit*

KIT *Karlsruher Institut für Technologie*

LHC *Large Hadron Collider*
LSU *Load/Store-Unit*
LUT *Lookup*-Tabelle

MAC *Multiply-Accumulate*
MLSA *Match-Line Sense Amplifier*
MOSFET *Metal-Oxide-Semiconductor* Feldeffekttransistor
MST *Mininum Spanning Tree*

NN *Nearest Neighbor*
NoC *Network on Chip*

PC *Program Counter*
PSR *Processor State Register*

RAM *Random Access Memory*
RF *Register File*
RISC *Reduced Instruction Set Computer*
RISPP *Rotating Instruction Set Processing* Plattform

SI Spezialinstruktion
SI ID *SI Identifier*
SOC *System On Chip*
SPARC *Scalable Processor Architecture*
SRAM *Static Random-Access Memory*
SRMMU *SPARC V8 Reference Memory Management Unit*
SSM *SI State Memory*

TCPA *Tightly-coupled Processor Array*
TLM *Tile-local Memory*
TSP *Traveling Salesman* Problem
TUM *Technische Universität München*

VHDL *Very High Speed Integrated Circuit Hardware Description Language*
VLCW *Very Long Control Word*

Literaturverzeichnis

[1] Duccio Abbaneo. Upgrade of the CMS tracker with tracking trigger. *Journal of Instrumentation*, 6(12):C12065, 2011.

[2] Yves Allkofer et al. Design and performance of the silicon sensors for the CMS barrel pixel detector. *Nucl. Instrum. Meth.*, A584:25–41, 2008.

[3] ALTERA. Volume 1: Device Interfaces and Integration (sv-5v1). *Stratix V Device Handbook*, 9 2018.

[4] Christian Amstutz. *Evaluation of an Associative Memory and FPGA-based System for the Track Trigger of the CMS-Detector*. PhD thesis, Karlsruhe, 2016.

[5] A. Annovi, R. Beccherle, M. Beretta, E. Bossini, F. Crescioli, M. Dell'Orso, P. Giannetti, J. Hoff, T. Liu, V. Liberali, I. Sacco, A. Schoening, H. K. Soltveit, A. Stabile, R. Tripiccione, and G. Volpi. Associative memory design for the fast track processor (FTK) at ATLAS. In *2011 IEEE Nuclear Science Symposium Conference Record*, pages 141–146, 10 2011.

[6] A. Annovi, G. Broccolo, A. Ciocci, P. Giannetti, F. Ligabue, D. Magalotti, A. Nappi, M. Dell'Orso, R. Dell'Orso, F. Palla, E. Pedreschi, M. Piendibene, L. Servoli, S. Taroni, and G. Volpi. Associative Memory for L1 Track Triggering in LHC Environment. *IEEE Transactions on Nuclear Science*, 60(5):3627–3632, 10 2013.

[7] Alberto Annovi. The FTK: A hardware track finder for the ATLAS Trigger. In *2014 19th IEEE-NPSS Real Time Conference*, pages 1–3, 5 2014.

[8] ARM Limited. *AMBA^{TM}Specification*, revision 2.0 edition, 1999.

[9] Igor Arsovski and Ali Sheikholeslami. A Current-Saving Match-Line Sensing Scheme for Content-Addressable Memories. In *2003*

IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC., pages 304–494, 2003.

[10] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An Optimal Algorithm for Approximate NearestNeighbor Searching in Fixed Dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.

[11] Paolo Azzurri. The CMS Silicon Strip Tracker. *Journal of Physics: Conference Series*, 41(1):127, 2006.

[12] Rajul Bansal and Abhijit Karmakar. Efficient integration of co-processor in LEON3 processor pipeline for System-on-Chip design. *Microprocessors and Microsystems*, 51:56–75, 2017.

[13] David Barney. CMS Detector Slice. CMS Collection., 1 2016.

[14] Lars Bauer. *RISPP: A Run-time Adaptive Reconfigurable Embedded Processor*. PhD thesis, 2009.

[15] Mandell Bellmore and George Nemhauser. The Traveling Salesman Problem: A Survey. *Operations Research*, 16(3):538–558, 1968.

[16] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Nielsen, and A. Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[17] Giovanni Bianchi. tkLayout: a design tool for innovative silicon tracking detectors. *Journal of Instrumentation*, 9(03):C03054, 2014.

[18] William Brinkman, Douglas Haggan, and William Troutman. A History of the Invention of the Transistor and Where It Will Lead Us. *IEEE Journal of Solid-State Circuits*, 32(12):1858–1865, 12 1997.

[19] Uwe Brinkschulte and Theo Ungerer. Mikrocontroller und Mikroprozessoren, 2010.

[20] David Britton and Stephen Lloyd. How to deal with petabytes of data: the LHC Grid project. *Reports on Progress in Physics*, 77(6):065902, 2014.

[21] O. Brüning, P. Collier, P. Lebrun, S. Myers, R. Ostojic, J. Poole, and P. Proudlock. LHC Design Report Vol.1: The LHC Main Ring. 2004.

[22] Oliver Brüning and Lucio Rossi. *The High Luminosity Large Hadron Collider: The New Machine for Illuminating the Mysteries of Universe*. World Scientific Publishing Company, 2015.

[23] Lea Caminada. *Study of the Inclusive Beauty Production at CMS and Construction and Commissioning of the l Barrel Detector (Springer Theses)*. Springer, 2012.

[24] K. Cankoçak, P. de Barbaro, D. Vishnevskiy, Y. Onel, and CMS-HCAL Collaboration. CMS HCAL installation and commissioning. *Journal of Physics: Conference Series*, 160(1):012055, 2009.

[25] A. Caratelli, D. Ceresa, J. Kaplon, K. Kloukinas, and S. Scarfi. Readout architecture for the Pixel-Strip module of the CMS Outer Tracker Phase-2 upgrade. Technical Report CMS-CR-2016-405, CERN, Geneva, 11 2016.

[26] CERN. LHC Guide. 3 2017.

[27] CERN. CERN Annual report 2017. Technical report, CERN, Geneva, 2018.

[28] Jason Chaves. Implementation of FPGA-based level-1 tracking at CMS for the HL-LHC. *Journal of Instrumentation*, 9(10):C10038, 2014.

[29] Cobham Gaisler AB. *GRLIB IP Core User's Manual*, 2018.1 edition, 2018.

[30] Cobham Gaisler AB. *GRLIB IP Library User's Manual*, 2018.1 edition, 2018.

[31] CMS Collaboration. CMS, the magnet project: Technical design report. 1997.

[32] CMS Collaboration. *The CMS muon project: Technical Design Report*. Technical Design Report CMS. CERN, Geneva, 1997.

[33] CMS Collaboration. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004, 2008.

[34] CMS Collaboration. Detector Drawings. 3 2012. CMS Collection.

[35] CMS Collaboration. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Physics Letters B*, 716(1):30 – 61, 2012.

[36] CMS Collaboration. The Phase-2 Upgrade of the CMS L1 Trigger Interim Technical Design Report. Technical Report CERN-LHCC-2017-013. CMS-TDR-017, CERN, Geneva, 9 2017.

[37] UA1 Collaboration. Experimental Observation of Isolated Large Transverse Energy Electrons with Associated Missing Energy at s=540 GeV. *Physics Letters B*, 122(1):103–116, 1983.

[38] D. Contardo, M. Klute, J. Mans, L. Silvestris, and J. Butler. Technical Proposal for the Phase-II Upgrade of the CMS Detector. Technical Report CERN-LHCC-2015-010. LHCC-P-008. CMS-TDR-15-02, Geneva, 6 2015.

[39] Thomas Cormen, editor. *Algorithmen - eine Einführung*. Oldenbourg, 4 edition, 2013.

[40] Marvin Damschen, Lars Bauer, and Jörg Henkel. Timing Analysis of Tasks on Runtime Reconfigurable Processors. *IEEE Trans. on Very Large Scale Integration Syst.*, 25(1):294–307, June 2016.

[41] M. Daněk, L. Kafka, L. Kohout, J. Sýkora, and R. Bartosinski. *UT-LEON3: Exploring Fine-Grain Multi-Threading in FPGAs*. Circuits and Systems. Springer, 2012.

[42] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 10 1974.

[43] S. Derrien, I. Puaut, P. Alefragis, M. Bednara, H. Bucher, C. David, Y. Debray, U. Durak, I. Fassi, C. Ferdinand, D. Hardy, A. Kritikakou, G. Rauwerda, S. Reder, M. Sicks, T. Stripf, K. Sunesen, T. ter Braak, N. Voros, and J. Becker. WCET-Aware Parallelization of Model-Based Applications for Multi-Cores: the ARGO Approach. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 286–289, 3 2017.

[44] Reinhard Diestel. *Graphentheorie (German Edition)*. Springer Spektrum, 2017.

[45] Richard Duda and Peter Hart. Use of the Hough Transformation to Detect Lines and Curves in Pictures. *Communications of the ACM*, 15(1):11–15, January 1972.

[46] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 6 2011.

[47] Lyndon Evans and Philip Bryant. LHC Machine. *Journal of Instrumentation*, 3(08):S08001, 2008.

[48] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer, 2014.

[49] Markus Friedl. *The CMS Silicon Strip Tracker and its Electronic Readout*. PhD thesis, 2001.

[50] L. Frontini, S. Shojaii, A. Stabile, and V. Liberali. A new XOR-based Content Addressable Memory architecture. In *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*, pages 701–704, 2012.

[51] A. Fyodorov, M. Korzhik, O. Missevitch, V. Pavlenko, V. Kachanov, A. Singovsky, A. Annenkov, V. Ligun, J. Peigneux, J. Vialle, J. Faure, and F. Binon. Progress in PbWO4 scintillating crystal. *Radiation Measurements*, 26(1):107–115, 1996.

[52] Jiri Gaisler. *A structured VHDL design method*. Gaisler Research.

[53] Jiri Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *Proceedings International Conference on Dependable Systems and Networks*, pages 409–415, 2002.

[54] R. Garner, A. Agrawal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck. The Scalable Processor Architecture (SPARC). In *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*, pages 278–283, 1988.

[55] Robert Garner. *SPARC: Scalable Processor Architecture*, pages 75–100. Springer New York, New York, NY, 1990.

[56] Luke Georghiou. Global cooperation in research. *Research Policy*, 27(6):611–626, 1998.

[57] Ralf Gessler and Thomas Mahr. *Hardware-Software-Codesign : Entwicklung flexibler Mikroprozessor-FPGA-Hochleistungssysteme*.

Vieweg, Wiesbaden, 2007.

[58] Ronald Graham and Pavol Hell. On the History of the Minimum Spanning Tree Problem. *Annals of the History of Computing*, 7(1):43–57, 1 1985.

[59] Hartwig Grote. Pattern recognition in high-energy physics. *Reports on Progress in Physics*, 50(4):473, 1987.

[60] G. Hall, M. Pesaresi, M. Raymond, D. Braga, L. Jones, P. Murray, M. Prydderch, D. Abbaneo, G. Blanchot, A. Honma, M. Kovacs, and F. Vasey. CBC2: A CMS microstrip readout ASIC with logic for track-trigger modules at HL-LHC. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 765:214–218, 2014.

[61] Geoffrey Hall. A time-multiplexed track-trigger for the CMS HL-LHC upgrade. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 824:292 – 295, 2016. Frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors.

[62] Geoffrey Hall, Mark Raymond, and Andrew Rose. 2-D PT module concept for the SLHC CMS tracker. *Journal of Instrumentation*, 5(07):C07012, 2010.

[63] Richard Hamming. Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, 29(2):147–160, 4 1950.

[64] Tao Han, editor. *Proceedings of Theoretical Advanced Study Institute in Elementary Particle Physics on The dawn of the LHC era (TASI 2008)*. World Scientific, World Scientific, 2010.

[65] Jan Heißwolf. *A Scalable and Adaptive Network on Chip for Many-Core Architectures*. PhD thesis, 2014.

[66] J. Henkel, L. Bauer, A. Grudnitsky, and H. Zhang. Adaptive Embedded Computing with *i*-core. *SIGBED Rev.*, 11(3):20–21, November 2014.

[67] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. Pujari, A. Grudnitsky, J. Heißwolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive Manycore Architectures. In *17th Asia and South Pacific Design Automation Conference*, pages 193–200, 1 2012.

[68] A. Hermann, L. Belloni, U. Mersits, D. Pestre, and J. Krige. *History of CERN, I: Volume I - Launching the European Organization for Nuclear Research*. North Holland, 1987.

[69] Alla Herve. The CMS Detector Magnet. *IEEE Transactions on Applied Superconductivity*, 10(1):389–394, 3 2000.

[70] Bernd Jähne. *Digitale Bildverarbeitung : mit Übungsaufgaben und CD-ROM*. Springer, Berlin, 6 edition, 2005.

[71] D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich. A Highly Parameterizable Parallel Processor Array Architecture. In *2006 IEEE International Conference on Field Programmable Technology*, pages 105–112, 12 2006.

[72] S. Korf, G. Sievers, J. Ax, D. Cozzi, T. Jungeblut, J. Hagemeyer, M. Porrmann, and U. Rückert. Dynamisch rekonfigurierbare Hardware als Basistechnologie für intelligente technische Systeme. pages 79–90, 2013.

[73] Nikolai Krasnikov and Viktor Matveev. Search for New Physics at Large Hadron Collider. *Physics of Atomic Nuclei*, 73(1):191–200, 1 2010.

[74] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 26(2):203–215, 2007.

[75] Syue-Wei Li, Apollo Go, Chia-Ming Kuo, and CMS Preshower Collaboration. Performance of CMS ECAL Preshower in 2007 test beam. *Journal of Physics: Conference Series*, 160:012052, 4 2009.

[76] Shen Lin and Brian Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, 1973.

[77] Hans Lipp and Jürgen Becker. *Grundlagen der Digitaltechnik*. Oldenbourg, München, 7 edition, 2011.

[78] N. Loubet, T. Hook, P. Montanini, C.-W. Yeung, S. Kanakasabapathy, M. Guillom, T. Yamashita, J. Zhang, X. Miao, J. Wang, A. Young, R. Chao, M. Kang, Z. Liu, S. Fan, B. Hamieh, S. Sieg, Y. Mignot, W. Xu, and M. Khare. Stacked nanosheet gate-all-around transistor to enable scaling beyond FinFET. pages T230–T231, 06

2017.

[79] D. Magalotti, M. Biasini, P. Gunnellini, A. Nappi, L. Servoli, and S. Taroni. Use of Associative Memories for L1 triggering in LHC environment. Workshop on Intelligent Tracker, WIT2012, 2012.

[80] Jan-Pierre Merlo. CMS Hadronic Forward Calorimeter. *Nuclear Physics B - Proceedings Supplements*, 61(3):41–46, 1998. Proceedings of the Fifth International Conference on Advanced Technology and Particle Physics.

[81] Dieter Meschede. *Gerthsen Physik (Springer-Lehrbuch) (German Edition)*. Springer Spektrum, 2015.

[82] Esma Mobs. The cern accelerator complexcomplexe des accélérateurs du cern. 12 2008.

[83] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 4 1965.

[84] Stefan Näher. *The Travelling Salesman Problem*, pages 383–391. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[85] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 5 2008.

[86] Kostas Pagiamtzis and Ali Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712–727, 2006.

[87] Fabrizio Palla, Mark Pesaresi, and Anders Ryd. Track Finding in CMS for the Level-1 Trigger at the HL-LHC. *Journal of Instrumentation*, 11(03):C03011, 2016.

[88] David Patterson and David Ditzel. The Case for the Reduced Instruction Set Computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, October 1980.

[89] Mark Pesaresi and Geoffrey Hall. Simulating the performance of a pT tracking trigger for CMS. *Journal of Instrumentation*, 5(08):C08003, 2010.

[90] B. Povh, K. Rith, C. Scholz, F. Zetsche, and W. Rodejohann. *Teilchen und Kerne: Eine Einführung in die physikalischen Konzepte (Springer-Lehrbuch) (German Edition)*. Springer Spektrum, 2013.

[91] Valeria Radicci. CMS pixel detector upgrade. *Journal of Instrumentation*, 4(03):P03022, 2009.

[92] Martin Riedlberger. Entwicklung des rekonfigurierbaren Prozessors *i*-Core auf Basis des LEON3 Systems. Diplomarbeit, Karlsruher Institut für Technologie, 2013.

[93] Michael Riordan and Lillian Hoddeson. The origins of the pn junction. *IEEE Spectrum*, 34(6):46–51, 6 1997.

[94] Karl Rupp. 40 Years of Microprocessor Trend Data, 2018. https://github.com/karlrupp/microprocessor-trend-data, zuletzt abgerufen am 25.02.2019.

[95] Armin Scheurer and the German CMS Community. German contributions to the CMS computing infrastructure. *Journal of Physics: Conference Series*, 219(6):062064, 2010.

[96] Burkhard Schmidt. The High-Luminosity upgrade of the LHC: Physics and Technology Challenges for the Accelerator and the Experiments. *Journal of Physics: Conference Series*, 706(2):022002, 2016.

[97] Jörg Schulze. *Konzepte siliziumbasierter MOS-Bauelemente*. Springer, Berlin, 2005.

[98] Giacomo Sguazzoni. Upgrades of the CMS Outer Tracker for HL-LHC. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 845:164–168, 2017. Proceedings of the Vienna Conference on Instrumentation 2016.

[99] John Shalf and Robert Leland. Computing beyond Moore's Law. *Computer*, 48(12):14–23, 12 2015.

[100] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[101] SPARC International, Inc. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[102] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. *Invasive Computing: An Overview*, pages 241–268. Springer New York, New York, NY, 2011.

[103] J. Teich, A. Weichslgartner, B. Oechslein, and W. Schröder-Preikschat. Invasive computing - Concepts and Overheads. In *Proceeding of the 2012 Forum on Specification and Design Languages*, pages 217–224, 9 2012.

[104] Konstantinos Theofilatos and the CMS ECAL Collaboration. CMS Electromagnetic Calorimeter status and performance with the first LHC collisions. *Journal of Physics: Conference Series*, 293(1):012042, 2011.

[105] Klaus Tönnies. *Grundlagen der Bildverarbeitung*. Informatik. Pearson Studium, München, 2005.

[106] Davide Tommasini, Marco Buzio, and Regis Chritin. Dipole Magnets for the LHeC Ring-Ring Option. *IEEE Transactions on Applied Superconductivity*, 22(3):4000203–4000203, 6 2012.

[107] Carsten Tradowsky. *Methoden zur applikationsspezifischen Effizienzsteigerung adaptiver Prozessorplattformen*. PhD thesis, Karlsruhe, 2016.

[108] Transregional Collaborative Research Centre 89. Invasive Computing Annual Report 2012, 12 2012.

[109] Alessia Tricomi. Upgrade of the CMS tracker. *Journal of Instrumentation*, 9(03):C03041, 2014.

[110] Berthold Vöcking. Algorithms unplugged, 2011.

[111] Craig Waller. Content addressable memory cells and systems and devices using the same, 2003. US Patent 6,597,594.

[112] David Weaver. *OpenSPARC Internals: OpenSPARC T1/T2 CMT Throughput Computing*. Sun Microsystems, 2008.

[113] S. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh. Design considerations for soft embedded programmable logic cores. *IEEE Journal of Solid-State Circuits*, 40(2):485–497, 2 2005.

[114] XILINX. Virtex-5 FPGA User Guide UG190 (v5.4). 3 2012.

[115] XILINX. UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices. 6 2014.

[116] XILINX. 7 Series FPGAs Configurable Logic Block User Guide ug474 (v1.8). 9 2016.

[117] XILINX. UltraScale Architecture and Product Data Sheet: Overview. 11 2018.

[118] XILINX. White Paper: The First Adaptive Compute Acceleration Platform (ACAP). 10 2018.

Betreute studentische Arbeiten

[Bad15] Bader, Jan: *Entwurf, Implementierung und Evaluierung einer Vergleichsmatrix für eine FPGA-Speicherstruktur mit Mustererkennung*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 2062, Dezember 2015

[Bru16] Bruckner, Max: *Entwurf, Implementierung und Evaluierung einer Pipeline-Architektur für eine FPGA-Speicherstruktur mit Mustererkennung*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 2129, Mai 2016

[He18] He, Zhuofan: *Entwurf und Modellierung eines laufzeitadaptiven Beschleunigers für eine rekonfigurierbare Prozessorarchitektur*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 2457, Dezember 2018

[Kim16] Kimmerle, Martin: *Erweiterung einer Prozessorarchitektur um approximative Hardwarestrukturen*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2197, März 2016

[Rei19] Reizinger, Patrik: *A Review of Approximate Computing methods for a Universal Approximate Hardware Accelerator*, Karlsruher Institut für Technologie, Seminararbeit, Januar 2019

[Sch17] Schade, Christoph: *Integration von laufzeit-adaptiver Rekonfiguration von Hardware-Beschleunigern in eine LEON3 Architektur*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2212, März 2017

[Seb14] Seboui, Mahmoud: *Entwurf, Implementierung und Evaluierung einer Speicherstruktur im Xilinx B-RAM; Design, Implementation and Evaluation of a Memory Architecture in XILINX B-RAM*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1920, November 2014

[Vil19] Villacis, Javier: *Implementierung und Evaluierung einer Vergleichsmatrix für eine inhalts-adaptive FPGA Speicherstruktur*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 2523, Mai 2019

[Zho16] Zhong, Huan: *Design, Implementation and Evaluation of an FPGA Memory Architecture with Pattern Recognition Capability*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2071, Januar 2016

Konferenzbeiträge

[ABB+16a] C. Amstutz, F. Ball, M. Balzer, J. Brooke, L. Calligaris, D. Cieri, E. Clement, G. Hall, T. Harbaum, K. Harder, P. Hobson, G. Iles, T. James, K. Manolopoulos, T. Matsushita, A. Morton, D. Newbold, S. Paramesvaran, M. Pesaresi, I. Reid, A. Rose, O. Sander, T. Schuh, C. Shepherd-Themistocleous, A. Shtipliyski, S. Summers, A. Tapper, I. Tomalin, K. Uchida, P. Vichoudis, and M. Weber. Emulation of a prototype FPGA track finder for the CMS Phase-2 upgrade with the CIDAF emulation framework. In *2016 IEEE-NPSS Real Time Conference (RT)*, pages 1–7, June 2016.

[ABB+16b] C. Amstutz, F. Ball, M. Balzer, J. Brooke, L. Calligaris, D. Cieri, E. Clement, G. Hall, T. Harbaum, K. Harder, P. Hobson, G. Iles, T. James, K. Manolopoulos, T. Matsushita, A. Morton, D. Newbold, S. Paramesvaran, M. Pesaresi, I. Reid, A. Rose, O. Sander, T. Schuh, C. Shepherd-Themistocleous, A. Shtipliyski, S. Summers, A. Tapper, I. Tomalin, K. Uchida, P. Vichoudis, and M. Weber. An FPGA-based track finder for the L1 trigger of the CMS experiment at the high luminosity LHC. In *2016 IEEE-NPSS Real Time Conference (RT)*, pages 1–9, June 2016.

[HBWB18] T. Harbaum, M. Balzer, M. Weber, and J. Becker. A Content - Adapted FPGA Memory Architecture with Pattern Recognition Capability and Interval Compressing Technique. In *2018 31st IEEE International System-on-Chip Conference (SOCC)*, pages 206–212, Sep. 2018.

[HSB+16] T. Harbaum, M. Seboui, M. Balzer, J. Becker, and M. Weber. A Content Adapted FPGA Memory Architecture with Pattern Recognition Capability for L1 Track Triggering in the LHC Environment. In *2016 IEEE 24th Annual International Sym-*

posium on Field-Programmable Custom Computing Machines (FCCM), pages 184–191, May 2016.

[HSD+17] T. Harbaum, C. Schade, M. Damschen, C. Tradowsky, L. Bauer, J. Henkel, and J. Becker. Auto-SI: An Adaptive Reconfigurable Processor with Run-time Loop Detection and Acceleration. In *30th IEEE International System-on-Chip Conference (SOCC)*, pages 224–229, 2017.

[OHK+11] J. Oberländer, T. Harbaum, G. Kurz, N. Ahmed, T. Kos-Grabar, A. Hermann, A. Rönnau, and R. Dillmann. A Student-built Ball-throwing Robotic Companion for Hands-on Robotics Education. In *CLAWAR 2011*, 2011.

[THDB13] C. Tradowsky, T. Harbaum, S. Deyerle, and J. Becker. LImbiC: An adaptable architecture description language model for developing an application-specific image processor. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 34–39, Aug 2013.

Veröffentlichungen der CMS-Kollaboration

[AA15] Aad, G. ; Abbott, N. B.and W. B.and Woods: Combined Measurement of the Higgs Boson Mass in *pp* Collisions at $\sqrt{s} = 7$ and 8 TeV with the ATLAS and CMS Experiments. In: *Phys. Rev. Lett.* 114 (2015), May, 191803. `http://dx.doi.org/10.1103/PhysRevLett.114.191803`. – DOI 10.1103/PhysRevLett.114.191803

[KST+15] Khachatryan, V. ; Sirunyan, A. M. ; Tumasyan, A. ; Adam, W. ; Taylor, D. ; Woods, N.: Search for supersymmetry with photons in *pp* collisions at $\sqrt{s} = 8$ TeV. In: *Phys. Rev. D* 92 (2015), Oct, 072006. `http://dx.doi.org/10.1103/PhysRevD.92.072006`. – DOI 10.1103/PhysRevD.92.072006

[KST+16] Khachatryan, V. ; Sirunyan, A. M. ; Tumasyan, A. ; Adam, W. ; Asilar, E. ; Bergauer, T. ; Savin, A. ; Sharma, A. ; Smith, N. ; Smith, W. H. ; Taylor, D. ; Woods, N.: Search for pair-produced vectorlike B quarks in proton-proton collisions at $\sqrt{s} = 8$ TeV. In: *Phys. Rev. D* 93 (2016), Jun, 112009. `http://dx.doi.org/10.1103/PhysRevD.93.112009`. – DOI 10.1103/PhysRevD.93.112009