High-Level FPGA-Programmierung mit automatisch generierten Netzwerken von Automaten

Vom Promotionsausschuss der

Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von

Henning Manteuffel

aus

Hamburg

2012

Erster Gutachter: Prof. Dr. Friedrich Mayer-Lindenberg
Zweite Gutachterin: Prof. Dr. Sibylle Schupp

Tag der mündlichen Prüfung: 06.09.2012

Veröffentlichungen

Die Ergebnisse der vorliegenden Arbeit wurden teilweise veröffentlicht:

- H. Manteuffel, C. Bassoy, F. Mayer-Lindenberg,
  The TransC Process Model and Interprocess Communication,
  The 2010 International Conference on Field-Programmable Technology (FPT'10),
  Peking, China, Dezember 2010.

- H. Manteuffel, C. Bassoy, F. Mayer-Lindenberg,
  FPGA-specific Optimizations by Partial Function Evaluation,
  The 2011 Electronic System Level Synthesis Conference (ESLsyn'11),
  San Diego, USA, Juni 2011.

- C. Bassoy, H. Manteuffel, F. Mayer-Lindenberg,
  Sharf: An FPGA-based customizable processor architecture,
  19th International Conference on Field Programmable Logic and Applications (FPL'09),
  Prag, CZ, September 2009.

Kurzfassung

In *Field Programmable Gate Arrays* (FPGAs) steigt die Anzahl der Logikzellen sowie der Umfang von festverdrahteten Komponenten ständig. Um diese zunehmende Komplexität zu beherrschen, gewinnt die Programmierung mit Hochsprachen auf Verhaltensebene statt mit Hardwarebeschreibungssprachen immer mehr an Bedeutung. Viele der bestehenden abstrakten Sprachansätze besitzen keine Paradigmen zur Beschreibung von Parallelität oder zur Integration von vordefinierten Komponenten. Diese sind aber nötig, um Algorithmen auf FPGAs unter Ausnutzung des hohen möglichen Parallelitätsgrades und der vorhandenen fest verdrahteten Schaltungen abzubilden. Andere Sprachen, die diese Eigenschaften bieten, sind kompliziert in der Handhabung aufgrund ihrer Syntax oder der Art und Weise, wie nebenläufige Vorgänge ausgedrückt werden. Außerdem unterstützen sie Konstrukte aus der Mikroprozessorprogrammierung, die auf FPGAs die Parallelität einschränken und nicht optimal implementiert werden können.

In dieser Arbeit wird die Hochsprache TransC vorgestellt, mit der Netzwerke von Zustandsautomaten beschrieben werden, die sich effizient in digitale Schaltungen übersetzen lassen. Dazu wird ein neues Prozessmodell eingeführt sowie Konstrukte zur Interprozesskommunikation und -synchronisation erläutert. TransC erweitert ANSI-C um nur wenige Paradigmen, was die Programmierung sowie die Portierung von existierendem C-Code im Vergleich zu anderen Ansätzen erleichtert. Die zusätzlichen Sprachfeatures dienen hauptsächlich zur Beschreibung von Nebenläufigkeit. Ein Compiler wurde implementiert, der effizienten VHDL-Code auch für Programme erzeugt, die aus mehreren Prozessen bestehen.

Dafür wurde ein Zwischenformat auf Basis eines Kontroll-Datenflussgraphen definiert sowie bestehende Optimierungen zur Steigerung der Schaltungsqualität angepasst. Zur Erhöhung des Durchsatzes sowie zur Verringerung des Ressourcenverbrauchs der generierten Schaltungen wurde eine neue Evaluationsmethode auf Basis der partiellen Funktionsauswertung entwickelt. Diese profitiert von statischen Kontrolldaten, die sich zwischen einzelnen Programmausführungen nicht ändern und bereits zur Kompilierzeit bestimmbar sind. Der Hauptaspekt liegt hierbei in der Sammlung von Zwischenergebnissen während der Evaluierung, die dann für hardware-spezifische Optimierungen verwendet werden.

Anhand quantitativer Untersuchungen der Syntheseergebnisse sowohl für einfachere als auch für komplexere Algorithmen, wie z. B. dem Secure Hash Algorithm 1 (SHA1[28]), CRC-Berechnungen oder Farbraumkonvertierungen, wird gezeigt, dass die durchschnittliche Fläche und Laufzeit um über 40 % im Vergleich zu anderen aktuellen Synthesetools reduziert werden kann. Vor allem Anwendungen mit einem hohen Anteil an Kontrollfluss sowie Speicherzugriffen und Schleifen werden effizient in digitale Schaltungen umgesetzt. Durch die partielle Funktionsauswertung konnte die Laufzeit von speicherintensiven Applikationen um bis zu 30% gesenkt werden.

TransC eignet sich nicht nur für die Umsetzung und Beschleunigung von Algorithmen oder Programmausschnitten, sondern auch für die Entwicklung von kompletten Systemen.

Die Tauglichkeit der Sprache und des Compilers für solche Anwendungen wird mit der Entwicklung eines Prototypen für die Verarbeitung und Modifikation von Audiosignalen gezeigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Anforderungen an elektronische Systeme und integrierte Schaltungen steigen ständig. Während auf der einen Seite die Schaltungen umfangreicher werden und die Komplexität zunimmt, werden auf der anderen Seite Eigenschaften wie geringer Platzbedarf oder ein geringer Stromverbrauch wichtiger. Die Systeme müssen eine hohe Zuverlässigkeit aufweisen und zeitkritische Anwendungen müssen mit hohen Geschwindigkeiten arbeiten, um geforderte Zeitschranken einzuhalten. Erschwerend kommt hinzu, dass die Entwicklungszeit aufgrund des Konkurrenzdrucks und der geringer werdenden Lebenszyklen von Produkten immer kürzer wird.

Produktivität

Die Zahl der Transistoren, die in den aktuellsten Technologien integriert werden kann, liegt höher als die Produktivität der Entwickler, was als *Design Productivity Gap* bezeichnet wird. Dieser Trend wird seit Ende der 90er Jahre beobachtet und konnte durch neue Entwurfsmethoden abgemildert werden, wie Abbildung 1.1.1 zeigt. Danach beliefen sich die durchschnittlichen Entwurfskosten im Jahr 2005 für einen Chip auf etwa 18 Millionen US-Dollar. Wären diese Schaltungen nur nach den Methodologien auf dem Register-Transfer-Level (RTL) entworfen worden, so lägen die Kosten bei knapp 900 Mio. US-Dollar. RTL ist eine Abstraktionsebene, die das Verhalten von synchronen Schaltkreisen durch den Signalfluss zwischen Registern beschreibt. Durch weitere Automatisierungen und Verbesserungen beim Entwicklungsprozess, die über den Register-Transfer-Level hinausgehen, können demnach auch in naher Zukunft die Kosten konstant gehalten werden. Nach der ITRS-Roadmap [32] verdoppelt sich die Leistungsfähigkeit der Produktionstechnologie sowie die Produktivität im Hardware- und Software-Design alle 36 Monate. Die Produktivität im Hardwareentwurf wurde in den letzten Jahren durch die Verwendung von Mehrkernkomponenten und Speicher verbessert und es wurde neue Funktionalität in zusätzlicher Software bereitgestellt. Allerdings verdoppelt sich die Produktivität in der Erstellung hardwareabhängiger Software nur alle fünf Jahre, wodurch wieder eine neue Lücke im Entwurfsprozess entsteht.

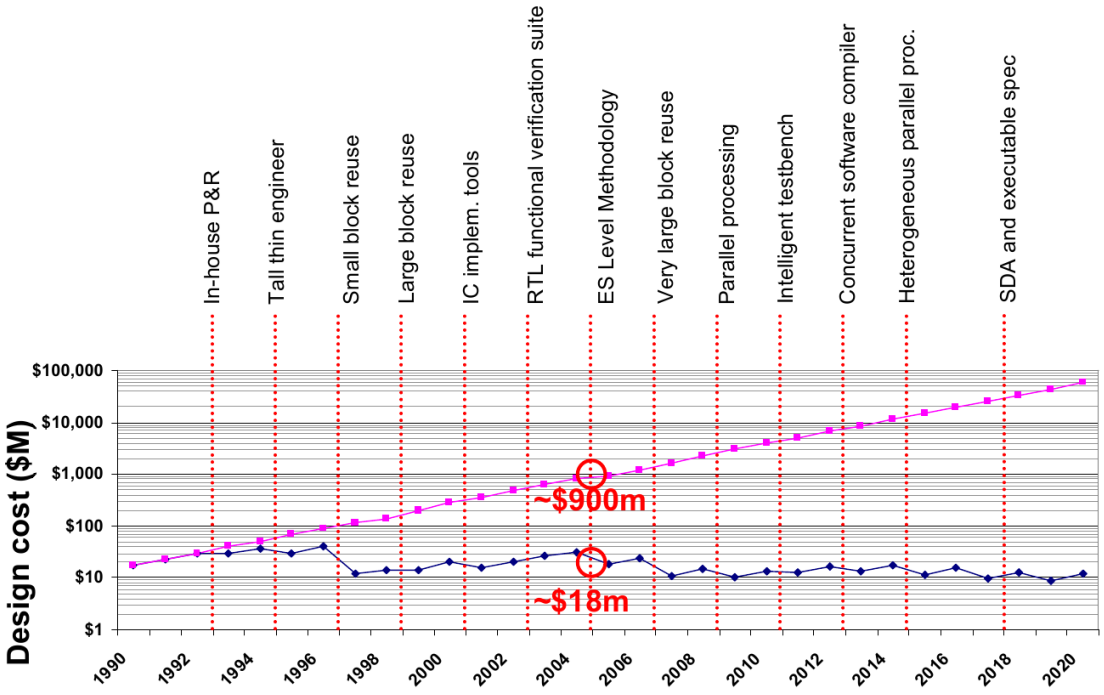


Abbildung 1.1.1: Verlauf der Entwurfskosten mit und ohne Verbesserungen beim Entwicklungsprozess (Quelle: [31])

Aus diesem Grund wurde in der Roadmap ein zusätzliches *Design-Gap* eingeführt, um in der Produktivität auch die Softwareanforderungen modellieren zu können.

Im Entwurfsfluss gibt es bereits seit Jahrzehnten verschiedene Abstraktionsebenen wie Blockdiagramme, Zustandsdiagramme oder Programmmodelle, allerdings noch mit wenig Unterstützung von Automationswerkzeugen [32]. Dies muss sich ändern, wenn die notwendigen Fortschritte in der Produktivität erreicht werden sollen und sich auch die Produktivitätslücke unter Berücksichtigung der Software eines Systems nicht weiter vergrößern soll. Hierfür muss die Spezifikation sowie die Implementierung und Verifikation von Systemen vereinfacht werden durch die Einführung weiterer Abstraktionsebenen über dem Register-Transfer-Level sowie durch die Verwendung von automatisierten Werkzeugen.

FPGAs

In den letzten Jahren werden in bestimmten Systemen zunehmend FPGAs (*Field-Programmable Gate Arrays*) statt ASICs (*Application-Specific Integrated Circuits*) verwendet. FPGAs sind integrierte Schaltungen, auf denen beliebige logische Schaltungen implementiert werden können [61]. Sie sind aus Blöcken (*Configurabe Logic Blocks*, CLBs) zusammengesetzt, die u.a programmierbare Lookup-Tabellen (LUTs) enthalten. Abhängig von der Zahl der verfügbaren Eingänge kann in LUTs jede beliebige n-stellige Binärfunktion realisiert werden. Die CLBs sind durch programmierbare Signalleitungen miteinander verbunden. Für längere Signalwege liegen zwischen den Blöcken gitterförmig angeordnete

Busstrukturen.

Im Vergleich zu ASICs hat der Entwickler die Möglichkeit, eine große Anzahl von digitalen Schaltungen zu niedrigen Kosten zu erstellen und auch nachträglich Fehler zu korrigieren, da nicht jedes Mal eine neue Schaltung gefertigt werden muss, sondern lediglich das FPGA umkonfiguriert wird. Auf der anderen Seite sind FPGA-Schaltungen größer, haben eine niedrigere Taktfrequenz und verbrauchen mehr Energie [64]. Ursprünglich wurden FPGAs zum Testen von Schaltungen verwendet, die später in ASICs implementiert wurden. Später wurden sie auch als Verbindungslogik (*Glue Logic*) zwischen verschiedenen Schnittstellentypen und Bussen oder zur Formatkonvertierung eingesetzt. Aufgrund der stetig wachsenden Anzahl von Logikzellen in FPGAs konnte der traditionelle Einsatz erweitert und auch zeitkritische oder rechenaufwändige Funktionen, beispielsweise von Signalverarbeitungsalgorithmen, in FPGAs ausgelagert werden. Aus diesem Grund eignen sie sich mittlerweile für Anwendungsgebiete, die bisher von digitalen Signalprozessoren (DSPs) abgedeckt wurden. Da die Recheneinheiten von DSPs auf häufig vorkommende Programmkonstrukte in Signalverarbeitungsalgorithmen spezialisiert sind, besitzen neuere FPGA-Generationen neben den Logikzellen auch hartverdrahtete Funktionseinheiten wie Block-RAMs, Multiplizierer oder Einheiten für DSP-Berechnungen (z.B. DSP48 bei Xilinx), was die Wettbewerbsfähigkeit zu ASICs und Prozessoren erhöht. Die dedizierten Einheiten sind im Vergleich zu in Logikzellen implementierten Komponenten schneller, kleiner und energieeffizienter. FPGA-Schaltungen haben bei zeitkritischen Anwendungen Vorteile gegenüber Prozessoren, da die Algorithmen in einem höheren Parallelitätsgrad und auf applikationsspezifischen gepipelinten Datenpfaden abgearbeitet werden können. Zudem können die Ausführungseinheiten auf applikationsspezifische Datentypen angepasst werden und sind nicht auf Standardtypen beschränkt. Allerdings laufen sie aufgrund der Rekonfigurierbarkeit mit einer geringeren Taktfrequenz. In [26] wird gezeigt, dass FPGA-Schaltungen dennoch mehrere Größenordnungen schneller arbeiten können als von-Neumann-Prozessoren. Die Erzeugung von applikationsspezifischen Schaltungen für FPGAs ist wesentlich aufwändiger und fehleranfälliger als die Programmierung von Mikroprozessoren. Da es sich bei FPGAs um digitale Schaltungen handelt, ist der Entwurfsfluss ähnlich zu dem von ASICs. FPGAs werden in Hardwarebeschreibungssprachen auf dem Register-Transfer-Level konfiguriert und können auch komplette Prozessoren enthalten. Auch hier hat man ein System, das aus Hardware und Software besteht, weshalb das oben genannte *Productivity Gap* wieder zur Geltung kommt.

Synthese

Durch die Anwendung der Synthese auf hohen Abstraktionsebenen kann bei der Entwicklung von digitalen Schaltungen eine wesentliche Produktivitätssteigerung erzielt werden. Manna [70] definiert die Synthese als eine formale Konstruktion von Implementierungen, die gegenüber der Spezifikation garantierte Korrektheit aufweisen. Im Gegensatz zur vollautomatischen Kompilierung wird hier von einem Prozess ausgegangen, der die einzelnen

Schritte bis zum Endresultat unter menschlichem Einfluss durchführt. Dabei werden neben der eigentlichen Spezifikation weitere Randbedingungen gegeben, die in den einzelnen Zwischenstufen der Synthese ggf. korrigiert werden, wenn die Ergebnisse nicht die geforderten Eigenschaften besitzen. Bei dem Entwurf von digitalen Systemen gibt es sechs verschiedene Syntheseschritte, die in der Regel aufeinanderfolgend ausgeführt werden: Systemsynthese, Architektursynthese, RTL-Synthese, Logiksynthese, Platzierung und Verdrahtung. Dabei setzen diese Schritte auf verschiedenen Abstraktionsebenen auf und überführen sie in die nächst niedrigere.

Systemsynthese

Bei der Systemsynthese wird eine Verhaltensbeschreibung, die aus Algorithmen, Prozeduren oder Prozessen aufgebaut ist, auf eine strukturelle Beschreibung abgebildet, dessen Grundelemente aus komplexen Hardwareschaltungen wie FPGAs, ASICs, Prozessoren, Speichern und Bussen bestehen. Die Kommunikation an den Schnittstellen zwischen Hardware und Software wird explizit modelliert. Die Systemsynthese setzt sich aus mehreren Teilproblemen zusammen, wie der Auswahl von Systemkomponenten (Allokation) und der Abbildung der einzelnen Algorithmen auf diese (Bindung). Dabei sind bestimmte Randbedingungen wie Zeit- oder Ressourcenbegrenzungen einzuhalten. Bei der Einhaltung von Zeitschranken muss eine Ablaufplanung durchgeführt werden. Diese gibt die Zeitpunkte vor, in der bestimmte Aufgaben zu starten sind, was wiederum die Komponentenauswahl beeinflusst.

Nach dem Schritt der Systemsynthese ist bekannt, welche Komponenten in Hardware und welche in Software implementiert werden. Diese werden dann weiter mit der Architektursynthese bzw. der Softwaresynthese (Kompilierung) verfeinert. Dabei müssen die Schnittstellen und die Softwaretreiber bereits feststehen. Wichtig auf der Systemebene sind gute Schätzungsverfahren, um die Entwurfsqualität zu beurteilen und eine möglichst gute Systemstruktur für die weiteren Schritte zu erzeugen. Die Automatisierung der Systemsynthese ist aktueller Forschungsgegenstand und wird im Bereich des Hardware-/Software-Codesigns untersucht.

Architektursynthese

Der Architektursynthese liegt eine Verhaltensbeschreibung zu Grunde, die z.B. aus einem Kontroll- und Datenfluss besteht. Die Datenknoten modellieren elementare arithmetisch-logische Operationen wie die Multiplikation oder die Addition und stellen ihre Abhängigkeiten in Bezug zu Vorgänger- und Nachfolgeoperationen dar. Der Kontrollpfad wird auf ein Steuerwerk abgebildet und der Datenfluss, der durch die Operationen entsteht, auf ein Rechenwerk. Dabei müssen wie auch während der Systemsynthese bei der Allokation Komponenten ausgewählt werden, die die Operationen ausführen können. Diese bestehen jetzt nicht mehr aus komplexen Teilsystemen, sondern aus Speichern und Funktionseinheiten wie Multiplizierern und Addierern. In der Ablaufplanung werden die Zeitpunkte bestimmt, an

denen die Operationen im Datenfluss ausgeführt werden. Während der Bindung werden Operationen und Variablen allokierten Funktionseinheiten und Speichern zugewiesen. Das Steuerwerk wird meist in Form eines Automaten (engl. *finite state machine*, FSM) oder eines programmierbaren Controllers implementiert. Auch hier müssen bestimmte Randbedingungen berücksichtigt werden, da Schaltungsfläche und Verarbeitungszeit wesentlich von Optimierungen und Entscheidungen auf dieser Abstraktionsebene abhängen. Das Ergebnis der Architektursynthese ist eine Beschreibung auf Register-Transfer-Level, bei der die Zahl der Zeitschritte zur Abarbeitung der Operationen der Verhaltensbeschreibung sowie teilweise die verwendeten Funktionseinheiten feststehen.

RTL-Synthese

Bei der RTL-Synthese wird die Strukturbeschreibung, die in Form eines Automaten und eines Datenpfades mit Registern, Multiplexern und Funktionseinheiten vorliegt, in eine Beschreibung auf der Logikebene transformiert, die aus kombinierten Logikblöcken, Booleschen Ausdrücken und Speicherelementen, wie Registern, besteht. Gesteuert werden die Komponenten von dem Automaten, der mittels Schaltwerksynthese in ein Schaltwerk auf Logikebene überführt wird. Hier kann zwischen verschiedenen Zustandscodierungen und Architekturen des Steuerwerks gewählt werden. Zudem kann auch das Steuerwerk auf Größe bzw. Geschwindigkeit ausgerichtet werden. Ein Optimierungziel der RTL-Synthese ist die möglichst gleichmäßige Verteilung der Logik zwischen den Registern, um eine möglichst hohe Taktfrequenz zu erzielen. Die Schaltungsbeschreibung auf Registertransferebene ist am gängigsten, auch Synthesetools wie ISE WebPACK von Xilinx [55] oder Alteras Quartus II [21] setzen mit Hardwarebeschreibungssprachen wie VHDL[91] oder Verilog[82] auf dieser Ebene auf.

Logiksynthese

Erster Schritt der Logiksynthese ist die Minimierung der während der RTL-Synthese entstandenen Booleschen Ausdrücke, beispielsweise mit dem Quine-McCluskey-Verfahren, und die Optimierung des Steuerwerks. Anschließend werden die Ausdrücke auf Standardzellen abgebildet, die in der jeweiligen Technologie-Bibliothek zur Verfügung stehen. Die Logik wird somit in eine Strukturbeschreibung aus Bibliothekszellen wie Gatter, Register oder komplexere Einheiten wie Multiplizierer oder Speicher transformiert (*Technology Mapping*). Die Strukturbeschreibung mit ihren Verbindungen wird auch als Netzliste bezeichnet. Da es mehrere Möglichkeiten für die Abbildung einer Booleschen Funktion gibt, ist es ein Ziel, die kostengünstigste zu finden, auch hier kann z.B. in Richtung Ressourcenverbrauch oder Geschwindigkeit optimiert werden.

Platzierung und Verdrahtung

Beim ersten Schritt, der Platzierung, wird bestimmt, an welcher Stelle im ASIC die Einheiten der Netzliste positioniert werden oder im FPGA, auf welche physikalische Ressource sie

abgebildet werden. Im Vordergrund steht dabei, stark vernetzte Komponenten oder solche mit zeitkritischen Verbindungen möglichst dicht nebeneinander zu setzen. Dabei kann eine grobe Aufteilung der Netzliste in einzelne Blöcke vorgegeben werden, die in bestimmten Bereichen des Chips platziert werden sollen (*Floorplanning*). Anschließend werden die Zellen miteinander verdrahtet. Hier müssen alle nötigen Verbindungen unter Berücksichtigung der Randbedingungen und der Limitierungen des Herstellungsprozesses gesetzt werden. Beim FPGA werden bei der Verdrahtung die vorhandenen Leitungen und Busse benutzt und nur die Verbindungen konfiguriert. Da die Platzierung und Verdrahtung ein NP-schweres Problem ist, werden heuristische Algorithmen angewendet, um möglichst nah am optimalen Ergebnis zu sein. Da nach diesem Prozess alle Verdrahtungswege bekannt sind, können die Signalverzögerungen durch kapazitive Leitungslasten genau ermittelt und zur Simulation in die Netzliste eingesetzt werden. Das Ergebnis der Platzierung und Verdrahtung bei ASICs ist das Layout, eine geometrische Beschreibung der Chipstruktur, nach der die Masken zur Produktion angefertigt werden. Bei FPGAs wird nach der Platzierung und Verdrahtung eine Binärdatei zur Konfiguration des FPGA erzeugt.

1.2 Stand der Synthesetools und Zielstellung

Um die steigenden Anforderungen beim Schaltungsentwurf zu erfüllen und nicht am oben genannten *Productivity Gap* zu scheitern, müssen Schaltungen auf immer höheren Abstraktionsebenen entworfen werden. Im Hinblick auf das *Productivity Gap*, das sowohl Hardware als auch Software berücksichtigt, müsste der Entwurfsprozess bereits bei der Systemsynthese beginnen. Hier wird auch die Erstellung hardwarenaher Software wie Treiber und Kommunikationsprotokolle automatisiert. Bereits Ende der 80er Jahre setzte sich die RTL-Synthese durch. Obwohl seit dieser Zeit auch die Architektursynthese, also der Entwurf von digitalen Systemen auf der Verhaltensebene vorgeschlagen wird [74], hat diese bis heute noch nicht den Stellenwert der RTL-Synthese erreicht. Diese ist jedoch eine Voraussetzung für den Entwurf auf Systemebene [32].

Innerhalb der letzten Dekade sind eine Vielzahl von Eingabesprachen entstanden, die auf C[12], SystemC[41] oder Dialekten davon basieren und durch entsprechende Programme in Hardwarebeschreibungssprachen auf der Registertransferebene wie VHDL[91] oder Verilog[82] umgewandelt werden. Die bekanntesten akademischen Synthesewerkzeuge sind ROCCC[43], Spark[44], Hybridthreads[95] und C-to-Verilog[90, 89]. Die Tools sind in der Lage, parallele Instruktionen aus sequentiellem C-Code zu erkennen und so im Ablaufplan anzuordnen, dass sie in der erzeugten Schaltung gleichzeitig ausgeführt werden. Da diese Art der Parallelisierung sehr feingranular ist und schnell an ihre Grenzen stößt, unterstützt Hybridthreads eine POSIX-Thread-API[17], womit explizit parallele Prozesse beschrieben werden können. Im kommerziellen Bereich gibt es Sprachen wie Handel-C[40], Impulse-C[53], Catapult-C[39] oder AutoESL[54], die ebenfalls parallele Prozesse und Mechanismen zur Interprozesskommunikation unterstützen. Trotz der zusätzlichen Komplexität und der Programmierparadigmen sind auch diese Sprachen ANSI-C-kompatibel und unterstützen

meist den vollen Sprachumfang. Das hat den Vorteil, dass bestehende Programme schnell umgesetzt werden können und auch die Erstellung von neuem Code erleichtert wird, da dieser mit herkömmlichen Software-Tools getestet und debugged werden kann. Auf der anderen Seite müssen die zusätzlichen Spracheigenschaften in bestimmte Funktionsaufrufe, Pragmas oder spezielle Kommentarformate gekapselt werden, was die Implementierung und die Übersicht erschwert. Somit müssen auch bestehende Programme angepasst werden, da zwar Software-Compiler die neuen Sprachen verarbeiten, die Hardware-Compiler aber aus unmodifizierten Programmen keine effizienten Schaltungen generieren können. In ROCCC z.B. sind Argumente und Rückgabewerte in Datenstrukturen einzufassen, bevor diese auf Ein- und Ausgänge der Schaltung abgebildet werden können. Um in Impulse-C auf gemeinsame Arrays zugreifen zu können, müssen spezielle Funktionsaufrufe verwendet werden.

Andere Compiler wie SR[35] oder p-Nets[72, 73] stellen eine neue Sprache zur Verfügung, die auf das jeweilige Programmiermodell angepasst ist. Die Benutzung einer Sprache mit einer komplett neuen Syntax erschwert die Portierung von existierender Software oder auch die Erstellung von neuen Programmen aufgrund der Einarbeitungszeit. Auf der anderen Seite führt die vollständige Kompatibilität zum ANSI-C-Standard zu ineffizienteren Schaltungen. Sprachfeatures, wie etwa generische Zeiger, die effizient auf Maschinenbefehle in Mikroprozessoren abgebildet werden können, verursachen Probleme bei der direkten Implementierung in Hardware. Die Verwendung von generischen Pointern würde einen gemeinsamen Speicherbereich für bestimmte Arrays eines Programms voraussetzen, da zur Kompilierzeit nicht immer ermittelt werden kann, auf welches Array genau zu einem bestimmten Programmpunkt gerade gezeigt wird. Wenn sich Arrays einen gemeinsamen Speicher teilen, können diese nicht mehr parallel gelesen oder geschrieben werden. Gerade die niedrige Effizienz im Vergleich zu auf der Registertransferebene erstellten Schaltungen wird in [32] als Grund für die geringe Akzeptanz der Architektursynthese aufgeführt.

Mit dieser Arbeit soll ein Teil der oben genannten Nachteile bei der Architektursynthese beseitigt und die Syntheseergebnisse verbessert werden. Dafür wird TransC, ein Synthesetool für die Generierung von digitalen Schaltkreisen in FPGAs vorgestellt. Um den hohen Parallelitätsgrad auszunutzen, den FPGAs zur Verfügung stellen, liegt das Hauptaugenmerk auf der Entwicklung eines Prozessmodells, das effizient auf Hardwarekomponenten abgebildet werden kann. Die Eingabesprache ist an C angelehnt und lediglich im Kern kompatibel zu ANSI-C, um bestehende Algorithmen schnell portieren zu können. Auf der anderen Seite werden neue Sprachfeatures durch neue Sprachkonstrukte dargestellt. Programmierparadigmen, die zu ineffizienter Hardware führen, werden weggelassen oder so umgeändert, dass sie effizient vom Synthesetool umgesetzt werden können. Durch die Anpassung bestehender Optimierung und die Entwicklung von neuen Methoden soll die Leistungsfähigkeit der resultierenden Hardwaremodule weiter erhöht werden. Die generierte Sprache ist VHDL, das auf der Registertransferebene vorliegt und das von Tools auf der Ebene der RTL-Synthese weiterverarbeitet wird. Der VHDL-Code soll lesbar sein und, wenn keine Optimierungen angewendet werden, sollen Konstrukte aus der Eingabesprache

leicht in der Hardwarebeschreibung wiederzufinden sein. Die Modulschnittstellen sind relativ simpel gehalten, so dass auch bestehende VHDL-Module durch einfache Wrapper in den Code eingefügt werden können. Ein- und Ausgabeleitungen für die Kommunikation mit Peripherie außerhalb des FPGAs werden ebenfalls unterstützt. Neben dem Prozessmodell ist ein Zeitmodell vorhanden, das die Implementierung von Echtzeitanwendungen ermöglicht. Die Prozess- und Zeitmodelle sowie die entwickelten und implementierten Synthesealgorithmen werden im Detail vorgestellt, um auch in andere Werkzeuge integriert werden zu können.

1.3 Eigener Beitrag

Diese Arbeit deckt mehrere Bereiche in der Architektursynthese ab, wie die Entwicklung einer geeigneten Eingabesprache mit den zugehörigen Prozess- und Zeitmodellen sowie die interne Repräsentation dieser Sprache innerhalb des Rechners. Außerdem werden Optimierungsalgorithmen und Methoden zur Erzeugung der Ausgabesprache vorgestellt. Folgende Teile wurden dabei behandelt und untersucht:

- Entwicklung eines hierarchischen Prozessmodells, das hinreichend viele Arten der Prozesssynchronisation und Interprozesskommunikation abdeckt und dabei effizient auf Hardware abgebildet werden kann.

- Entwicklung einer C-ähnlichen Sprache, die auf dem Prozessmodell aufbaut und deren Sprachkonstrukte effizient auf Automaten und Hardwarekomponenten abgebildet werden können.

- Entwicklung eines Zwischenformats, das leicht optimiert und modifiziert werden kann und das durch Erweiterungen auch für die Repräsentation von anderen Sprachen geeignet ist.

- Anpassungen von bestehenden Optimierungsalgorithmen an das Zwischenformat und Entwicklung von neuen Optimierungen zur Verbesserung der Schaltungsqualität.

- Implementierung eines VHDL-Backends auf Basis des Zwischenformats.

- Es wird beschrieben, wie das Synthesesystem effizient für eine schnelle Entwicklung von Hardwarekomponenten und auch für die Implementierungen von Anwendungen eingesetzt werden kann und auch bestehende Hardwarekomponenten eingebunden werden können.

- Implementierung eines Audio-Effektgerätes komplett auf FPGA-Basis mit Hilfe des hier vorgestellten Synthesetools.

1.4 Struktur der vorliegenden Arbeit

Kapitel 2 befasst sich mit den notwendigen Grundlagen zur Architektursynthese und stellt verschiedene Datenstrukturen für die Repräsentation von Programmen vor. Außerdem werden bekannte Prozessmodelle und Optimierungsalgorithmen behandelt und diese Arbeit in den gesamten Entwurfsfluss für digitale Schaltungen eingeordnet. Anhand dieser Grundlagen wird in Kapitel 3 ein eigenes Prozessmodell entwickelt, das bestimmte Anforderungen für die Generierung von digitalen Schaltungen erfüllt und effizient umsetzbar ist. Dieses Prozessmodell wird in eine an C angelehnte Programmiersprache integriert, dessen Konzepte in Kapitel 4 beschrieben sind. Kapitel 5 behandelt den Zwischencode, durch den die Sprache dargestellt wird. In Kapitel 6 wird beschrieben, wie bestimmte Optimierungen, die in 2.4 vorgestellt wurden, auf den Zwischencode angepasst werden können. Außerdem wird ein neuer Optimierungsalgorithmus erläutert, der auf der partiellen Evaluierung von Funktionen beruht. Die Erzeugung des VHDL-Codes wird in Kapitel 7 behandelt. Die Praxistauglichkeit des Synthesetools wird durch Benchmarks und Beispielimplementierungen in Kapitel 8 aufgezeigt. Die Zusammenfassung dieser Arbeit sowie die Diskussion der Ergebnisse und der Ausblick auf weiterführende Arbeiten befindet sich in Kapitel 9. Im Anhang sind Beispiele und weitere Dokumentationen über die Sprache beigefügt.

Kapitel 2

Grundlagen

In den nächsten Abschnitten werden Themen wie die Architektursynthese oder die dort verwendeten Zwischenformate und Optimierungen ausführlicher behandelt, um die notwendigen Grundlagen für die danach folgenden Kapitel zu schaffen. Außerdem werden bestehende Prozessmodelle beschrieben.

2.1 Phasen der Architektursynthese

Wie in der Einführung erwähnt, wird bei der Architektursynthese eine Verhaltensbeschreibung, die aus arithmetisch-logischen Operationen und Kontrollanweisungen besteht, auf eine strukturelle Beschreibung mit einem Datenpfad und einem Steuerwerk abgebildet ([97], S. 218). Der Datenpfad besteht aus Funktionseinheiten wie Multiplizierern, ALUs oder Komparatoren. Weitere Bestandteile sind Kommunikationseinheiten und Speicher wie Block-RAMs, FIFOs (engl. *first in first out*) oder einfache Register. Die Ressourcen des Datenpfades sind über Multiplexer oder Busse miteinander verbunden, für die das Steuerwerk entsprechende Kontrollsignale erzeugt, so dass die Schaltung die entsprechenden Berechnungen im gleichen zeitlichen Ablauf durchführt wie in der Verhaltensbeschreibung festgelegt worden ist. Dabei kann die Verhaltensbeschreibung durch bestimmte Optimierungen vor der Schaltungsgenerierung modifiziert worden sein.

Die Architektursynthese lässt sich in mehrere Schritte aufteilen, die in Abb. 2.1.1 dargestellt sind. Ausgangspunkt ist die Verhaltensbeschreibung eines Algorithmus. Wenn diese in Quellcodeform vorliegt, so muss sie zunächst in der lexikalischen, der syntaktischen und der semantischen Analyse in ein Zwischenformat umgewandelt werden, das sich für die rechnerinterne Speicherung und Modifikation eignet. Liegt die Verhaltensbeschreibung nicht direkt als Quellcode vor, sondern wurde z.B. von vorhergehenden Synthesestufen wie der Systemsynthese übergeben, so fallen diese Analysephasen weg. Auf dem Zwischenformat werden Optimierungen und Transformationen nach bestimmten Randbedingungen durchgeführt. Dabei kann sich auch der grundlegende Aufbau des Zwischenformats ändern. Anschließend werden die fundamentalen Grundaufgaben der Synthese [97], Allokation, Ablaufplanung und Bindung gelöst. Diese Grundaufgaben sind unabhängig vom Verfeinerungsgrad und

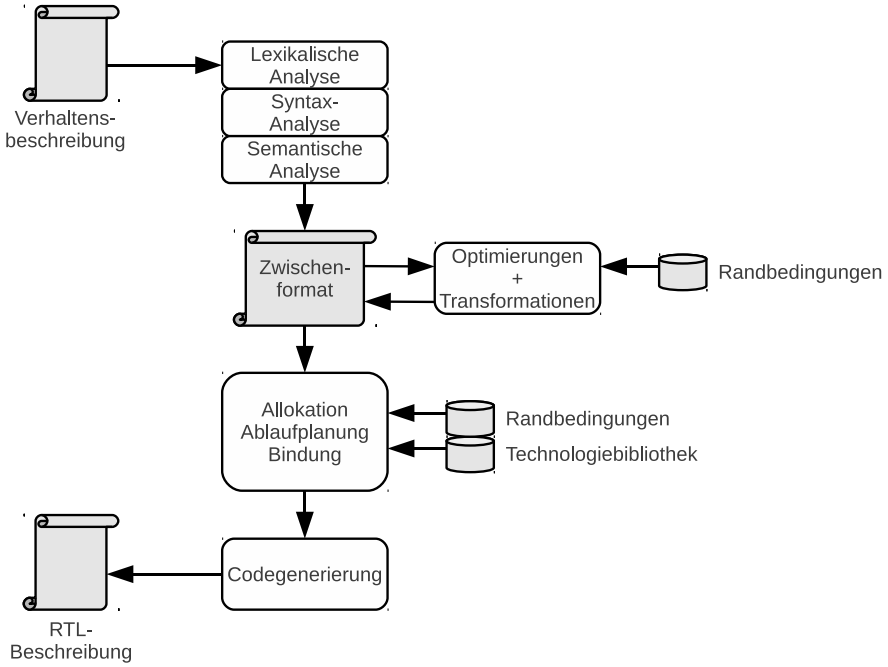


Abbildung 2.1.1: Phasen der Architektursynthese im TransC-Compiler

lassen sich neben der Architektursynthese auch auf die übrigen Abstraktionsebenen wie die System-, RTL- oder Logikebene übertragen. Bei der Allokation werden Funktionseinheiten aus einer Technologiebibliothek ausgewählt, bei der Ablaufplanung erhalten die Operationen aus dem Zwischenformat Zeitschritte, in denen sie später ausgeführt werden und bei der Bindung wird bestimmt, auf welchen Instanzen der allokierten Ressourcen diese Operationen ausgeführt werden. Die Schritte hängen erneut von Randbedingungen und den Komponenten in der verwendeten Technologiebibliothek ab. So macht es beispielsweise einen Unterschied, ob die Bibliothek parallele oder serielle Multiplizierer besitzt oder ob Zeit- bzw. Größenvorgaben einzuhalten sind. Bei der Allokation, der Ablaufplanung und der Bindung gibt es keine feste Ausführungsreihenfolge. Diese drei Schritte sind eng miteinander verbunden und beeinflussen sich gegenseitig. Eine in den Randbedingungen gegebene Allokation beeinflusst z.B. die Ablaufplanung, da die Auswahl an Ressourcen bereits begrenzt ist und zwei Operationen nicht gleichzeitig auf derselben Ressource ausgeführt werden können. Bei einer gegebenen oberen Zeitschranke muss zunächst die Ablaufplanung durchgeführt werden, wodurch dann wieder die Allokation und die Bindung eingeschränkt wird. Die Grundaufgaben wurden zunächst in vereinfachender Weise separat betrachtet [34]. Später löste man diese z.B. in Arbeiten wie [66] gemeinsam. Nach der Durchführung der oben genannten Schritte wird eine Beschreibung auf Registertransferebene implementiert, wenn alle Randbedingungen eingehalten worden sind. Die folgenden Abschnitte beschreiben die in Abb. 2.1.1 aufgeführten Phasen im Detail.

Lexikalische Analyse Bei der lexikalischen Analyse [97] wird der in Textform vorliegende Quellcode von links nach rechts eingelesen und die dabei entstehende Zeichenfolge in

eine Folge von Symbolen (*Tokens*) umgewandelt. Mit Hilfe von regulären Ausdrücken ist für jeden Symboltyp definiert, welche Zeichenfolgen zu diesem passen. Das für die lexikalische Analyse zuständige Modul wird auch als Scanner bezeichnet. Jedem Symbol aus der Folge kann seine ursprüngliche Zeichenkette, für die es steht, weiterhin zugeordnet werden. Die Analyse des Ausdrucks '`res=36*value+457`' könnte in die Symbolfolge '`ident op const op ident op const`' umgewandelt werden, wenn Zahlen das Symbol `const`, Zeichenketten aus Buchstaben das Symbol `ident` und einfache mathematische Operatoren das Token `op` haben. In dieser Arbeit wird der Scanner mit Hilfe des Scannergenerators Flex[87] erzeugt.

Syntaktische Analyse Bei der Syntaxanalyse[97] wird versucht, in den Symbolfolgen des Scanners Sätze zu erkennen und diese zusammenzufassen. Welche Sätze gültig sind und welche nicht, ist zuvor über eine kontextfreie Grammatik $G(V_N, V_T, P, S)$ definiert worden, die aus Terminalsymbolen V_T, Nichtterminalen V_N, einem Startsymbol S und Produktionsregeln P besteht. V_T enthält Symbole, die der Scanner zurückgibt, V_N hingegen Symbole, die wiederum aus Terminalsymbolen zusammengesetzt sind. Können Symbolfolgen nicht durch die Grammatik beschrieben werden, so wird ein Syntaxfehler ausgegeben. Das Modul für diese Analyse wird auch als Parser bezeichnet. Die Ausgabe des Parsers ist ein sogenannter Syntaxbaum, der den eingelesenen Quellcode in einer baumförmigen Struktur mit den Symbolen aus der Grammatik darstellt. In dieser Arbeit wird der Parser mit Hilfe des Parsergenerators Bison[51] erstellt. Der Syntaxbaum wird genauer in Abschnitt 2.2.1 erläutert.

Semantische Analyse Die semantische Analyse[97] folgt der Syntaxanalyse und überprüft Eigenschaften eines Programms, die sich nicht durch eine Grammatik beschreiben lassen. Beispielsweise können in einem Programm benutzte Variablen nicht deklariert worden sein bzw. doppelte Deklarationen vorkommen usw. Bei dieser Analyse werden die Knoten des bestehenden Syntaxbaums mit zusätzlichen Eigenschaften versehen, wie z.B. Typinformationen zu Variablen oder Symboltabellen. Solch ein Syntaxbaum wird als dekorierter Syntaxbaum bezeichnet.

Transformationen und Optimierungen Aus dem Syntaxbaum wird in der Regel ein Zwischenformat erzeugt, das leicht generierbar, für Optimierungen gut zu modifizieren sowie unabhängig von der Eingabesprache und der Zielausgabe ist. Durch die Loslösung von der Eingabesprache und der Zielausgabe wird erreicht, dass das Zwischenformat auch aus anderen Sprachen generiert und für verschiedene Zielarchitekturen verwendet werden kann. In dieser Arbeit wird aus einer Eingabesprache ein Kontroll-Datenflussgraph (CDFG) generiert, dessen Eigenschaften in 2.2.4 kurz erläutert werden. Die ausführliche Beschreibung steht in Kapitel 5. In Optimierungsschritten wird versucht, das Programm zu verändern, z.B. mit dem Ziel, die Laufzeit zu verringern oder Ressourcen einzusparen.

```
1 while(i<n) {
2     sum += a[i];
3     ++i;
4 }
```

Abbildung 2.1.2: C-Code für eine Aufsummierung eines Arrays

Allokation Die Allokation ordnet jedem Ressourcentypen die Anzahl verfügbarer Instanzen zu. Bei der Architektursynthese sind diese Typen funktionale Einheiten aus einer Technologiebibliothek wie Multiplizierer, Addierer oder auch Speicher und Busse. Wenn R die Menge aller Ressourcetypen ist, dann ist die Allokation eine Funktion α, die jedes $r \in R$ auf eine natürliche Zahl N_0 abbildet.

Stünden für das Beispiel in Abb. 2.1.2 die vier Ressourcetypen Komparator r_C, Addierer r_A, Multiplizierer r_M und Speicher r_S zur Verfügung ($R = \{r_C, r_A, r_M, r_S\}$), so wäre eine mögliche Allokation $\alpha(r_C) = 1$, $\alpha(r_A) = 2$, $\alpha(r_M) = 0$ und $\alpha(r_S) = 1$.

Ablaufplanung Bei der Ablaufplanung (engl. *Scheduling*) werden für die einzelnen Operationen Startzeiten unter Berücksichtigung der Datenabhängigkeiten festgelegt. Alle Operationen, dessen Ergebnis als Operand für eine auszuführende Operation o verwendet wird, müssen bereits abgeschlossen sein. Die Ablaufplanung wird mit Hilfe einer Funktion τ beschrieben, die jede Operation o auf einen Zeitschritt abbildet. Neben den Datenabhängigkeiten muss auch berücksichtigt werden, dass die Zahl der allokierten Ausführungseinheiten relativ gering im Vergleich zur Anzahl der Operationen in einem Algorithmus ist und zu einem Zeitpunkt eine Ausführungseinheit nicht von zwei Operationen gleichzeitig belegt sein kann. In [97] z.B. werden mehrere Algorithmen zur Ablaufplanung vorgestellt, die sich hinsichtlich der Komplexität und der Qualität der Ergebnisse unterscheiden.

Betrachtet man die Operationen im Schleifenkörper von Abb. 2.1.2 ohne den Vergleich, so könnte die Inkrementierung des Schleifenzählers o_3 und der Speicherzugriff o_1 im ersten Zeitschritt durchgeführt werden. Da die Summation vom Speicherinhalt o_2 vom Speicherzugriff o_1 abhängt, muss diese in den nächsten Zeitschritt verlagert werden, wenn das Lesen des Speichers innerhalb eines Zeitschrittes abgeschlossen ist. Man erhält folgenden Ablaufplan:

- $\tau(o_1) = 0$
- $\tau(o_2) = 1$
- $\tau(o_3) = 0$

Der Schleifenkörper benötigt zur Ausführung zwei Zeitschritte, wobei im ersten ein Addierer und ein Speicherport und im zweiten nur ein Addierer benötigt wird.

Bindung Die Bindung ist eine Abbildung, die jeder Operation o eine konkrete Instanz i einer Funktionseinheit zuordnet, die vorher durch die Allokation bestimmt worden ist. Nach [97] besteht die Bindung aus zwei Funktionen β und γ, wobei

- $\beta(o) = r \in R$ und
- $\gamma(o) = i \leq \alpha(\beta(o))$.

Die Instanz muss kleiner gleich der Anzahl allokierter Instanzen für einen Typ einer Funktionseinheit sein. Im einfachsten Fall besteht R nur aus dedizierten Funktionseinheiten, die jeweils nur eine einzige Operation ausführen können. Wenn eine Operation auf verschiedene Funktionseinheiten abgebildet werden kann, so muss nach anderen Kriterien die optimale Einheit ausgewählt werden. Wichtig ist, dass in einem Algorithmus jede Operation auf mindestens eine Funktionseinheit abgebildet werden kann, da der Algorithmus sonst nicht umsetzbar ist.

Wenn der Speicherzugriff im Schleifenkörper von Abb. 2.1.2 mit o_1, die Summation mit o_2 und die Inkrementierung mit o_3 bezeichnet wird, so kann nach dem im vorigen Abschnitt vorgestellten Ablaufplan folgende Bindung vorgenommen werden:

- $\beta(o_1) = r_S$, $\gamma(o_1) = 1$
- $\beta(o_2) = r_A$, $\gamma(o_2) = 1$
- $\beta(o_3) = r_A$, $\gamma(o_3) = 1$

Da nur eine Addition zur gleichen Zeit durchgeführt wird, können sowohl o_2 als auch o_3 auf derselben Instanz ausgeführt werden, der zweite Addierer wird hier nicht benutzt.

Codegenerierung Nach der Bearbeitung der Syntheseaufgaben und vor der eigentlichen Implementierung der Schaltung sind weitere Entscheidungen hinsichtlich dessen Architektur zu treffen. Bestimmte Vorgaben existieren bereits durch die Auswahl der Zieltechnologie. So eignen sich FPGAs aktuell nur für synchrone Schaltungen [21, 55]. Durch Block-RAMs und Distributed-RAMs stehen den Funktionseinheiten bereits lokale Speicher zur Verfügung. Reichen diese Speicher nicht aus, so müssen bestimmte Datenstrukturen auf externe Speicher außerhalb des FPGAs ausgelagert werden. Das Steuerwerk kann sowohl in Form einer Direktimplementierung als Automat oder als mikroprogrammiertes Steuerwerk synthetisiert werden. Bei der Direktimplementierung wird jeder Zustand im Kontrollfluss der Verhaltensbeschreibung auf einen Zustand des Automaten abgebildet. In den einzelnen Zuständen werden anschließend die jeweiligen Steuersignale für die Funktionseinheiten im Datenpfad aktiviert. Bei einem mikroprogrammierten Steuerwerk werden die Informationen über die Ansteuerung der Funktionseinheiten in einem Speicher abgelegt. Dabei besitzt jeder Zustand einen eigenen Eintrag im Speicher. Je komplexer der Datenpfad ist, desto breiter ist der Speicher auszulegen.

Die Kommunikation zwischen den Funktionseinheiten im Datenpfad mit den Registern und Speichern kann sowohl über Busse als auch über Multiplexer realisiert werden. Beide Varianten haben ihre Vor- und Nachteile, welche die Effizienz der späteren Implementierung erheblich beeinflussen. Bei der multiplexerbasierten Architektur werden die Signale zwischen den Einheiten als Punkt-zu-Punkt-Verbindung realisiert, wobei bei mehreren Quellen

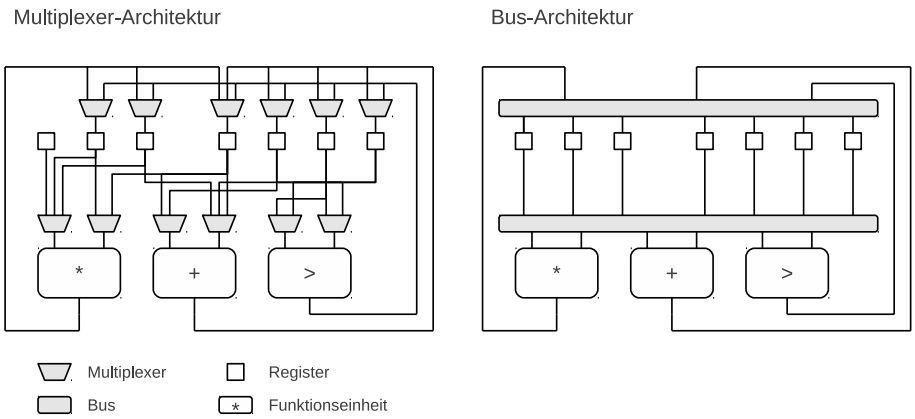


Abbildung 2.1.3: Architekturen für den Datenpfad

und Senken Multiplexer in die Leitungen eingefügt werden. Diese Architektur besitzt eine hohe Flexibilität, gilt aber als sehr aufwändig in Bezug auf die Verdrahtungsressourcen [47]. Bei busorientierten Architekturen teilen sich mehrere Funktionseinheiten dieselben Leitungen, indem sie sich mittels Tristates mit dem Bus verbinden. Dadurch ist der Verdrahtungsaufwand geringer, allerdings stellt der Bus einen Flaschenhals dar und kann die Parallelität bei der Kommunikation einschränken. Häufig liegen Mischformen von Bus- und Multiplexerarchitekturen vor [97]. Heutige FPGAs besitzen neben Multiplexern auch intern einzelne Tristates und Tristate-Busse in der Verdrahtungslogik für entfernte Verbindungswege [42]. Aufgrund des höheren Parallelitätsgrades basiert die in dieser Arbeit generierte RTL-Beschreibung auf Multiplexerarchitekturen. Abb. 2.1.3 zeigt eine Bus- und eine Multiplexer-Architektur.

2.2 Zwischenformate

Im Übersetzerbau werden Programme häufig in Form von Graphen repräsentiert, durch die sowohl der Kontroll- als auch der Datenfluss zwischen den Operationen modelliert werden kann. Beim Parsen des Quellcodes werden Graphen erzeugt und anschließende Programmoptimierungen und -modifikationen durch Transformationen herbeigeführt. Bei der Generierung der Zielsprache werden die dabei entstandenen Knoten- und Kantenstrukturen traversiert und die Konstrukte in entsprechender Form ausgegeben. Die folgenden Abschnitte geben eine Übersicht über die hier verwendeten Graphen.

2.2.1 Syntaxbaum

Der Syntaxbaum [97] entsteht beim Parsen und ist die direkte Überführung des linear und in Textformat vorliegenden Quellcodes einer Sprache in eine hierarchische baumartige Datenstruktur. Die Syntax in einer Sprache sei durch eine kontextfreie Grammatik $G(V_N, V_T, P, S)$ beschrieben, wobei V_N die Menge der Nichtterminale, V_T die Menge der Terminalsymbole, P die Menge der Produktionsregeln und $S \in V_N$ das Startsymbol ist.

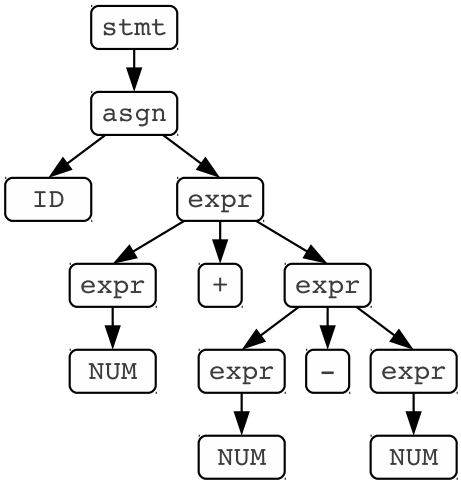


Abbildung 2.2.1: Syntaxbaum zur Anweisung `a=3+b-c`

Für einen Baum B, der ein Wort aus G darstellt, müssen die Ausgangskanten jedes Knotens geordnet sowie die Blätter mit Symbolen aus $V_T \cup \epsilon$ markiert sein und die übrigen Knoten mit Nichtterminalsymbolen V_N. Mit ϵ gekennzeichnete Blätter müssen der einzige Nachfolger des Vorgängers sein. Außerdem muss G für einen Knoten a, dessen Nachfolger mit den Symbolen $b_1, ..., b_n$ markiert sind, die Produktionsregel $a \rightarrow b_1, ..., b_n$ besitzen. Die Beschriftung der Wurzel von B ist nicht notwendigerweise S. Wenn dies jedoch der Fall ist, so ist B ein vollständiger Syntaxbaum. Beim Parsen erzeugt jede erkannte Produktionsregel einen neuen Knoten im Baum, der in den aktuellen Knoten eingefügt wird. Je nach Art der verwendeten Sprachgrammatik werden Syntaxbäume entweder von der Wurzel her (*Top-Down*) oder von den Blättern zur Wurzel (*Bottom-Up*) aufgebaut. Der Top-Down-Aufbau entsteht bei der LL(k)-Analyse[4], Bottom-Up bei der LR(k)-Analyse[4]. Abbildung 2.2.1 zeigt den Syntaxbaum für die Anweisung `a=3+b-c`, wobei Anweisungen als `stmt`-Symbole und Zuweisung mit `asgn` bezeichnet werden. Die Addition bzw. Subtraktion sei als arithmetischer Ausdruck `expr` in der Grammatik vorhanden. Variablen und Zahlen seien Terminalsymbole, die mit `ID` und `NUM` gekennzeichnet sind.

Programmoptimierungen und Umstellungen von Ausdrücken können zwar auf der Ebene des Syntaxbaums durchgeführt werden, auf Kontroll- bzw. Datenflussgraphen sind diese Transformationen allerdings einfacher.

2.2.2 Kontrollflussgraph

Ein Kontrollflussgraph (CFG) ist ein gerichteter kantengeordneter Graph $G(N, E)$ mit den Knoten N und den Kanten E, der zur Modellierung von Kontrollstrukturen eines Programms wie Verzweigungen und Schleifen verwendet wird. Für jede Anweisung S_i innerhalb eines Programms gibt es einen eindeutigen Knoten $n_i \in N$, dem diese zugeordnet ist. Die Kanten E stehen für die Übergänge im Kontrollfluss. Durch die Schleifen können auch Zyklen entstehen. Knoten mit mehr als einem Nachfolger heißen Verzweigungen, Knoten mit mehr als einem Vorgänger Vereinigungen. Der von Verzweigungen ausgehende

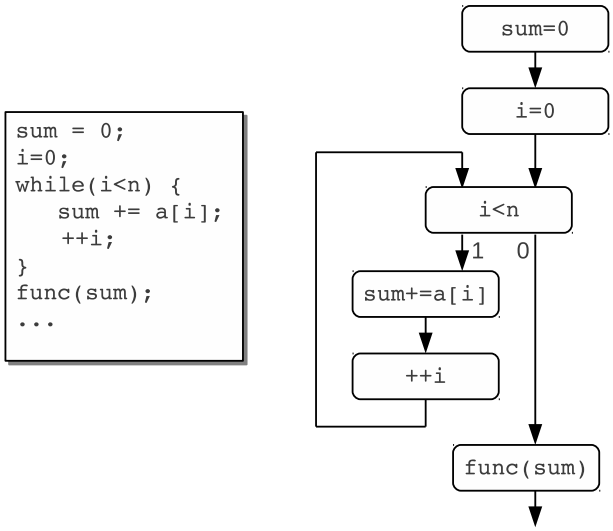


Abbildung 2.2.2: C-Programm mit CFG

Kontrollfluss ist alternativ, sodass immer nur ein Nachfolgezweig durchlaufen wird. Die Auswahl der Verzweigung hängt von dem Ergebnis der Anweisung innerhalb des Verzweigungsknotens ab. Ein CFG hat einen eindeutigen Eintrittsknoten. Abbildung 2.2.2 zeigt ein C-Programm mit zugehörigem Kontrollflussgraphen.

2.2.3 Datenflussgraph

Ein Kontrollflussgraph besitzt keine Mechanismen zur Darstellung von Datenabhängigkeiten zwischen den Anweisungen, diese werden in Datenflussgraphen modelliert. Ein Datenflussgraph (DFG) ist ein gerichteter azyklischer Graph $G(N, E)$, mit den Knoten N und den Kanten E. Die Knotenmenge stellt die Operationen dar, die Kanten stehen für die Datenabhängigkeiten zwischen den Operationen und legen somit dessen Ausführungsreihenfolge fest. Die Berechnung eines Knotens wird lediglich über die Verfügbarkeit von den Daten der Vorgänger gesteuert. Die Menge der Vorgänger eines Knotens n wird in dieser Arbeit mit $Prev(n)$, die der Nachfolger mit $Next(n)$ bezeichnet. Datenflussgraphen sind vergleichbar mit Petri-Netzen[14], bei dem die Knoten N als Transitionen und die Kanten E als mit den Transitionen verbundene Stellen modelliert werden. Stellen ohne Vorbereich stehen für Eingangskanten von Knoten ohne Vorgänger, Stellen ohne Nachbereich stellen Ausgangskanten von Knoten ohne Nachfolger dar. Abbildung 2.2.3 zeigt Operationen eines C-Programms mit dem entsprechenden Datenflussgraphen.

2.2.4 Kontroll-Datenflussgraph

Da ein Datenflussgraph keine Verzweigungen und Iterationen modellieren und ein Kontrollflussgraph keine Datenabhängigkeiten in den Berechnungen darstellen kann, wurde mit dem Kontroll-Datenflussgraphen (CDFG) ein heterogenes Modell eingeführt, das sowohl daten- als auch kontrollflussorientierte Konstrukte vereinigt. Ein CDFG besteht aus einem

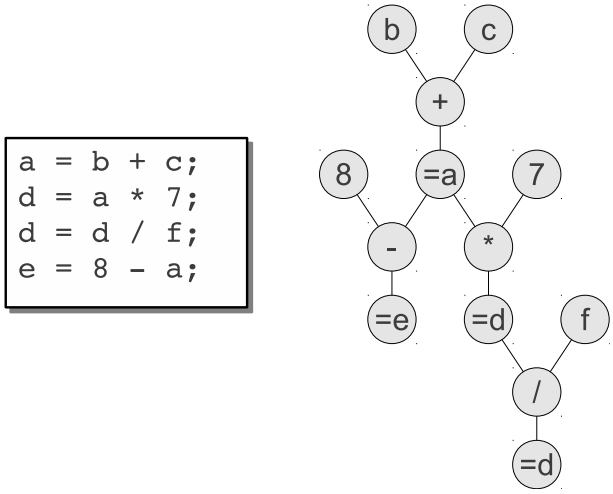


Abbildung 2.2.3: C-Programm mit DFG

Kontrollflussgraphen $G_C(N_C, E_C)$ und einem Datenflussgraphen $G_D(N_D, E_D)$ mit jeweils eigenen Knoten und Kanten. Jedem Knoten des Kontrollflussgraphen sind dabei Datenflussknoten zugeordnet. Wird ein Kontrollknoten $n_{Ci} \in N_C$ durchwandert, so werden alle dazugehörigen Datenflussknoten N_{Di} nach den Ausführungsregeln des DFGs berechnet. Abbildung 2.2.4 stellt einen Kontroll-Datenflussgraphen dar. Die großen weißen Kästen zeigen den Kontrollfluss zusammen mit dem Datenfluss, der in den jeweiligen Kontrollzuständen ausgeführt wird und durch graue Knoten dargestellt ist. Die Datenabhängigkeiten, die zwischen den DFGs der einzelnen Kontrollflussknoten verlaufen, sind der Übersicht halber weggelassen worden. Je nach durchwanderten Kontrollknoten können unterschiedliche Datenknoten referenziert werden. In der letzten Codezeile von Abb. 2.2.4 z.B. kann sich das `i` und das `s` entweder auf die vor oder die in der Schleife geschriebenen Variablen beziehen. Dabei wird immer die Variable referenziert, deren Block am spätesten ausgeführt wurde.

Die DFGs der Kontrollknoten {1, 2} sowie {4, 5} können kombiniert werden, da hier weder Verzweigungen noch Vereinigungen im Kontrollfluss vorkommen. In diesem Fall besitzen alle Kontrollknoten den maximal möglichen Datenfluss. Solche Knoten werden in [6] oder [97] als Grundblöcke bezeichnet. In dieser Arbeit stellen Grundblöcke wie in [37] nicht notwendigerweise solche Segmente von maximalem Datenfluss dar, sondern lediglich einen Kontrollknoten mit einem beliebigen DFG, der auch leer sein kann, um beispielsweise Nulloperationen darzustellen. So hat man auch die Möglichkeit der sequentiellen Abfolge mehrerer Grundblöcke, was in [97] oder [6] ausgeschlossen wird. Dadurch können in bestimmten Transformationen Datenflussgraphen auf mehrere Grundblöcke aufgeteilt bzw. die Graphen von mehreren Grundblöcken in einem Block vereinigt werden.

2.3 Ablaufplanung

Der folgende Abschnitt beschreibt die Ablaufplanung (*Scheduling*) und grundlegende Algorithmen dazu nochmals im Detail. Im Rahmen dieser Arbeit wurde die Implementierung

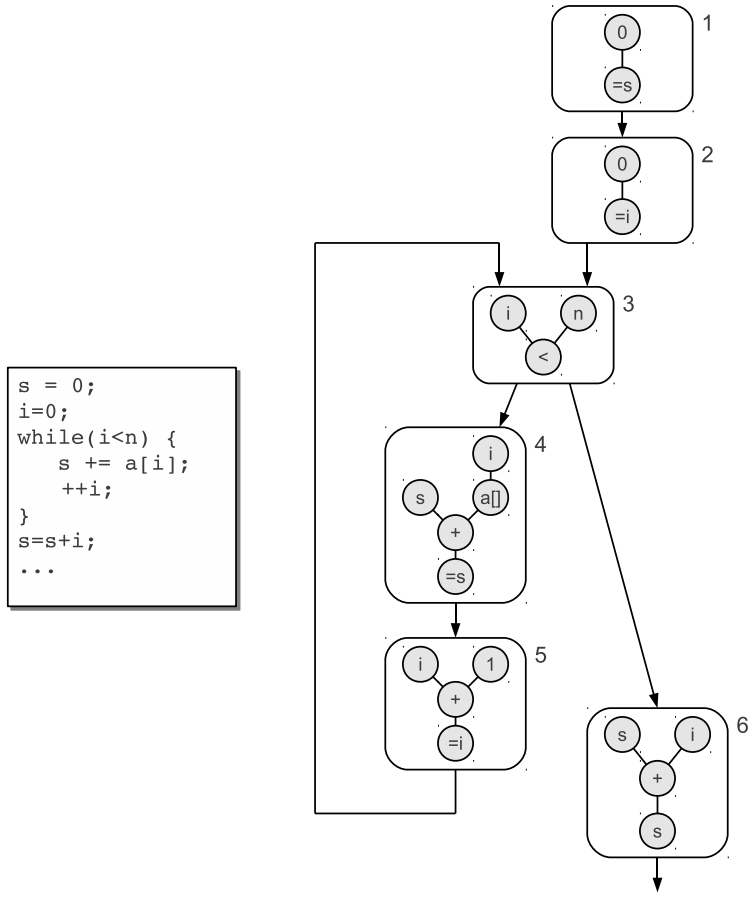


Abbildung 2.2.4: C Programm mit CDFG

einer effizienten Ablaufplanung höher priorisiert als z.B. die einer Bindung. Der generierte VHDL-Code verwendet nach Möglichkeit Operatoren und keine Funktionseinheiten für grundlegende arithmetische und logische Operationen. Dadurch verschiebt sich die Bindung und Allokation von einem großen Teil der Ressourcen in die Phase der RTL-Synthese, die jedoch von einem externen Programm durchgeführt wird, das den hier erzeugten VHDL-Code weiterverarbeitet. Lediglich die Allokation und Bindung von komplexen Einheiten findet hier während der Architektursynthese mit Hilfe von einfachen Heuristiken statt [97]. Die Ablaufplanung hingegen wird auf allen Operationen ausgeführt, wodurch gerade hier das Gesamtergebnis und die Effizienz der resultierenden Schaltung durch die Verwendung besserer Algorithmen stärker beeinflusst werden kann.

Die Ablaufplanung unterteilt sich in die Kategorien statisch und dynamisch bzw. in präemptiv und nicht-präemptiv. Bei statischen Ablaufproblemen sind die Ausführungszeiten sowie der Typ und die Anzahl vollständig zur Kompilierzeit bekannt. Bei präemptiven Systemen können Operationen während der Ausführung unterbrochen und zu einem späteren Zeitpunkt wieder fortgesetzt werden, wie es z.B. bei Prozessen von Betriebssystemen der Fall ist. Da die Operationen einer Verhaltensbeschreibung bereits zur Kompilierzeit bekannt sind und grundlegende Operationen in Hardware in der Regel nicht unterbrochen werden, ist hier lediglich das statische, nicht-präemptive Scheduling von Interesse.

Wenn der Ablaufplan bestimmte Zeitschranken einhalten muss, spricht man von einem zeitbeschränkten Scheduling, bei Vorgaben bezüglich des Ressourcenverbrauchs von einem ressourcenbeschränkten Scheduling.

2.3.1 ASAP

Der ASAP-Algorithmus (engl. *as soon as possible*[100]) liefert einen latenzoptimalen Ablaufplan ohne Ressourcenbeschränkungen. Der Algorithmus wird auf einem DFG G(N,E) angewendet, wobei jede Operation ausgeführt wird, sobald ihre Datenabhängigkeiten erfüllt sind. Algorithmus 1 zeigt die Vorgehensweise im Detail. Neben dem DFG wird für jeden Knoten n von N die Ausführungszeit $d(n)$ benötigt. Ausgegeben wird die Startzeit $\tau(n)$.

Algorithm 1: ASAP

Data: $G(N, E), d$
Result: $\tau(n)\ \forall\ n \in N$
begin
  foreach $n \in N$ *ohne Vorgänger* **do**
    $\tau(n) \leftarrow 0$
  repeat
    Nimm Knoten n, dessen Vorgänger geplant sind
    $\tau(n) \leftarrow max\{\tau(n_j) + d(n_j) \mid n_j \in \text{Prev}(n)\}$
  until *alle* $n \in N$ *geplant* ;
end

2.3.2 ALAP

Der ALAP-Algorithmus (engl. *as late as possible*) ermittelt für jeden Knoten eines DFGs $G(N, E)$ die spätestmöglichen Startzeitpunkte im Gegensatz zum ASAP, der die frühestmöglichen berechnet. ALAP hat als zusätzliche Eingabe eine obere Latenzschranke L, zu der alle Operationen ohne Nachfolger fertig sein müssen. Der Ablauf ist in Algorithmus 2 gezeigt.

Algorithm 2: ALAP

Data: $G(N, E), d, L$
Result: $\tau(n)\ \forall\ n \in N$
begin
  foreach $n \in N$ *ohne Nachfolger* **do**
    $\tau(n) \leftarrow L - d(n)$
  repeat
    Nimm Knoten n, dessen Nachfolger geplant sind
    $\tau(n) \leftarrow min\{\tau(n_j) \mid n_j \in \text{Next}(n)\} - d(n)$
  until *alle* $n \in N$ *geplant* ;
end

Erhält man für einen Knoten negative Ergebnisse, so ist die gegebene Latenzschranke L nicht einzuhalten, als Wert wird üblicherweise die Latenz des ASAP-Schedules gewählt. Ist $\tau(n)^S$ der Startzeitpunkt eines Knoten nach dem ASAP und $\tau(n)^L$ nach dem ALAP, so steht $\tau(n)^L - \tau(n)^S$ für die Mobilität eines Knotens. Die Mobilität gibt die Größe des Intervalls an, in dem Operationen zur Einhaltung der Latenzschranke gestartet werden können. Operationen mit einem Wert von 0 sind kritisch, da ihre Ausführung nur zu einem einzigen Zeitpunkt möglich ist. In anderen Scheduling-Algorithmen wird diese Mobilität der Operationen ausgenutzt.

ASAP und ALAP können erweitert werden, um Randbedingungen hinsichtlich Ressourcenbeschränkungen zu berücksichtigen [99, 59]. Dazu wird zunächst ein Ablaufplan nach dem oben vorgestellten ASAP bzw. ALAP-Algorithmus angefertigt. Werden die Randbedingungen nicht eingehalten, so werden Operationen entsprechend in einen früheren (ALAP) bzw. späteren Zeitpunkt (ASAP) verschoben. Die hiermit erzielten Ergebnisse sind allerdings nicht optimal, bessere Ergebnisse mit Ressourcenbeschränkungen liefern z.B. List-Scheduling-Algorithmen.

2.3.3 List-Scheduling

Das List-Scheduling [84] ist ein heuristisches Verfahren, bei dem jedem Kontrollschritt unter Berücksichtigung von Ressourcenbeschränkungen geeignete Operationen zugewiesen werden. Im Vergleich zum ASAP oder ALAP-Algorithmus bekommt jede Operation am Anfang eine Priorität zugeteilt. Anschließend wird für jeden Kontrollschritt t die Menge an Operationen bestimmt, die noch keinem Zeitschritt zugeordnet wurden, zudem auf einem bestimmten Ressourcentyp r ausgeführt werden können und dessen Datenabhängigkeiten erfüllt sind, d.h. dessen Vorgängeroperationen bereits abgeschlossen sind. Diese Menge wird auch als Kandidatenmenge $K_{t,r}$ bezeichnet. Für den Ressourcentyp r wird außerdem die Menge $G_{t,r}$ an Operationen ermittelt, die zum Zeitpunkt t noch nicht abgeschlossen sind und somit die Ressourcen belegen. Anschließend wird aus der Kandidatenmenge $K_{t,r}$ eine Untermenge $S_{t,r}$ der Operationen maximaler Priorität ausgewählt, so dass $|S_{t,r}| + |G_{t,r}| < \alpha(r)$, wobei $\alpha(r)$ die Zahl der Ressourcen vom Typ r darstellt. Jedem Element der Menge $S_{t,r}$ wird der Kontrollschritt t zugeordnet. Anschließend wird t inkrementiert und alle Schritte ab der Ermittlung der Kandidatenmenge wiederholt. Durch die Inkrementierung werden belegte Ressourcen frei, die dann wieder neuen Kandidaten zugeteilt werden können.

Besitzen alle Operationen einen Kontrollschritt, so terminiert der Algorithmus. Algorithmus 3 zeigt detailliert alle Schritte des List-Schedulings. Benötigt werden hier der Datenflussgraph mit den Kanten und Operationsknoten $G(N, E)$ sowie die Allokation α und die Prioritäten p. Als Ergebnis erhält man für jeden Knoten in N den Kontrollschritt τ.

Ein häufig verwendetes Kriterium für die Priorität ist die Mobilität der Operationen, also die Differenz zwischen ALAP- und ASAP-Kontrollschritt [84, 49]. Die Operationen mit

Algorithm 3: List-Scheduling

```
Data: G(N, E), α
Result: τ(n) ∀n ∈ N
begin
    t ← 0  repeat
        foreach Ressourcentyp r do
            Bestimme Kandidatenmenge K_{t,r}
            Bestimme Menge der nicht abgeschlossenen Operationen G_{t,r}
            Bestimme Menge der Operationen mit max. Priorität S_{t,r}, sodass
            |S_{t,r}| + |G_{t,r}| < α(r)
            foreach v ∈ S_{t,r} do
                τ(v) ← t
        t ← t + 1
    until alle n ∈ N geplant ;
    return τ
end
```

der geringsten Mobilität bekommen die höchste Priorität, da diese weniger alternative Zeitpunkte für die Zuweisung des Kontrollschrittes besitzen. Operationen auf dem kritischen Pfad haben eine Mobilität von 0 und besitzen somit die höchste Priorität. Eine Erweiterung stellt die Berücksichtigung der Anzahl der Nachfolger einer Operation dar. Operationen mit vielen Nachfolgern bekommen eine höhere Priorität, da sich so die Wahrscheinlichkeit für eine größere Kandidatenmenge erhöht. Die verschiedenen Prioritätskriterien werden in [2] verglichen.

2.3.4 Weitere Scheduling-Algorithmen

Neben den hier vorgestellten Techniken gibt es eine Reihe anderer Verfahren, die teilweise bessere Ergebnisse liefern, aber auch wesentlich komplexer in ihrer Implementierung sind. Ein Beispiel hierfür ist das Force-Directed-Scheduling[85], welches eine Erweiterung des List-Schedulings darstellt. Die Priorität einer Operation wird hier nicht statisch am Anfang des Algorithmus bestimmt, vielmehr kann sie sich dynamisch in jedem neuen Kontrollschritt ändern. Exakte Verfahren zur Ablaufplanung liefert die ILP-Technik (*integer linear programming*). Hier wird eine gegebene Zielfunktion minimiert unter Einhaltung von bestimmten Randbedingungen. In [102, 50] werden Methoden gezeigt, die Lösungen unter Zeit- und Ressourcenbeschränkungen berechnen.

2.3.5 Pipelining

Durch Fließbandverarbeitung (engl. *Pipelining*) soll in Schaltungen der Durchsatz erhöht werden, indem durch das Einfügen von Registerstufen der kritische Pfad verkürzt wird und die Logik zwischen den Registerstufen gleichzeitig genutzt werden kann. Es gibt zwei Arten von Pipelining, strukturelles und funktionales. Funktionales Pipelining bezieht sich auf die Generierung von pipelinefähigen Schaltungen zur Maximierung des Durchsatzes

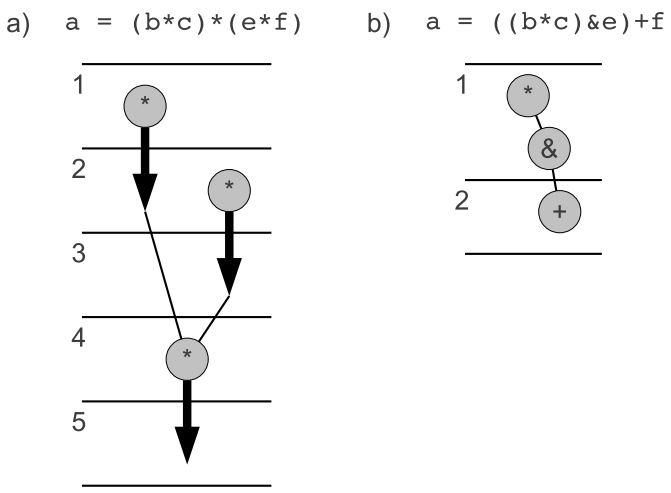


Abbildung 2.3.1: a) Pipelining der Funktion '`a=(b*c)*(d*e)`', wobei die Multiplikation zwei Zyklen benötigt. Die Datenabhängigkeiten verhindern, dass die letzte Multiplikation im dritten Schritt ausgeführt werden kann. b) Verkettung der Funktion '`y = ((a*b) & c ) + d`', die Multiplikation und die UND-Operation werden im selben Takt ausgeführt.

unter minimalem Ressourcenverbrauch, worauf in Abschnitt 6.3 weiter eingegangen wird. Für die Ablaufplanung ist das strukturelle Pipelining von Bedeutung, da sich dieses auf die Ausnutzung der Pipelinefähigkeit von entsprechenden Funktionseinheiten bezieht. In [9] etwa wurde das Pipelining vollständig in die Verhaltenssynthese integriert. Die Zeit, nach der neue Operanden an eine Funktionseinheit angelegt werden können, wird als Iterationsintervall oder Datenverarbeitungsintervall bezeichnet. Wenn das Iterationsintervall kleiner als die Latenz ist, so eignen sich Funktionseinheiten für die Fließbandverarbeitung, d.h. es können bereits neue Operanden angelegt werden, bevor das Ergebnis berechnet ist. Pipelining kann nur zwischen zwei Knoten im DFG ausgenutzt werden, die keine Datenabhängigkeit untereinander aufweisen, da in diesem Fall vor der Ausführung des zweiten Knotens auf das Ergebnis des ersten gewartet werden muss.

2.3.6 Chaining

Wenn die gesamte Ausführungszeit von aufeinanderfolgenden Operationen kleiner als die Taktperiode ist, so können diese innerhalb eines Zyklus ausgeführt werden, indem dessen Funktionseinheiten direkt ohne Speicherelemente hintereinandergeschaltet werden. Dieses Hintereinanderschalten wird als Verketten (engl. *chaining*) bezeichnet und kann in der Ablaufplanung berücksichtigt werden. In diesem Fall wird zwei verketteten Operationen derselbe Kontrollschritt zugewiesen. Wenn z.B. die Funktion '`y = ((a*b) & c ) + d`' in einer Schaltung mit einer Taktperiode von 10 ns implementiert werden soll und die Multiplikation 7 ns, die UND-Operation 1 ns und die Addition 5 ns benötigt, so können die Multiplikation und die UND-Operation im ersten Takt, die Addition im zweiten ausgeführt werden.

[69] untersucht Ansätze, bei dem List-Scheduling auf strukturelles Pipelining erweitert

wird und gleichzeitig aufeinanderfolgende Operationen miteinander verkettet werden. In [104] werden Algorithmen zur Verkettung und Ablaufplanung vorgestellt, die die Operationen möglichst gleichmäßig über die Kontrollschritte verteilen und somit den kritischen Pfad zwischen den Registerstufen reduzieren.

2.4 Optimierungen

In der Softwaresynthese haben Optimierungen den Zweck, entweder die Laufzeit von Programmen zu verringern oder den Bedarf an Speicherplatz zu minimieren. In der Hardwaresynthese gibt es neben der Laufzeit andere Kriterien wie die Schaltungsgröße oder die Reduzierung des Energieverbrauchs, die teilweise widersprüchlich sind. So hat eine Implementierung, die weniger Rechenzeit benötigt, oftmals breitere Busse und mehrere parallele Ausführungseinheiten, die mehr Platz in Anspruch nehmen als die Einheiten einer langsameren Variante. Die verschiedenen Implementierungsvarianten bilden Punkte innerhalb des Entwurfsraumes, dessen Achsen durch die Optimierungskriterien aufgespannt werden. Wenn es für einen Punkt im Entwurfsraum keinen weiteren Punkt gibt, der die Optimierungskriterien mindestens gleich gut erfüllt, so wird dieser als *nichtdominiert* bezeichnet und bildet eine optimale Implementierung hinsichtlich eines Optimierungsziels. Für dominierte Punkte gibt es dementsprechend mindestens einen weiteren Punkt, der alle Kriterien mindestens gleich gut erfüllt. Die nicht-dominierten Punkte werden auch als Pareto-Menge [29] bezeichnet. Bei einem Punkt aus der Pareto-Menge kann kein Parameter verbessert werden, ohne die anderen zu verschlechtern (*Pareto-Superiorität*). Abbildung 2.4.1 zeigt einen Entwurfsraum mit zwei Parametern. Die Suche nach der Pareto-Menge bzw. einer möglichst guten Annäherung wird als Entwurfsraumexploration bezeichnet. Nach der Exploration kann durch eine Abwägung der unterschiedlichen Kriterien durch den Benutzer eine optimale Lösung ausgewählt und implementiert werden.

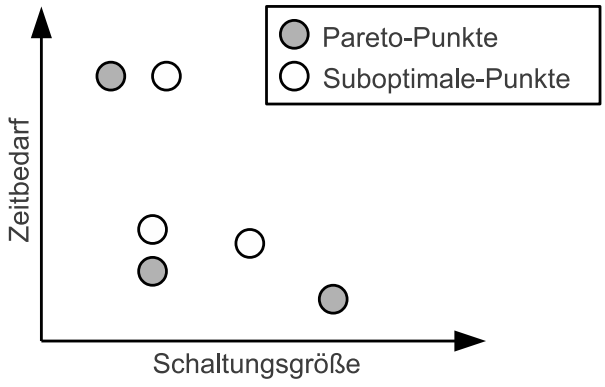


Abbildung 2.4.1: Zweidimensionaler Entwurfsraum mit Pareto-Punkten

Einige Optimierungen wie die Elimination von nicht benötigten arithmetischen Ausdrücken verbessern die resultierende Schaltung hinsichtlich aller Parameter, bei anderen Optimierungen wie der spekulativen Berechnung etwa wird der Zeitbedarf auf Kosten

der Schaltungsgröße gesenkt. Die nachfolgenden Unterabschnitte erläutern die wichtigsten Optimierungsarten, die auch im TransC-Synthesetool zum Einsatz kommen. Die meisten Transformationen können sowohl für die Software- als auch für die Hardwaresynthese verwendet werden. Hier wird zwischen Kontroll- und Datenflusstransformationen unterschieden. Optimierungen, die sich nur für die Hardwaresynthese eignen, werden in einem separaten Abschnitt behandelt.Viele der in Software angewendeten Optimierungen sind bei der Erzeugung von digitalen Schaltungen nicht so relevant. Zusätzliche Berechnungen etwa können in Hardware oftmals parallel zu anderen Operationen verarbeitet werden. Wenn die dafür benötigten Ausführungseinheiten bereits vorhanden sind, verursachen diese Operationen bis auf die zusätzliche Steuerungslogik keinen Aufwand. Auf Mikroprozessoren hingegen beansprucht jede Instruktion in der Regel zusätzliche Takte. Aus diesem Grund wird bei jeder Transformation die Relevanz für die Hardwaresynthese kurz erläutert.

Weitere Optimierungen, die im Rahmen dieser Arbeit allerdings nicht angewendet wurden, sind u.a in [97] oder [37] erläutert.

2.4.1 Kontrollfluss-Transformationen

Elimination leerer Grundblöcke Durch den Aufbau eines CDFGs beim Parsen können leere Grundblöcke entstehen oder durch bestimmte Optimierungsschritte wie dem Entfernen von Operationen alle Operationen innerhalb eines Grundblocks wegfallen. Diese Grundblöcke können aus dem Kontrollfluss entfernt werden. Wenn bei Verzweigungen alle einzelnen Zweige leer sind, so zeigen die Verzweigungen nach der Elimination der leeren Blöcke auf denselben Nachfolgeblock, so dass die Verzweigungsbedingung ebenfalls überflüssig wird (Abb. 2.4.2). Durch den Wegfall der Blöcke vereinfacht sich die Steuerung des Programms, da je nach Art der Hardwareimplementierung leere Blöcke auch Taktzyklen in Anspruch nehmen können. Außerdem werden weitere Optimierungen wie das Verschmelzen von Grundblöcken ermöglicht.

Elimination von Verzweigungsbedingungen Durch vorhergehende Optimierungsschritte können Verzweigungsbedingungen auftreten, die den konstanten Wert *wahr* oder *falsch* besitzen. Auch wenn in Verzweigungsbedingungen eine Variable mit einer Konstanten verglichen wird und durch vorhergehende Analysen der Wertebereich dieser Variable eingeschränkt werden konnte, kann es vorkommen, dass die Bedingung ebenfalls immer das gleiche Ergebnis liefert. Als Folge ist es möglich, den Zweig im Kontrollfluss, der bei der Programmausführung traversiert wird, bereits zur Kompilierzeit zu bestimmen.

In solchen Fällen kann die Verzweigungsbedingung vollständig entfernt werden, wodurch auch der Grundblock nur noch einen Nachfolger besitzt. Dieser Nachfolger kann wiederum mit dem Grundblock zusammengefasst werden, wodurch sich weitere Optimierungsmöglichkeiten auch auf dem Datenfluss ergeben. Abb. 2.4.3a zeigt Quellcode und den CDFG mit einer Verzweigungsbedingung. Wenn bekannt ist, dass `a` beispielsweise immer einen Wert zwischen 3 und 8 annimmt, kann der Vergleich eliminiert werden, was den

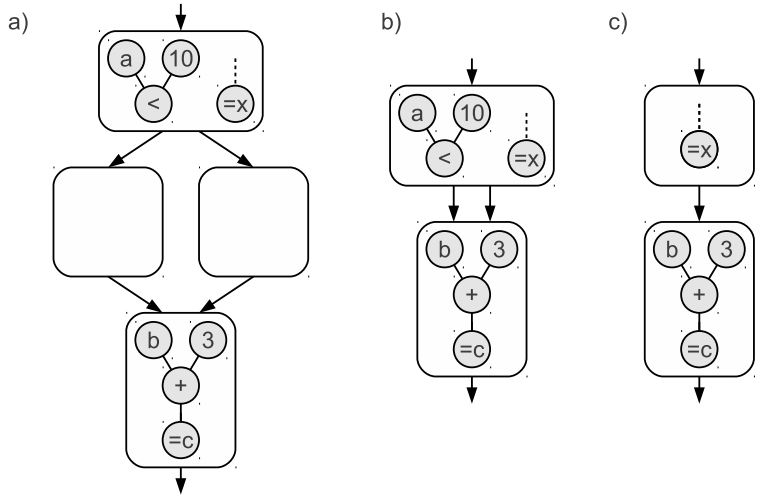


Abbildung 2.4.2: Verzweigung mit leeren Blöcken (a). Nach dem Entfernen der Blöcke zeigen beide Kanten auf denselben Nachfolger (b), wodurch eine Kante und die Verzweigungsbedingung entfernt (c) und ggf. die Blöcke verschmolzen werden können.

Datenfluss vereinfacht (Abb. 2.4.3b) und Optimierungen wie z.B. die Eliminierung unerreichter Grundblöcke und die Verschmelzung ermöglicht (Abb. 2.4.3c).

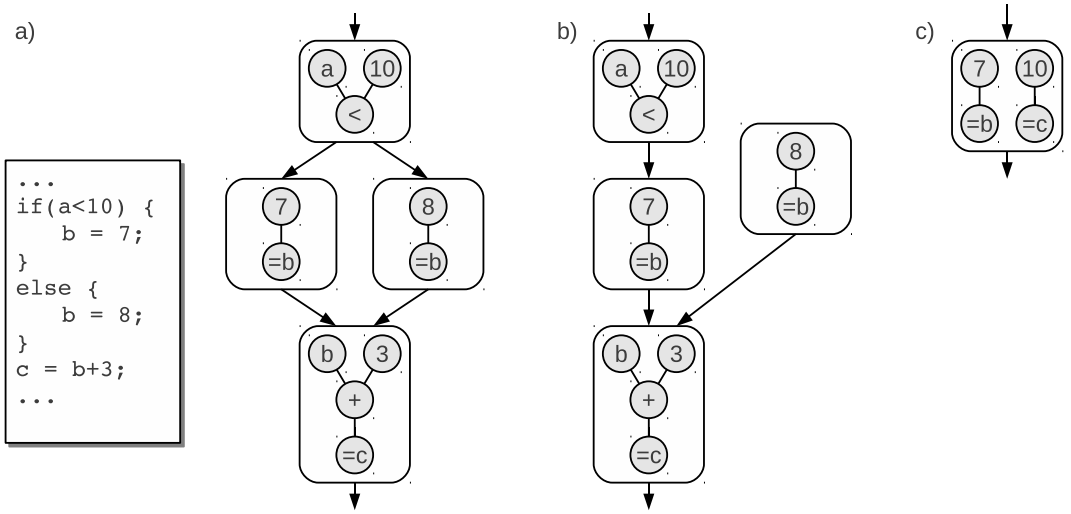


Abbildung 2.4.3: (a) Kontroll-Datenflussgraph mit bekannter Verzweigungsbedingung, (b) CDFG nach der Elimination, (c) CDFG nach Anwendung weiterer Optimierungsschritte

Elimination unerreichter Grundblöcke Durchwandert man einen Kontrollflussgraphen rekursiv ab dem Startblock entlang aller Kanten und entfernt die durchwanderten Blöcke aus der Menge aller Blöcke, so erhält man die Menge unerreichter Grundblöcke. Diese können aus dem Graphen entfernt werden, da sie zu keinem Zeitpunkt bei der späteren Programmausführung vom Startblock aus erreicht werden können. Solche Konstrukte entstehen etwa durch Code hinter Endlosschleifen oder Return-Anweisungen. Auch nach der Elimination von Verzweigungsbedingungen können unerreichbare Grundblöcke auftre-

ten, wie in Abb. 2.4.3b gezeigt. Durch die Elimination verkürzt sich zwar nicht die Laufzeit des Programms, allerdings verringert sich die Komplexität des Steuerwerks, da sonst auch die Zustände der leeren Blöcke berücksichtigt werden.

Abrollen von Schleifen Beim Abrollen von Schleifen wird der Code innerhalb der Schleife vervielfältigt und die Zahl der Schleifeniterationen dementsprechend verringert. Dadurch kann die Gesamtlaufzeit vermindert werden, da ein größerer Schleifenkörper mehr Optimierungen auf dem Datenpfad ermöglicht, wenn die Schleife nur aus einem einzigen Grundblock besteht. Außerdem muss die Sprungbedingung der Schleife nicht mehr so häufig überprüft werden, so dass auch hier Operationen eingespart werden können. Da in Hardware die Sprungbedingung oftmals parallel zum Schleifenkörper berechnet werden kann, lohnt sich diese Optimierung eher wegen der zusätzlichen Optimierungen, die auf einem längeren Schleifenkörper ggf. möglich sind.

Eine Schleife kann partiell oder vollständig abgerollt werden, wodurch die Schleife und deren Kontrollkonstrukte komplett wegfallen. Auf der einen Seite wird die Anzahl an abzuarbeitenden Instruktionen geringer, auf der anderen Seite wird das resultierende Programm insgesamt größer. So kann es z.B. bei Mikroprozessoren vorkommen, dass das Programm nicht mehr in den schnellen Instruktionscache passt oder bei der Implementierung in Hardware die Komplexität des Datenpfades so zunimmt, dass durch die aufwändigere Verbindungslogik die Taktfrequenz sinkt. Das effiziente Abrollen ist nur für Schleifen möglich, bei denen die Iterationszahl bekannt ist. Prinzipiell können auch Schleifen mit unbekannter Iterationszahl abgerollt werden [77], allerdings sind dafür zusätzliche Kontrollstrukturen notwendig. Das folgende Beispiel zeigt eine Schleife vor und nach der Umformung:

```
1 //vorher:                          //nachher:
2 for(i=0; i<4000; ) {               for(i=0; i<4000; ) {
3     a[i] = b[i] + c[i];                a[i] = b[i] + c[i]; ++i;
4     ++i;                               a[i] = b[i] + c[i]; ++i;
5 }                                      a[i] = b[i] + c[i]; ++i;
6                                        a[i] = b[i] + c[i]; ++i;
7                                    }
```

Partitionierung von Grundblöcken Wie in Abschnitt 2.2.4 erwähnt stellt in dieser Arbeit ein Grundblock nicht unbedingt den maximal möglichen Datenfluss zwischen Verzweigungen und Vereinigungen dar, so dass auch mehrere Grundblöcke in einer sequentiellen Abfolge ausgeführt werden können. Folglich ist es möglich, den Datenfluss innerhalb eines Blockes aufzutrennen und auf zwei Blöcke aufzuteilen, so dass die resultierenden Grundblöcke jeweils eine geringere Komplexität aufweisen, wie in Abb. 2.4.4 gezeigt. Die Partitionierung wird oft in Verbindung mit anderen Optimierungen wie der Loop-Invariant-Code-Motion durchgeführt. Hier wird innerhalb eines Schleifenkörpers der schleifenunabhängige Datenfluss in einen separaten Block ausgelagert, den man dann wiederum vor die Schleife schieben kann.

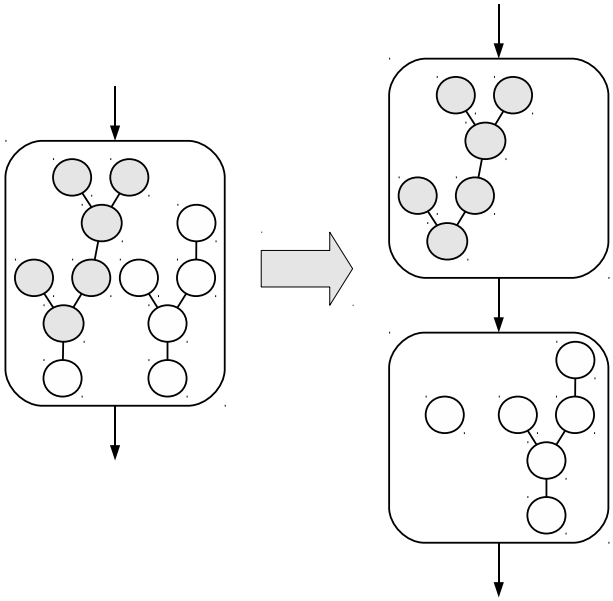


Abbildung 2.4.4: Partitionierung eines Grundblocks

Außerdem kann durch Partitionierung indirekt auf die Ablaufplanung oder die Allokation Einfluss genommen werden, da Grundblöcke dort elementare Einheiten darstellen, auf denen die Algorithmen angewendet werden. Durch die geringere Komplexität dieser Einheiten ist es möglich, diese Verfahren zu beschleunigen.

Verschmelzen von Grundblöcken Die Verschmelzung von Grundblöcken ist die inverse Transformation zur Partitionierung. Dabei werden die einzelnen Datenflussgraphen von zwei aufeinanderfolgenden Grundblöcken zu einem komplexeren Graphen zusammengefasst. Da viele Optimierungsalgorithmen grundblockweise arbeiten, erhöht sich hierdurch das Optimierungspotential. Auf der anderen Seite vergrößert sich durch die komplizierteren Datenflussgraphen die Komplexität von bestimmten Synthesealgorithmen wie der Ablaufplanung oder der Allokation. Die Verschmelzung findet z.B. beim Abrollen von Schleifen statt. Hier wird der Schleifenkörper, der in Form von einem oder mehreren Grundblöcken vorliegt, dupliziert. Die neu entstandenen aufeinanderfolgenden Blöcke können wiederum zu einem einzigen Block zusammengefasst werden.

Verschieben schleifen-invarianter Codesegmente Das Verschieben von schleifeninvarianten Codesegmenten (engl. *Loop Invariant Code Motion*[5]) dient dazu, Ausdrücke vom Schleifenrumpf in einen neuen Block vor den Schleifenkopf (*Pre-header*) zu verlagern, ohne dabei die Semantik des Programms zu verändern. Dazu dürfen sich die Ausdrücke während der einzelnen Schleifeniterationen nicht verändern. Da bestimmte Zwischenergebnisse dann nicht während jedes Durchlaufs, sondern nur einmal vor der Schleife berechnet werden, wird i. A. eine Beschleunigung erreicht. Auf der anderen Seite müssen diese Zwischenergebnisse in Registern gehalten werden, was bei Prozessoren zu Auslagerungen in den Speicher führen kann und das Programm wieder verlangsamt. In Hardware lohnt sich

diese Optimierung nur, wenn die ausgelagerten Operationen nicht parallel zu den übrigen Operationen berechnet werden können. Aus diesem Grund muss abgeschätzt werden, ob sich tatsächlich eine Verkürzung der Laufzeit ergibt.

Voraussetzung für das Erkennen von schleifen-invarianten Codesegmenten ist die in Abschnitt 5.3.6 beschriebene Lebenszeitanalyse und die dadurch erzeugten *Live-In*- und *Live-Out*-Mengen.

$$t : x = y \bullet z$$

sei ein Ausdruck innerhalb einer Schleife mit der Operation $\bullet$, der definierten Variablen x und den gelesenen Variablen y und z. Der Ausdruck ist schleifen-invariant

1. wenn die gelesenen Variablen entweder konstant sind,
2. die Definitionen der gelesenen Variablen, die den Ausdruck erreichen, außerhalb der Schleife durchgeführt wurden oder
3. wenn die Definitionen der gelesenen Variablen, die den Ausdruck erreichen, ebenfalls schleifen-invariante Ausdrücke sind.

Der schleifeninvariante Ausdruck t darf dann in den *Pre-header* ausgelagert werden, wenn

1. der Grundblock von t alle Schleifenausgänge dominiert, bei denen x nach Ausführung lebendig ist,
2. x nur einmal innerhalb der Schleife definiert wird und
3. x nicht hinter dem *Pre-header* lebendig ist.

Außerdem darf der auszulagernde Ausdruck keine Operation mit Seiteneffekten besitzen.

Expansion von Funktionsaufrufen Durch die Expansion von Funktionsaufrufen (engl. *function inlining*) wird der gesamte Code der aufgerufenen Funktion unter Berücksichtigung von z.B. Namenskonflikten bei Variablen in die Aufruferfunktion integriert, so dass der eigentliche Aufruf wegfällt. Dadurch wird auch der Aufwand für den Funktionsaufruf eliminiert, der bei Mikroprozessoren die Sicherung des Programmzählers, der Register, den Programmsprung usw. beinhaltet. Außerdem werden z.B. durch konstante Funktionsargumente weitere Optimierungen ermöglicht, die sonst aufgrund der Hierarchiegrenzen nicht anwendbar gewesen wären. Auf der anderen Seite kann sich das Programm erheblich vergrößern. Bei Mikroprozessoren entstehen dadurch Nachteile, wenn das Programm nicht mehr in den Instruktions-Cache passt und Instruktionen aus dem langsameren Hauptspeicher geholt werden müssen. Bei der Hardwaregenerierung kann ein größeres Programm eine größere Schaltung mit einem komplexeren Zustandsautomaten zur Folge haben, die mehr Ressourcen verbraucht oder mit einer geringeren Taktfrequenz läuft, weshalb sich die Expansion oftmals nur lohnt, wenn sich aufgrund von konstanten Argumenten die meisten Operationen zur Kompilierzeit berechnen lassen.

Spezialisierung von Funktionen Im Vergleich zur Expansion von Funktionsaufrufen, wo der Funktionsaufruf eliminiert und der Inhalt der Funktion vollständig in die aufrufende Funktion integriert wird, wird bei der Spezialisierung (engl. *function cloning*) die aufgerufene Funktion durch eine optimierte Variante ersetzt. Dies ist möglich, wenn Argumente in der Schnittstelle konstant sind oder die Argumente Datenabhängigkeiten zueinander besitzen. In diesem Fall wird eine Kopie der Funktion mit anderem Namen und reduzierter Argumentenliste erzeugt. Innerhalb der Kopie wird ein neuer Grundblock generiert, in dem die aus der Argumentenliste entfernten Parameter initialisiert werden. Durch die Initialisierung ergeben sich Möglichkeiten für weitere Optimierungen, wie die Propagierung von Konstanten, die Auswertung von Verzweigungsbedingung oder die Elimination unerreichbarer Konstrukte. Neben der einfacheren Schnittstelle erhält man somit auch eine effizientere Funktion. Durch die Duplizierung der bestehenden Funktion wird das Programm zwar länger, allerdings kann die spezialisierte Variante wiederverwendet werden, wenn mehrere Aufrufe mit denselben Konstantenwerten vorhanden sind. Zudem wird bei einer Hardwareimplementierung der Zustandsautomat nicht größer als bei der Expansion von Aufrufen, da die Funktionen nicht verschmelzen.

Elimination unerreichbarer Funktionen Durch eine Programmanalyse, in der alle aufgerufenen Funktionen protokolliert werden, kann die Menge der nicht aufgerufenen Funktionen identifiziert werden. Diese entstehen z.B. durch nicht verwendete Bibliotheksfunktionen oder durch die Spezialisierung bzw. Expansion von Funktionsaufrufen. Unerreichte Funktionen können aus dem Programm entfernt werden, wodurch sich dessen Größe und damit der Ressourcenverbrauch sowie die Kompilierzeit verringert.

2.4.2 Datenfluss-Transformationen

Konstantenfaltung Konstantenfaltung ist die Ersetzung einer Operation mit konstanten Operanden durch dessen Ergebnis. Ausdrücke wie

$$c = 34 + 17$$

werden transformiert zu

$$c = 51.$$

Somit wird die Operation eingespart, da sie schon zur Kompilierzeit ausgewertet wurde. Häufig kann die Konstantenfaltung nicht direkt ausgeführt werden, sondern erst nach vorangegangenen Optimierungsschritten wie dem Abrollen von Schleifen und der Konstantenpropagation. Gerade durch die Konstantenfaltung ergeben sich viele weitere Optimierungen, die auch den Kontrollfluss betreffen.

Konstantenpropagation Bei der Konstantenpropagation wird eine Variable durch dessen Wert ersetzt, wenn dieser zur Kompilierzeit bekannt ist. Ausgangspunkt ist dabei die

Zuweisung an eine Variable. Bei Lesezugriffen von dieser Variablen kann diese dann direkt durch den zugewiesenen Wert substituiert werden, wenn keine zwischenzeitliche Manipulation stattgefunden hat.

Anweisungen wie

```
1 c = 12;
2 a = c + 3;
3 c = d;
4 b = c * 7;
```

werden zu

```
1 a = 12 + 3;
2 c = d;
3 b = c * 7;
```

Die Konstantenpropagation ermöglicht eine Vielzahl weiterer Optimierungen, wie z.B. die Auswertung von Verzweigungsbedingungen während der Kompilierung. So können neben der Eliminierung von Operationen auch unerreichte Grundblöcke bestimmt und aus dem Kontrollfluss entfernt werden. Dies geschieht häufig in Kombination mit der Konstantenfaltung.

Reduktion von Operator-Kosten Bestimmte Operationen können durch äquivalente Ausdrücke ersetzt werden, dessen Ausführungseinheiten schneller sind oder weniger Ressourcen verbrauchen. So könnte die Multiplikation oder Division von Ganzzahlen mit einer Zweierpotenz durch Schiebeoperationen ersetzt werden. Allerdings ist diese Optimierung stark von der Zielarchitektur abhängig. Bei einer Hardwareimplementierung würde sich die Transformation lohnen, da konstante Schiebeoperationen theoretisch keine Ausführungseinheiten benötigen würden. In Software macht die Transformation nur Sinn, wenn der Schiebebefehl weniger Takte als die Multiplikation verbraucht.

Algebraische Optimierungen Algebraische Optimierungen stellen algebraische Ausdrücke hinsichtlich mathematischer Regeln wie dem Assoziativ-, Kommutativ- oder Distributivgesetz um. Die Anwendung dieser Transformation hängt allerdings stark von den Optimierungszielen ab. Eine Umstellung des Ausdrucks $a*(b*(c*d))$ nach $(a*b)*(c*d)$ erhöht den möglichen Parallelitätsgrad, da $a*b$ und $c*d$ gleichzeitig berechnet werden können. Wenn jedoch die Zahl der Ausführungseinheiten begrenzt ist, so erhöht die Transformation den Registerverbrauch, da mehr Zwischenergebnisse gehalten werden müssen. Diese Umstellung lohnt sich also nur, wenn in Richtung Geschwindigkeit optimiert werden soll und genügend Ausführungseinheiten zur Verfügung stehen. Bestimmte Transformationen können weitere Optimierungen wie die Konstantenfaltung ermöglichen. $(3+a)+4$ etwa kann nach Anwendung des Kommutativ- und Assoziativgesetzes zu $a+(3+4)$ umgestellt und weiter zu $a+7$ reduziert werden.

Elimination gemeinsamer Teilausdrücke Bei der Eliminierung gemeinsamer Teilausdrücke (engl. *Common Subexpression Elimination*, CSE) werden gleiche Instruktionen bei der Berechnung von arithmetischen Ausdrücken gesucht und anschließend eliminiert. Beispielsweise können die Anweisungen

```
1 a = x * z + 3;
2 b = x * z * 4;
```

zu

```
1 tmp = x * z;
2 a = tmp + 3;
3 b = tmp * 4;
```

transformiert werden, wodurch man die erneute Berechnung von `x*z` spart. Dies lohnt sich allerdings nur, wenn die Kosten für die Speicherung von `tmp` geringer sind als für die Auswertung, d.h. ausreichend Register zu Verfügung stehen. Das Gegenteil der Elimination ist die Expansion gemeinsamer Teilausdrücke, bei der das Speichern von Zwischenergebnissen durch die erneute Berechnung ersetzt wird. Wenn die Eliminierung innerhalb eines Grundblocks angewendet wird so spricht man auch von lokaler, ansonsten von globaler CSE. In Hardware spart man durch die Elimination Funktionseinheiten bzw. kann diese für andere Berechnungen verwenden.

Skalarisierung von Feldern Die Skalarisierung von Feldern hat das Ziel, lokale Arrays vollständig durch Variablen zu ersetzen. Lokale Arrays sind Felder, die innerhalb einer Funktion definiert sind, nur von dieser Funktion benutzt werden und somit nicht zur Übergabe von Daten an weitere Funktionen dienen. Wenn in diesem Fall alle Zugriffe mit bekannten Indizes erfolgen, kann jedes Element innerhalb des Arrays durch eine temporäre Variable ersetzt werden, wodurch sich auch die Zugriffszeit verringert. Voraussetzung dafür sind meistens andere Optimierungen wie das Abrollen von Schleifen sowie Konstantenpropagation und -faltung. Felder können oftmals in Situationen skalarisiert werden, in denen sie innerhalb von Schleifen zur Aufnahme von Zwischenergebnissen eingesetzt werden. Besonders Implementierungen in Hardware profitieren von dieser Optimierung, da Algorithmen, die mit Hilfe von Arrays und Schleifen beschrieben sind, nach der Transformation nur noch aus einem Datenfluss bestehen. Dieser Datenfluss kann direkt in eine gepipelinte Schaltung umgesetzt werden.

Optimierung von Feldzugriffen Bei der Optimierung von Feldzugriffen werden mehrfache Zugriffe auf identische Elemente eliminiert, was man an identischen Indizes erkennen kann. Dies gilt sowohl für das mehrfache Lesen desselben Elementes als auch für das mehrfache Schreiben oder das Schreiben und eine darauf folgende Leseoperation. Dadurch fallen Arrayzugriffe weg, die in der Regel einen höheren Aufwand als Zugriffe auf Variablen erfordern. Arrayzugriffe setzen Indexberechnungen voraus, außerdem können Variablen oftmals

auf Register abgebildet werden, während Arrays als externe Speicher implementiert werden. Arrayzugriffe wie

```
1 buffer[x] = a;
2 ...
3 b = buffer[x];
```

können zu

```
1 buffer[x] = a;
2 tmp = a;
3 ...
4 b = tmp;
```

reduziert werden, was die Leseoperation einspart. Wenn `a` zwischenzeitlich nicht modifiziert worden ist, so kann auch `a` statt `tmp` verwendet werden.

Wenn ein Array von mehreren parallelen Prozessen genutzt wird, sind diese Eliminierungen nicht mehr möglich, wenn sich zwischen den beiden Arrayoperationen die Prozesse synchronisieren, da sonst falsche Werte im Array gespeichert werden, bzw. falsche Werte für die Elemente vorausgesetzt werden.

Besonders effektiv ist die Optimierung von Feldzugriffen innerhalb von Schleifen, bei denen ein Arrayelement mit konstantem Index als temporäre Speicherstelle dient und bei jeder Iteration gelesen und geschrieben wird, wie das folgende Beispiel zeigt:

```
1  //vorher:
2  for(i=0; i<n; ++i) {
3      for(j=i; j<m; ++j) {
4          a[i] = a[i] + ...;
5      }
6  }
7
8  //nachher:
9  for(i=0; i<n; ++i) {
10     tmp = a[i];
11     for(j=i; j<m; ++j) {
12         tmp = tmp + ...;
13     }
14     a[i] = tmp;
15 }
```

Elimination unbenutzter Berechnungen Wenn die Ergebnisse von Ausdrücken nicht benötigt werden, so können diese entfernt werden. Im folgenden Beispiel ist die Multiplikation überflüssig, die sowohl in Hardware als auch in Software zusätzliche Zyklen zur Berechnung beansprucht.

```
1 int func(int a) {
2     int c = 3+a;
3     int d = c*7;
4     int e = subfunc()+5;
```

```
5     return c;
6 }
```

Wenn innerhalb der nicht benötigten Berechnung Funktionsaufrufe vorkommen, die ggf. Seiteneffekte haben oder wo das Halteproblem für die Funktion nicht gelöst werden kann, so muss der Aufruf dennoch erfolgen.

```
1 int func(int a) {
2     int c = 3+a;
3     subfunc();
4     return c;
5 }
```

2.4.3 Hardwarenahe Transformationen

Wortbreitenminimierung Die Minimierung von Wortbreiten ist eine Optimierung, die sich besonders für die Hardwaresynthese eignet. Dabei wird ausgenutzt, dass die Breiten der durch den Programmierer vorgegebenen Datentypen meist größer sind als die tatsächlich zur Berechnung benötigten. Durch die Reduzierung der Datenbreiten können Ressourcen eingespart bzw. die Berechnungszeit von Operationen verkürzt werden, da ggf. auch die Funktionseinheiten kleiner werden. Algorithmen zur Wortbreitenminimierung nutzen bei Operationen den Zusammenhang der Wortbreiten aus, der zwischen dem Ergebnis und den Operanden besteht. Das folgende Beispiel zeigt ein C-Beispiel mit einer Schiebeoperation und einer Addition, dessen Ergebnisse in 32-Bit integer Variablen abgespeichert werden, wo die Operanden aber nur eine Breite von 16-Bit haben. Somit kann das Ergebnis der Schiebeoperation in einem 13-Bit Register gehalten werden, für die Addition wird lediglich ein 17-Bit Register benötigt. Für die UND-Operation benötigt man nur eine 8-Bit Ausführungseinheit und ein 8-Bit Register.

```
1 short a, b, c;
2 int d, e;
3 ...
4 c = a >> 3;
5 d = (int)a + b;
6 e = d & 0xff;
```

Ein Verfahren, das sogar einzelne Bits betrachtet, ist beispielsweise in [16] beschrieben. Dort werden die Datenpfade einer Vorwärts- und einer Rückwärtsanalyse unterzogen, wodurch für jeden einzelnen DFG-Knoten die Breite ermittelt werden kann. Es wird gezeigt, dass bei größeren Reduzierungen von ganzen Bytes selbst die Softwaresynthese von den Optimierungen profitiert, da auf manchen Prozessoren etwa 16-Bit-Typen effizienter verarbeitet werden können als 32-Bit-Typen. In Abschnitt 6.2 wird eine Datenpfadminimierung mit Hilfe der partiellen Evaluierung von Funktionen erreicht.

Spekulative Berechnungen Bei spekulativen Berechnungen werden in einem bestimmten Zustand im Programmfluss zusätzliche Ressourcen verwendet, um Berechnungen durchzuführen, deren Ergebnisse zu einem späteren Zeitpunkt benötigt werden könnten. Spekulative Berechnungen in der Architektursynthese werden u.a. im Spark-Compiler [44] zur Erhöhung der Parallelität generiert, aber auch in Mikroprozessoren durchgeführt, um nicht verwendete Ausführungseinheiten auszulasten [88].

Eine in dieser Arbeit angewendete Möglichkeit zur Durchführung von spekulativen Berechnungen ist die Ersetzung von Verzweigungen im Kontrollfluss durch Multiplexer im Datenfluss. Dabei werden bei einer einfachen Verzweigung, an der höchstens zwei Grundblöcke beteiligt sind, die Datenflussgraphen in einem Block zusammengefügt. Die Ergebnisse der jeweiligen Anweisungen werden in Multiplexer geführt, wobei der Selektoreingang mit dem Ergebnis der *if*-Bedingung gesteuert wird. Variablen, die nur in einem Zweig geschrieben werden, bekommen durch den Multiplexer entweder den neu berechneten oder ihren alten Wert wieder. Auf diese Weise werden beide Zweige immer ausgewertet, die Ergebnisse des nicht genommenen Zweigs aber verworfen. Beide Blöcke sollten etwa die gleiche Laufzeit besitzen, da durch die spekulative Berechnung der neue Block immer die maximale Laufzeit von beiden Ursprungsblöcken aufweist. In Arbeiten wie [45] gibt es daher spezielle *Cancel*-Tokens, an denen die spekulativen Berechnungen abgebrochen werden können, wenn die Verzweigungsbedingung berechnet ist. Die spekulativ ausgeführten Knoten dürfen keine Seiteneffekte verursachen wie z.B. Arrays beschreiben oder Funktionen mit Seiteneffekten aufrufen. Durch das Verschmelzen von Grundblöcken und das Zusammenfassen der DFGs eröffnen sich weitere Optimierungsmöglichkeiten. Abb. 2.4.5 zeigt einen Algorithmus, der den größten gemeinsamen Teiler von zwei Zahlen berechnet sowie den CDFG des Schleifenkörpers vor und nach der Optimierung. In einer Hardwareimplementierung können nach der Optimierung der Vergleich als auch die Addition und die Subtraktion parallel ausgeführt und mit dem Multiplexer verkettet werden, wodurch die Laufzeit erheblich gesenkt wird. Allerdings steigt auch der Ressourcenverbrauch, da aufgrund der höheren Parallelität mehr Ausführungseinheiten benötigt werden.

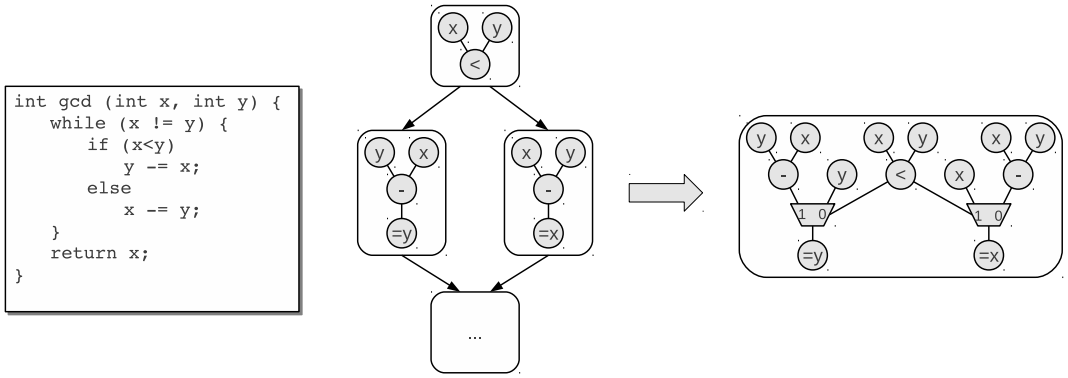


Abbildung 2.4.5: Zwischencodetransformation durch spekulative Berechnungen

Diese Transformation wird auch in Arbeiten wie [18] verwendet, um die Parallelität zu erhöhen und den Kontrollfluss zu vereinfachen.

Transformation von Schleifenbedingungen Die Transformation der Schleifenbedingung ist eine Optimierung, die sich nur bei der Schaltungssynthese lohnt. Diese Optimierung setzt eine Schleifenanalyse voraus, bei der die Schleifengrenzen sowie der Inkrementierungswert der Induktionsvariablen bestimmt worden sind. Wenn die Schleife die Form '`for(i=K; i<L; i=i+M)`' hat, wobei $K < L$, $L = 2^n$ mit $n \in N$ und L ein ganzzahliges Vielfaches von M ist, so muss nur ein einziges Bit der Induktionsvariablen überprüft werden. Hierdurch spart man große Komparatoren und reduziert den Ressourcenverbrauch. Wenn sowieso Komparatoren vorhanden sind, kann die Schaltungsgröße nicht reduziert werden. Allerdings ist der Vergleich eines einzigen Bits schneller, wodurch man bei der Ablaufplanung mehr Möglichkeiten beim Chaining erhält.

2.5 Prozessmodelle

Ein Prozess ist ein Programm oder eine Funktion in der Ausführung. Zur Beschreibung von Prozessen werden u.a. Zustandsmaschinen benutzt. Nach [72] ist ein Prozess die Instantiierung eines Automaten, der durch eine Transitionsfunktion und einen Zustandsspeicher charakterisiert ist. Der Prozess steuert die Ausführung der Operationen im Datenpfad und erzeugt Ausgabedaten, wobei er seinen Zustand ändert. Der Datenpfad kann Eingabedaten beispielsweise von einem anderen Prozess oder von der Außenwelt erhalten, wobei die Außenwelt wiederum durch Prozesse dargestellt werden kann.

Für die Modellierung von Prozessen in nebenläufigen Systemen gibt es formale Spezifikationsmethoden, die auf Prozessalgebren basieren. Eine Prozessalgebra ist eine präzise Sprache, um die möglichen Ausführungsschritte von Programmen zu beschreiben. Mit Hilfe von Operatoren und syntaktischen Regeln werden Prozesse aus einfachen atomaren Komponenten zusammengesetzt. Die Eigenschaften einer Prozessalgebra eignen sich beispielsweise für formale Korrektheitsprüfungen [80], in denen gezeigt werden kann, dass zwei Prozesse äquivalent sind und somit das gleiche Verhalten aufweisen. Dazu werden die Anforderungen an ein System sowohl als abstrakter als auch als detaillierter Prozess spezifiziert und durch die Anwendung von algebraischen Regeln ineinander umgeformt.

Nach [19] besteht eine Prozessalgebra aus vier Basiskomponenten. Durch eine Sprache wird ein System aus Prozessen spezifiziert, wobei das Verhalten und die einzelnen Ausführungsschritte von jedem Prozess mit Hilfe einer eindeutigen Semantik beschrieben sind. Durch Äquivalenzrelationen kann das Verhalten miteinander verglichen werden und durch eine Menge von algebraischen Regeln können Prozessspezifikationen syntaktisch umgeformt werden. Die Operatoren zur Beschreibung von Systemen sind im Wesentlichen der Präfix-Operator, der die Reihenfolge von Ereignissen festlegt. Zudem gibt es einen Alternativ-Operator, der eine von mehreren möglichen Alternativen auswählt, und einen Operator für die Parallelität, der angibt, welche Ereignisse gleichzeitig ablaufen.

In den nächsten Abschnitten werden zeitbehaftete als auch nicht zeitbehaftete Spezifikationsmethoden vorgestellt, die im Bereich von verteilten Systemen Anwendung finden. Im Vordergrund stehen dabei die allgemeinen Konzepte hinsichtlich der Handhabung von

nebenläufigen Prozessen und nicht die Äquivalenzrelationen oder die Umformungsregeln. Die zeitbehafteten Methoden eignen sich auch für die Spezifikation und Verifikation von Echtzeitsystemen.

2.5.1 Petri-Netze

Petri-Netze [14] sind Konstrukte auf grafischer Ebene, die sich u.a. zur Modellierung von verteilten Systemen sowie zur Darstellung von Nebenläufigkeit und Parallelität eignen. Petri-Netze wurden 1962 von C. A. Petri eingeführt und sind seitdem erweitert und durch zusätzliche Typen ergänzt worden. Neben der grafischen Darstellung sind Beschreibungssprachen für solche Netze hinzugekommen [56]. Ein Petri-Netz stellt einen bipartiten Graphen dar, der aus zwei Knotensorten besteht: Stellen und Transitionen. Stellen entsprechen Zwischenspeicher für Daten und werden durch Kreise dargestellt, Transitionen stellen die Verarbeitung von Daten dar und werden durch Rechtecke oder Balken symbolisiert. Stellen und Transitionen werden durch gerichtete Kanten verbunden, die als Flußrelationen bezeichnet werden, wobei eine Kante nicht zwei Knoten der gleichen Sorte miteinander verbinden darf. Stellen und Transitionen müssen einander immer abwechseln. Die Stellen vor einer Transition werden als Eingangsstellen, hinter einer Transition als Ausgangsstellen bezeichnet. Zur Beschreibung dynamischer Vorgänge werden Stellen mit Marken belegt, die durch schwarze Punkte dargestellt werden. Mit Hilfe von Schaltregeln können Marken innerhalb des Netzes bewegt werden. Beim Schalten werden Marken an den Eingangsstellen von Transitionen entfernt und gleichzeitig an allen Ausgangsstellen hinzugefügt, dabei kann sich die Gesamtzahl ändern. Das Schalten kann nur stattfinden, wenn alle Eingangsstellen einer Transition Marken enthalten. Abbildung 2.5.1 zeigt drei Ausschnitte aus einem Petri-Netz. In 2.5.1a sind nicht alle Stellen vor der Transition mit Marken belegt, so dass hier kein Schaltvorgang stattfinden kann. Erst bei einem Zustand wie in 2.5.1b ist das Schalten möglich, wodurch die Marken auf die Ausgangsstellen (Abb. 2.5.1c) übertragen werden.

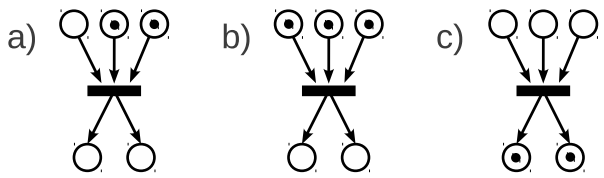


Abbildung 2.5.1: Schaltvorgänge beim Petri-Netz

Auf diese Weise können Petri-Netze mehrere parallele Prozesse innerhalb eines Graphen modellieren, wobei die Marke die gerade durchlaufene Position im Kontrollfluss darstellt. An den Stellen mit mehreren Eingangs- und Ausgangsstellen synchronisieren sich die Prozesse. Abb. 2.5.2 zeigt ein Beispiel mit zwei zyklischen Prozessen A und B, die aus den Stellen $\{s_{A0}, s_{A1}, s_{A2}\}$ bzw. $\{s_{B0}, s_{B1}, s_{B2}\}$ bestehen. Prozess B wartet auf Prozess A und kann erst fortgesetzt werden, wenn die Marke von Prozess A die Stelle s_{A1} erreicht hat.

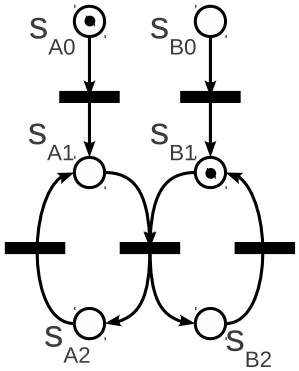


Abbildung 2.5.2: Prozesssynchronisation mit einem Petri-Netz modelliert

```
[Generate::
   a:integer;
   a := 0;
   *[true
      Consume ! a;
      a := a + 1;
   ]
]
||
[Consume::
   i:integer;
   *[true
      Generate ? i;
      ...
   ]
]
```

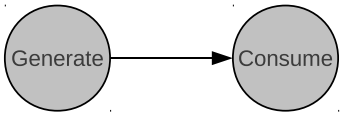


Abbildung 2.5.3: Zwei Prozesse in CSP

2.5.2 CSP

Communicating Sequential Processes (CSP) [48] von C. A. R. Hoare ist eine Hybridsprache, um konkurrente und verteilte Berechnungen zu modellieren. Hybrid bedeutet, dass CSP sowohl eine Prozessalgebra als auch eine Programmiersprache ist, bei der ein imperativer Kern mit bestimmten Befehlen erweitert wird. Die Hauptgründe für die Entwicklung von CSP war die Suche nach einer Sprache, mit der man Betriebssysteme beschreiben und gleichzeitig die Programmkorrektheit beweisen kann. Abbildung 2.5.3 zeigt zwei miteinander kommunizierende Prozesse, `Generate` und `Consume`. `Generate` erzeugt Daten und sendet diese an `Consume`. `Consume` liest diese Daten und speichert sie in der Variablen `i`. Die eckigen Klammern mit dem "`*`"- und dem `true`-Operator bilden eine Endlosschleife, die eingefassten Anweisungen werden ständig wiederholt.

Die Prozesse in CSP sind statisch, d.h. die Prozesse werden nicht zur Laufzeit erzeugt, auch die Struktur der Interprozesskommunikation ist bereits bei der Systemerzeugung bekannt. Durch spezielle Parallel-Operatoren (`||`) wird angezeigt, welche Prozesse gleichzeitig ausgeführt werden. Die Kommunikation läuft synchron und unidirektional über spezielle Eingabe- und Ausgabeanweisungen (`?` bzw. `!`) zum Empfangen und Senden ab. Beide Kommunikationspartner benennen sich dabei gegenseitig, der Datenaustausch ist also nicht an-

onym. Dies erschwert die Implementierung von Bibliotheken, weshalb Makros in der Sprache eingeführt wurden. Dadurch wurde es möglich, die Namen der Kommunikationspartner zu ersetzen.

Führt ein Prozess eine Kommunikationsanweisung aus, blockiert dieser so lange, bis der andere Prozess, mit dem kommuniziert werden soll, die entsprechende Anweisung ausführt. Versucht ein Prozess, mit einem bereits terminierten Prozess zu kommunizieren, so terminiert dieser ebenfalls. Die auszutauschenden Daten können nicht in FIFOs oder Warteschlangen gepuffert werden. Ist eine Zwischenspeicherung dennoch erwünscht, so muss diese explizit als weiterer Prozess beschrieben werden, der sich zwischen den beiden Kommunikationspartnern befindet. Ein Datentransfer etwa über globale Variablen oder einen gemeinsamen Speicher ist nicht vorgesehen.

Mit Hilfe von Input-Guards ist es möglich, Alternativ-Operatoren zu implementieren. Dazu werden mehrere Boolesche Bedingungen angegeben, wobei jeder ein Block mit Anweisungen zugeordnet wird. Wird eine Bedingung wahr, so wird nur dessen Anweisungsblock ausgeführt und die anderen Blöcke werden ignoriert. Der letzte Ausdruck solch einer Booleschen Bedingung darf eine Eingabeanweisung sein. Diese Anweisung wird erst als wahr angesehen, wenn die entsprechende Ausgabeanweisung im anderen Prozess ausgeführt wurde und Daten gelesen werden können. Somit erlauben die Input-Guards die Auswahl und Ausführung von Anweisungen abhängig von der Sendebereitschaft anderer Prozesse. Auf Sendebefehle dürfen diese Guards nicht angewendet werden, da dies ein komplexeres Protokoll für die Interprozesskommunikation erfordern würde [60]. Wenn das Verhalten eines Output-Guard benötigt wird, muss der Programmierer zusätzlichen Kommunikationsaufwand betreiben. Auflistung 2.1 zeigt ein Beispiel für die Verwendung von Input-Guards. Hier werden die drei Prozesse `A`, `B` und `C` gleichzeitig überwacht, was durch den `0`-Operator gezeigt wird. Sendet z.B. `B` zuerst Daten, so werden diese in `k` gespeichert und Anweisungsblock `Pb` ausgeführt. Anschließend wird das Programm bei `Pp` fortgesetzt.

Listing 2.1: Input-Guards in CSP

```
1  ...
2  [
3      A ? k -> Pa
4      0
5      B ? k -> Pb
6      0
7      C ? k -> Pc
8  ]
9  Pp
10 ...
```

2.5.3 CCS

Calculus of Communicating Systems (CCS) [75] wurde von Robin Milner entwickelt und modelliert wie CSP die Interaktion zwischen Prozessen. Grundprozess ist der leere Prozess *nil*, der keine Anweisungen besitzt. Neue Prozesse werden dabei aus bereits bestehenden

zusammengesetzt. Die bestehenden Prozesse können dabei entweder sequentiell, alternativ oder parallel ausgeführt werden, was durch bestimmte Operatoren (“.”, “+”, “|”) angegeben wird. In den folgenden Anweisungen wird ein Prozess P_0 definiert, der dann in P_1 verwendet wird:

$$P_0 ::= nil,$$
$$P_1 ::= \alpha.P_0.$$

Dabei führt P_1 zunächst die Aktion α aus und verhält sich anschließend wie P_0. Der “.”-Operator bewirkt also eine sequentielle Ausführung. Zusätzlich gibt es den Auswahloperator “+” und die parallele Komposition “|”. In der Anweisung

$$E ::= P + Q$$

verhält sich E entweder wie P oder wie Q, was von Aktionen innerhalb der Prozesse abhängt. Bei der Komposition

$$E ::= P|Q$$

laufen die in P und Q definierten Aktionen parallel ab. Sich wiederholende Vorgänge werden über rekursive Anweisungen definiert, wie z.B. in

$$P ::= \alpha_1.\alpha_2.P,$$

wobei zunächst die Aktion α_1, dann α_2 ausgeführt wird und sich P anschließend wiederholt.

Die Prozesse und die Kommunikationsstruktur ist wie in CSP ebenfalls statisch. Im Gegensatz zu CSP besitzen die Prozesse Ports, um mit anderen Prozessen zu kommunizieren. Zwischen zwei Prozessen können beliebig viele Ports erstellt werden, im Unterschied zu CSP, wo es nur einen Kanal gibt. Die Kommunikation läuft ebenfalls synchron und unidirektional ab. In dem folgenden Beispiel wird ein Prozess mit einem Eingangsport e und einem Ausgangsport $\overline{a}$ definiert, der durch die Überstreichung gekennzeichnet ist:

$$P ::= e.\alpha.\overline{a}.$$

Die Eingangs- und Ausgangsports von Prozessen, die mit dem “|”-Operator kombiniert werden, sind automatisch zusammengeschaltet, wenn dessen Namen identisch sind. In

$$P_S ::= A.\overline{v}.B$$

wird ein Prozess P_S definiert. P_S führt zunächst A aus, sendet dann ein Ereignis über $\overline{v}$ und führt anschließend B aus.

$$P_R ::= X.v.B$$

empfängt neben der Prozessausführung ein Ereignis über Port v. Da v und $\overline{v}$ komplementär sind, werden diese durch

$$P_a ::= P_S|P_R$$

zusammengeschaltet, wie in Abb. 2.5.4a dargestellt. Wenn ein weiterer Prozess den Empfangsport v besitzt, so wird auch dieser in die Kommunikation mit einbezogen (Abb. 2.5.4b):

$$P_{R2} ::= M.v.N$$
$$P_b ::= P_a|P_{R2}$$

Ports können mit Hilfe des "\"-Operators gesperrt werden und sind dann von außen nicht mehr sichtbar. Das hat den Vorteil, dass sie bei weiterer Komposition mit anderen Prozessen nicht mehr mit dessen Ports zusammengeschaltet werden. In dem Prozess

$$P_{a2} ::= (P_a)\backslash\{v\}$$

ist Port v nach außen hin verborgen, so dass er bei paralleler Zusammenschaltung mit P_{R2} nicht mehr verbunden wird (Abb. 2.5.4c):

$$P_c ::= P_{a2}|P_{R2}.$$

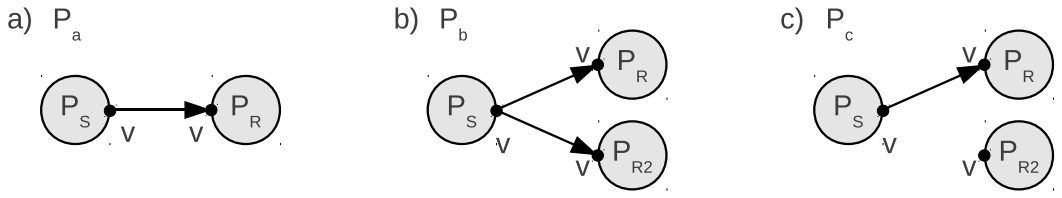


Abbildung 2.5.4: Kommunizierende CCS-Prozesse mit ihren Ports

Sollen Ports von Prozessen zusammengeschaltet werden, die unterschiedliche Namen besitzen, gibt es die Möglichkeit der Portumbenennung. Dabei werden Prozesse gekapselt und über eine Funktion (*relabeling function*) definiert, welche Ports welchen neuen Namen bekommen sollen. Zwei Prozesse P_a mit Eingangsport a und P_b mit Ausgangsport $\overline{b}$ können kommunizieren, indem z.B. a in Prozess P_a mit Hilfe einer Funktion f umbenannt wird:

$$f(a) = b,$$
$$f(l) = l.$$

Dabei wird a in b umbenannt und alle anderen Ports behalten ihren Namen. $P_a[f]$ kommuniziert nach außen über b statt a, sodass in $P_a[f]|P_b$ eine Kommunikation zwischen P_a und P_b stattfindet. Ports können auch direkt umbenannt werden ohne Definition einer Funktion. $P_a[b\backslash a]$ kapselt P_a in einem neuen Prozess, wobei a in b umbenannt wird und somit äquivalent zu $P_a[f]$ ist.

Zum Übertragen von Daten können Ports mit Hilfe von Variablen und Ausdrücken parametrisiert werden.

$$P_i ::= in(x).\overline{out}(x+1).P_i$$

empfängt über Port in einen Wert, speichert diesen in x und schreibt ihn um 1 erhöht auf Port $\overline{out}$. Die in CSP beschriebenen Guards können in CCS mit dem "+"-Operator implementiert werden.

$$P ::= (in1(k).F + in2(k).G).H$$

führt entweder F oder G aus, je nachdem, über welchen Port zuerst Daten empfangen wurden. Anschließend wird H ausgeführt. P ist äquivalent zu dem aufgeführten Beispiel des CSP-Modells auf Seite 54. Im Gegensatz zu CSP kann man nicht nur den Empfang, sondern auch das Senden von Daten an mehrere Prozesse überwachen, was auch Output-Guards ermöglicht. $P ::= (\overline{out1}(k).F + \overline{out2}(k).G).H$ sendet k entweder über Port $\overline{out1}$ oder $\overline{out2}$. Anschließend wird F oder G ausgeführt.

2.5.4 pi-Kalkül

Das pi-Kalkül ist eine Weiterentwicklung des Calculus of Communicating Systems. Im Gegensatz zu CCS ist die Struktur der Prozesse und der Kommunikation nicht statisch, sondern über die Laufzeit veränderbar. Die Kanalenden können wie Daten über andere Ports geschickt werden, was eine Umkonfigurierung ermöglicht.

2.5.5 p-Nets

p-Nets [73, 72] ist eine von F. Mayer-Lindenberg entwickelte experimentelle Programmiersprache zur Beschreibung von verteilten Systemen, die sich über mehrere Prozessoren und FPGAs erstrecken können. p-Nets ist imperativ und besitzt ein eigenes Prozessmodell sowie Zeitbedingungen für die Implementierung von Echtzeitsystemen.

Prozesse werden in Form von Automaten dargestellt, die Datenströme ein- und ausgeben sowie den Zustand von anderen Automaten überwachen. Die gesamte Applikation besteht dabei aus einem hierarchischen Netzwerk von Automaten, wobei auch Grundoperationen als elementare, zustandsfreie Automaten betrachtet werden. Die Prozess- und Kommunikationsstruktur ist statisch und alle Prozesse laufen zyklisch ab, bis die Anwendung terminiert. Neben Prozessen können auch Funktionen definiert werden.

Der Datenaustausch zwischen Prozessen kann über Datenströme realisiert werden, die als Streams bezeichnet werden. In p-Nets erscheint ein Stream als Variable, in die der Sender schreibt und aus der der Empfänger liest. Im Vergleich zu den oben erwähnten Modellen ist nur der Empfang synchron und kann den Prozess blockieren, wenn keine Daten vorhanden sind. Das Senden ist immer asynchron, was einen unendlich großen Puffer zwischen Sender und Empfänger impliziert. Weitere Möglichkeiten für die Kommunikation bestehen in Zugriffen auf einen gemeinsamen Speicher oder dem sogenannten State Sampling. Hier kann ein Prozess die Variable eines anderen Prozesses lesen und somit seinen Zustand überwachen. Dafür muss die Variable für andere Prozesse sichtbar sein. Das State Sampling ist immer asynchron. Die folgenden Anweisungen definieren einen Prozess `calc`, der eine Funktion `fak` aufruft:

```
1  b16 fct 1 1 fak {
2      -> a
3      if a=0, 1
4      else    a*fak(a-1)
5  }
6
7  cpc calc {
8      p0 -> tmp1
9      fak(p0) -> tmp2
10     tmp2 >> p1
11 }
```

Die Funktion, die mit dem Schlüsselwort `fct` gekennzeichnet ist, hat ein Argument und einen Rückgabewert vom Typ `b16`, einem Bitfeld der Breite 16. Die erste Zeile in `fak` liest das Argument in die Variable `a` ein, die letzten Anweisungen des Kontrollflusses, also entweder `1` oder "`a*fak(a-1)`" stellen die Rückgabewerte zur Verfügung. In `calc` wird zunächst der Eingabestream `p0` gelesen und das Zwischenergebnis in einer lokalen Variable gespeichert. Mit dieser wird schließlich `fak` aufgerufen, das Ergebnis in der lokalen Variable `tmp2` zwischengespeichert und anschließend auf den Ausgabestream `p1` geschrieben.

Ein Prozess kann in mehrere Subprozesse zerlegt werden, was mit Hilfe des `#`-Operators geschieht. Dies ermöglicht eine dynamische Steuerung von Prozessen aus dem Kontrollfluss heraus. Die einzelnen Subprozesse werden als Prozessgruppe bezeichnet und können parallel zueinander laufen, sofern es die Datenabhängigkeiten erlauben. Somit ist es möglich, im Gegensatz zu anderen Prozessmodellen einen Prozess nicht nur auf einem Prozessor auszuführen, sondern auf mehrere Prozessoren zu verteilen. Der folgende Prozess `pp` besteht aus drei Subprozessen `sp0`, `sp1` und `sp2`:

```
1  cpc pp {
2      #sp0
3      si -> tmp1
4      tmp1 + 4 -> tmp2
5      #sp1
6      2 * tmp2 -> tmp3
7      tmp3 >> so1
8      #sp2
9      5 * tmp2 -> tmp4
10     tmp4 >> so2
11 }
```

Eingangsdaten werden vom Eingabestream `si` gelesen und die Ergebnisse auf `so1` und `so2` geschrieben. Wegen der Datenabhängigkeiten können nur `sp1` und `sp2` parallel laufen. Intern kommunizieren `sp0` und `sp1` sowie `sp0` und `sp2` über Streams, die zwischen den Prozessen automatisch eingerichtet werden. `sp1` und `sp2` müssen warten, bis ihr Eingabestream Daten enthält, wodurch eine Synchronisation über Datenabhängigkeiten innerhalb der Prozessgruppe stattfindet.

Zusätzlich zu Eingabestreams können Prozesse durch sog. Wait-Operatoren synchro-

nisiert werden, die durch das "$$"-Symbol dargestellt sind. Der Wait-Operator überwacht Variablen und hält den Prozess so lange an, bis eine bestimmte Bedingung auf diesen Variablen erfüllt ist.

p-Nets besitzt ein Zeitmodell, welches Echtheit unterstützt und mit Hilfe des Wait-Operators realisiert wird. Der Wait-Operator kann für die Überwachung von globalen Zeitvariablen verwendet werden, die in bestimmten Zeitabständen hochgezählt werden und somit als Zeitbedingung dienen. Diese Bedingungen unterteilen einen Prozess in mehrere Zeitabschnitte. Nach dem in p-Nets verwendeten Zeitmodell werden alle Befehle innerhalb dieser Abschnitte in 0 Zeiteinheiten ausgeführt und auch alle Zustandsvariablen und Streams dort gleichzeitig aktualisiert. Somit kann mit den Wait-Anweisungen genau spezifiziert werden, wie viel Zeit man für die einzelnen Abschnitte auf realer Hardware zur Verfügung hat und nach der Implementierung überprüfen, ob Zeitbedingungen verletzt werden. Um eine bestimmte Anzahl von Zeitschritten zwischen zwei Anweisungblöcken zu warten, wird wie erwähnt der $$-Operator verwendet.

```
1    A $$10 B
```

wartet z.B. 10 Zeitschritte zwischen der Ausführung von Anweisungsblock `A` und `B`. Dabei bilden `A` und `B` Subprozesse, die allerdings nicht parallel zueinander ausgeführt werden können. Zusätzlich können Zeitbedingungen definiert werden, die die Ausführungszeit von Anweisungsblöcken festlegen.

```
1    A $$t+10 B
```

`t` ist hier die physikalische Zeit, die seit dem Start der Applikation vergangen ist. Somit wird `A` zum Zeitpunkt `t` und `B` zum Zeitpunkt `t+10` ausgeführt. Die Ausführung von `A` darf somit nicht länger als 10 Zeitschritte dauern.

2.5.6 Zusammenfassung

Dieser Abschnitt stellt die zentralen Eigenschaften der vorgestellten Prozessmodelle nochmals in einer Tabelle dar. Spalte 1 gibt Auskunft darüber, ob das Modell statisch ist oder sich die Prozess- und Kommunikationsstrukturen zur Laufzeit ändern können. Die zweite Spalte zeigt die Möglichkeiten zur Kommunikation zwischen Prozessen. Ob die Kommunikation synchron oder asynchron ist, wird in 3 dargestellt. 4 zeigt, ob die Kommunikationspartner direkt genannt werden müssen oder der Datenaustausch anonym abläuft. Die übrigen Punkte stellen dar, ob das Modell die im CSP-Abschnitt beschriebenen Guards unterstützt, bzw. ob zwischen den Kommunikationskanälen FIFOs implementiert werden können.

	1) Prozesse und IPC statisch/ dynamisch	2) IPC-Mechanismen	3) IPC-Synchronität
CSP	statisch	Kanäle	Synchron
CCS	statisch	Kanäle	Synchon
pi-Kalkül	dynamisch	Kanäle	Synchron
p-Nets	statisch	Kanäle, gemeinsamer Speicher	Streams sind senderseitig synchron und empfängerseitig asynchron, zusätzliches State-Sampling zur asynchronen Kommunikation

Tabelle 2.1: Prozessmodelle im Vergleich (a)

	4) Kommunikation anonym	5) Guards	6) FIFO zwischen Kommunikationskanälen	7) Zeitmodell vorhanden
CSP	Nein	I	-	-
CCS	Ja	I/O	-	-
pi-Kalkül	Ja	I/O	-	-
p-Nets	Ja	-	Zwischen Streams	Ja

Tabelle 2.2: Prozessmodelle im Vergleich (b)

Kapitel 3

TransC-Prozessmodell

In diesem Kapitel wird ein Prozessmodell für das TransC-Synthesewerkzeug entworfen, das sich für die Implementierung von parallel arbeitenden und miteinander kommunizierenden Modulen innerhalb von FPGAs eignet. Dabei werden zunächst alle Anforderungen an solch ein Prozessmodell herausgearbeitet und anschließend die Eigenschaften der in Abschnitt 2.5 vorgestellten Modelle analysiert und ermittelt, welche in das neue Prozessmodell übernommen werden.

3.1 Anforderungen

Das Prozessmodell sollte leicht in einer Sprache integriert werden können und für den Programmierer einfach zu benutzen sein. Die Funktionalität und Struktur muss ohne großen Aufwand beschrieben werden können, trotzdem sollte der Quellcode übersichtlich und wartbar bleiben. Als Folge dessen müssen auch ausreichend viele Möglichkeiten zur Interprozesskommunikation (IPC) und -synchronisation zur Verfügung stehen, so dass bestimmte Verhaltensweisen direkt ausgedrückt werden können und nicht durch aufwändige Hilfskonstrukte realisiert werden müssen. Da mit Hilfe des hier entwickelten Prozessmodells digitale Schaltungen synthetisiert werden, darf es keine Eigenschaften enthalten, durch die sich die Effizienz der generierten Schaltungen erheblich verschlechtert. Dies impliziert auch einen geringen Protokollaufwand bei der IPC. Da in vielen Systemen Echtzeitverhalten gefordert wird, sollte ein Zeitmodell und Sprachkonstrukte für dessen Implementierung zur Verfügung stehen. Außerdem muss das Prozessmodell flexibel genug sein und sich z.B. für die Implementierung von Bibliotheksprozessen eignen. Folgende Anforderungen werden somit in dieser Arbeit an das Prozessmodell gestellt:

- Leicht zu benutzen
- Mechanismen zur Interprozesskommunikation
- Mechanismen zur Prozesssynchronisation
- Effizient in Hardware implementierbar

- Geringer Protokollaufwand bei der Interprozesskommunikation
- Einfache Implementierung der Prozesse als Bibliothek und einfache Benutzung der Bibliotheken
- Zeitmodell zur Realisierung von Echtzeitverhalten
- Mechanismen ähnlich zu Guards in CSP zur Abfrage von synchronen Kommunikationskanälen

3.2 Prozesse

pi-Kalkül bietet ein dynamisches Prozessmodell, bei dem Prozesse zur Laufzeit erzeugt werden und sich auch die Kommunikationsstrukturen zur Laufzeit ändern können. Bei der Abbildung auf Schaltungen hat dieses Modell Nachteile, da ein Umschalten der Kommunikationswege dort zusätzliche Multiplexer erfordert, die erhöhte Signallaufzeiten und einen erhöhten Ressourcenverbrauch mit sich führen. In Hardware sind die meisten Komponenten fest miteinander verdrahtet. Aus diesem Grund soll das TransC-Prozessmodell keine dynamischen Strukturen bieten, alles wird wie in CCS oder p-Nets statisch definiert. Sollten umschaltbare Kommunikationswege notwendig sein, so kann dies mit Prozessen ggf. aus Bibliotheken realisiert werden, die diese Umschaltung durchführen. Durch die statischen Strukturen lassen sich Prozesse direkt auf Hardware-Komponenten abbilden, die Verschaltung ist durch die Kommunikationsstruktur gegeben. Bei der Verwendung einer dynamischen Prozesserzeugung, wie es auch bei Threads in Software geschieht [22, 17] wäre eine solche Abbildung nicht so einfach möglich, da die instantiierten Prozesse erst zur Laufzeit bekannt sind.

Zur leichten Benutzung sollen Prozesse in der Handhabung ähnlich wie Funktionen in imperativen Programmiersprachen sein, so dass auch Definition und Instantiierung voneinander getrennt sind. Ein Prozess wird definiert oder deklariert, eine Instanz entsteht aber erst durch den Aufruf aus dem Kontrollfluss eines anderen Prozesses, ähnlich wie in p-Nets [73]. Neben Prozessen sollen auch herkömmliche Funktionen realisierbar sein. Der Unterschied zwischen Funktionen und Prozessen liegt in der Verhaltensweise. Ein aufgerufener Prozess läuft parallel zum aufrufenden Prozess, bei einer Funktion hingegen wartet der Aufrufer auf das Beenden (Abb. 3.2.1, Aufruf von `myFunc`). Prozesse sollen wie Funktionen Argumente übergeben bekommen und Rückgabewerte zur Verfügung stellen. Wird der Rückgabewert benötigt, so hält der aufrufende Prozess an, bis der Subprozess terminiert und den Wert zur Verfügung stellt. Ein aufgerufener Prozess wird in dieser Arbeit auch als Unterprozess oder Subprozess bezeichnet. Wie in p-Nets wird der Aufruf eines Unterprozesses in den Kontrollfluss des Superprozesses integriert. Subprozesse in TransC können weitere Aufrufe auf Subprozesse besitzen, so dass man eine hierarchische Prozessstruktur erhält. Da diese Struktur statisch definiert ist, sind bereits alle Prozesse vor der Programmausführung bekannt. Somit wird ein Prozess nicht erst bei einem Aufruf erzeugt,

```
int myFunc(int a);
process int myProcA(int a);
process int myProcB(int a);

int main() {
    ...
    x=myProcA(7);
    ...
    y=myFunc(3);
    ...
    if(...) {
        z=myProcB(32);
    }
    ...
}
```

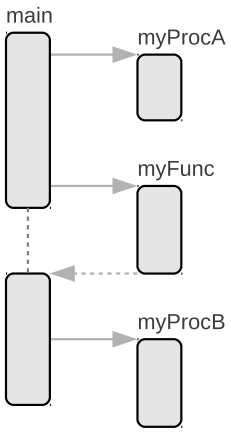


Abbildung 3.2.1: Prozesshierarchie

sondern ist schon beim Programmstart vorhanden und befindet sich in einem Wartezustand. Der Prozessaufruf bewirkt eine Aktivierung des Prozesses und das Verlassen des Wartezustands. Nach Beenden springt der Prozess wieder in diesen Zustand zurück und kann erneut aufgerufen werden. Das folgende Codebeispiel in Abb. 3.2.1 zeigt die Deklaration von zwei Prozessen und einer Funktion. Das Hauptprogramm läuft zum aufgerufenen Prozess `myProcA` weiter, auf das Beenden der anschließend gestarteten Funktion `myFunc` muss allerdings gewartet werden. Der Prozess `myProcB` wird nur gestartet, wenn die *if*-Bedingung erfüllt ist. Aufgrund des statischen Prozessmodells ist er aber instantiiert, auch wenn der Kontrollfluss dort nicht durchlaufen wird. Das Diagramm in Abbildung 3.2.1 stellt die zugehörige Prozesshierarchie dar. Da in TransC der einzige Unterschied von Prozessen zu Funktionen im Verhalten des Aufrufers liegt, ist die Funktion ebenfalls als Subroutine integriert. Die gezeigte Struktur lässt sich direkt auf Schaltungskomponenten abbilden. `main` wird in ein Modul umgesetzt, das neben Logik zur Realisierung der eigenen Operationen auch die drei Subkomponenten `myProcA`, `myProcB` und `myFunc` enthält. Diese sind wiederum als komplette Module implementiert.

3.3 Prozessinstanzen

Die Zuordnung eines Prozessaufrufs auf eine Prozessinstanz verursacht Probleme in Programmen, die den gleichen Prozess mehrmals aufrufen. So können die Prozesse von Typ `func` in Auflistung 3.1 entweder sequentiell auf einer Instanz oder parallel auf zwei Instanzen ausgeführt werden. Dies hätte nicht nur Unterschiede in der Laufzeit zur Folge, sondern könnte auch zu unterschiedlichen Ergebnissen führen, etwa wenn die Prozesse auf gemeinsamen Daten arbeiteten. Durch einen optionalen Identifizierer bei einem Aufruf kann explizit angegeben werden, welche Instanz ein Prozess belegen soll. Ist der Identifizierer bei zwei verschiedenen Aufrufen identisch, so werden die Prozesse auch derselben Instanz zugeord-

net, unterschiedliche Identifizierer haben unterschiedliche Instanzen. So bekommen z.B. die beiden `func`-Aufrufe in Auflistung 3.2 zwei Instanzen zugeteilt. Wird der Identifizierer weggelassen, ist die Zahl der Instanzen nicht definiert, wodurch man den Synthesetools mehr Freiheiten bei Optimierungen lässt.

Neben Argumenten können Prozesse auch Ressourcen wie Kommunikationskanäle übergeben bekommen, was in den nächsten Abschnitten noch genauer erläutert wird. In diesem Fall müssen bei zwei verschiedenen Aufrufen derselben Instanz die übergebenen Ressourcen immer identisch sein. Dies erleichtert die Synthese des Programms in eine digitale Schaltung. Wie die Prozessstruktur soll auch die Zuordnung der Ressourcen statisch sein, um eine effizientere Implementierung zu ermöglichen.

Listing 3.1: TransC-Beispiel mit Arraydefinitionen und Prozessaufrufen

```
1 int calc(int k, int l, int m, int n) {
2    int x,y,z;
3    int a[100];
4    x = func(a,k);
5    y = func(a,l);
6    z = m+n;
7    return x+y+z;
8 }
```

Listing 3.2: TransC-Beispiel mit Prozessaufrufen auf verschiedene Instanzen

```
1 int calc(int k, int l, int m, int n) {
2    int x,y;
3    x = func(k,l)@m1;
4    y = func(m,n)@m2;
5    return x+y;
6 }
```

3.4 IPC

Eine einfache Art der Interprozesskommunikation kann über Parameter und Rückgabewerte geschehen, da die Prozesse ähnlich wie Funktionen behandelt werden. Der aufrufende Prozess übergibt dem Subprozess vor der Ausführung Argumente. Nachdem der Subprozess die Verarbeitung abgeschlossen hat, kann der aufrufende Prozess die Ergebnisse in Form von Rückgabewerten abholen. Wenn diese Rückgabewerte für weitere Berechnungen im Superprozess benötigt werden, muss dieser warten, wodurch man auch einen einfachen Synchronisationsmechanismus erhält.

Argumente und Rückgabewerte sind somit eine sehr simple und fehlertolerante Art der Interprozesskommunikation, allerdings können Subprozesse nach dem Start nicht mehr beeinflusst oder überwacht werden. Außerdem ist die Verwendung von Rückgabewerten für zyklische Prozesse, die nicht terminieren, ungeeignet. Daher sind zusätzlich Mechanismen

wie in anderen Prozessmodellen notwendig, die auch eine Kommunikation zur Laufzeit des Subprozesses ermöglichen.

Die Handhabung von Prozessen und die Interprozesskommunikation ist in Modellen wie CSP sehr leicht zu realisieren durch die direkte Nennung des Kommunikationspartners beim Datenaustausch. Auf der anderen Seite wird das Modell dadurch sehr unflexibel, da man sich schon während der Definition des Prozesses auf einen bestimmten Kommunikationspartner festlegt. Der Einsatz von Makros zur Umgehung dieses Problems ist wiederum aufwändig. CCS bietet hier Vorteile, da die Prozesse über Ports kommunizieren. Ports mit gleichem Namen werden automatisch zusammengeschaltet. Das Problem hier ist, dass sich die Prozesse während der Definition zwar nicht auf einen Kommunikationspartner festlegen müssen, dafür aber auf einen gemeinsamen Kanal, der ebenfalls schon während der Implementierung festgelegt wird. Um dies zu umgehen, können Ports verborgen oder umbenannt werden, worunter allerdings die Übersichtlichkeit leidet. Die Kommunikation in p-Nets ist ähnlich wie in CCS, nur dass die Kommunikationskanäle als Streams bezeichnet werden. Durch Definition eines einzigen Streams, der zur Kommunikation genutzt wird, müssen sich beide Prozesse auf einen gemeinsamen Namen festlegen, wodurch dasselbe Problem wie in CCS auftritt.

In TransC werden die Kommunikationskanäle wie in p-Nets als Streams bezeichnet, die Daten eines bestimmten Typs zwischen Prozessen transportieren und dort in Ausdrücken verwendet werden. Wird ein Stream innerhalb eines Ausdrucks gelesen, so wird für die Berechnungen das im Stream enthaltene Datum verwendet. Zuweisungen an einen Stream schreiben Daten in den Kommunikationskanal hinein, wodurch sie an der anderen Seite dem Empfänger zur Verfügung stehen. Ausdrücke arbeiten somit mit den Daten des Streams und nicht mit dem Stream selber. Werden Streams an Funktionen und Prozesse übergeben, so wird der Stream in die Funktion geleitet, wenn das Argument ebenfalls ein Stream ist. Daher gelten für das Weiterleiten von Streams bestimmte Restriktionen, die in Abschnitt 3.7 näher erläutert werden.

Für eine möglichst hohe Flexibilität und Übersichtlichkeit soll die Kommunikationsstruktur erst bei der Prozessinstantiierung festgelegt werden, nicht schon zur Prozessdefinition. Die Interprozesskommunikation soll unabhängig von den Streamnamen sein, die innerhalb des Prozesses verwendet werden. Ähnlich wie Signale in VHDL soll ein Prozess lediglich mit den in seiner Schnittstelle definierten Ein- und Ausgabestreams kommunizieren. Erst bei der Instantiierung werden diese Streams in der Schnittstelle durch Verbindungsstreams mit externen Prozessen verschaltet. Ein weiterer Vorteil durch die Verwendung von Schnittstellen ist die direkte Sichtbarkeit der Kommunikationskanäle in der Deklaration ohne Kenntnis von Implementierungsdetails. Damit werden drei verschiedene Arten von Streams benötigt: Eingabestreams, Ausgabestreams und Verbindungsstreams. Neben Prozessen sollen auch Funktionen solche Kommunikationskanäle besitzen können.

Abb. 3.4.1 zeigt ein Beispiel für die Definition und Instantiierung von Prozessen mit Kommunikationskanälen. Die hier definierten Streams sind wie in den anderen Prozessmodellen synchron, die synchronen Eingabestreams werden *sistream*, synchrone Ausgabe-

```
process void procA1(sostream<int> a) {
    a = 3;
}
process void procA2(sostream<int> x) {
    procA1(x);
}
process void procB(sistream<int> y);

void main() {
    snstream n;
    procA2(n);
    procB(n);
}
```

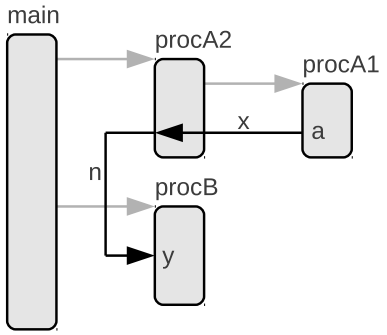


Abbildung 3.4.1: Prozesshierarchie und Kommunikationsstruktur

```
process void procA(sostream<int> x);
process void procB(sostream<int> x);


void main() {
    snstream n[16];
    procA(n);
    procB(n);
}
```

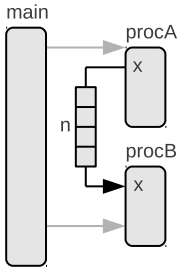


Abbildung 3.4.2: FIFO zwischen Sender und Empfänger

streams *sostream* und synchrone Verbindungsstreams *snstream* genannt. Zur einfachen Lesbarkeit sind die Definitionen an die C-Syntax angelehnt. Die Prozess- und Kommunikationsstruktur wird durch eine Grafik verdeutlicht. Graue Pfeile stehen für die Prozesshierarchie, schwarze Pfeile für die Kommunikationswege. Man kann erkennen, dass der Ausgabestream von `procA1` durch `procA2` durchgeleitet wird.

In CSP und CCS ist die IPC synchron, sowohl Sender als auch Empfänger warten auf den jeweiligen Kommunikationspartner. Da die Daten nicht gepuffert werden, muss der Austausch gleichzeitig stattfinden. Bei p-Nets ist die Kommunikation empfängerseitig synchron, aber nicht auf der Senderseite, was einen theoretisch unendlich großen Zwischenspeicher in Form einer FIFO impliziert. Dieser ist bei realen Implementierungen allerdings begrenzt, so dass auch Mechanismen zur Blockierung des Senders oder ähnliches vorhanden sein müssen. Wenn die Größe des Zwischenspeichers durch den Programmierer bestimmbar ist, können zum einen Ressourcen eingespart werden, zum anderen erhält man mehr Ausdrucksmöglichkeiten bei der Programmbeschreibung. Da die IPC zwischen Prozessen über Verbindungsstreams stattfindet, bietet sich bei diesen die Angabe der FIFO-Größe an. Außerdem wird die FIFO-Größe und ob überhaupt eine vorhanden sein soll auch hier von der Prozessdefinition entkoppelt und erst während der Instantiierung festgelegt.

In Abb. 3.4.2 wird eine FIFO mit 16 Einträgen zwischen Sender und Empfänger instan-

```
process void procA(sostream<int> x,
                   ostream<bool> a);
process void procB(sistream<int> y);

void main(ostream<bool> b) {
    snstream n;
    procA(n,b);
    procB(n);
}
```

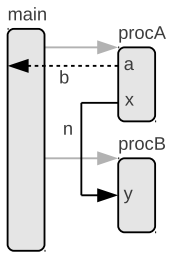


Abbildung 3.4.3: Asynchrone und synchrone Streams

tiiert. Ohne der Angabe einer FIFO-Größe muss die Kommunikation, d.h. die Übertragung eines Datums auf beiden Seiten gleichzeitig stattfinden, da kein Zwischenspeicher vorhanden ist.

p-Nets bietet zusätzlich zu der synchronen Kommunikation State-Sampling an, eine asynchrone Kommunikationsart, bei der bestimmte Variablen eines Prozesses von einem anderen Prozess gelesen werden können. Auch hier werden interne Variablen eines Prozesses direkt referenziert, wodurch die Kommunikation bereits bei der Prozessdefinition festgelegt wird. Der asynchrone Kommunikationsmechanismus soll in das TransC-Modell übertragen werden, dabei bietet sich die Erweiterung des Stream-Konzeptes auf asynchrone Streams an, die die gleiche Wirkung wie das State-Sampling in p-Nets haben. Die Verschaltungsarten und die Art der Definition kann hier direkt von den synchronen TransC-Streams übernommen werden, der einzige Unterschied wäre, dass Zugriffe keine blockierende Wirkung haben und für asynchrone Verbindungsstreams keine Zwischenspeicher definiert werden können. Statt *sistream* usw. werden diese mit *istream*, *ostream* bzw. *nstream* bezeichnet. In Abb. 3.4.3 werden asynchrone Streams als gestrichelte Linien dargestellt. Der Ausgabestream von `procA` wird durch `main` durchgeleitet. Neben der Überwachung von Prozessen eignen sich asynchrone Streams auch zum Abtasten oder zur Erzeugung von Signalen zur Kommunikation mit der Außenwelt.

p-Nets bietet auch eine Methode der Interprozesskommunikation durch die Verwendung eines gemeinsames Speichers an, was in TransC übernommen werden kann. Da die gesamten Seiteneffekte in der Deklaration eines Prozesses sichtbar sein sollen, werden auch diese Übergaben in die Schnittstelle des Prozesses integriert. Hier kann sich an der C-Syntax und dessen Arrayübergaben an Funktionen orientiert werden. Bei dem gleichzeitigen Zugriff von mehreren Prozessen auf dasselbe Array soll nur ein Prozess den Zugang erhalten und alle anderen so lange blockieren, bis die Arrayoperation abgeschlossen ist. Für die Synchronisation bei mehreren parallelen Schreibzugriffen ist wie bei der Threadprogrammierung in Software der Programmierer zuständig.

3.5 Synchronisierung

In p-Nets sind nicht alle Prozesse sofort bei Programmstart aktiv, vielmehr werden Subprozesse wie erwähnt aus dem Kontrollfluss von Superprozessen aus gestartet. Dadurch kann

ebenfalls eine einfache Synchronisation erzielt werden, was z.B. bei CSP oder CCS auf diese Weise nicht möglich ist. Da auch im TransC-Modell Subprozesse aus dem Kontrollfluss heraus aufgerufen werden, kann diese Art der Synchronisierung dort auch angewendet werden.

Bei allen in Kapitel 2.5 diskutierten Prozessmodellen kann eine Synchronisation unter anderem über die Kommunikationskanäle erzielt werden. In CSP oder CCS muss dafür nicht unbedingt ein Datenaustausch stattfinden, allein das Senden oder der Empfang eines Ereignisses reicht aus. Da in TransC der zu übertragene Datentyp mit angegeben wird, kann das Senden eines Ereignisses mit Hilfe von Streams vom Typ *void* erreicht werden. In C ist *void* ein leerer Datentyp, der keine Informationen enthält. Synchrone Streams vom Typ *void* in TransC übertragen dementsprechend keine Daten.

In Software-Threadbibliotheken gibt es Mechanismen, bei denen eine Routine auf das Terminieren eines Threads warten muss, bevor sie fortgesetzt werden kann. In CSP oder CCS muss dies über IPC geschehen. In TransC erhält man diese Funktionalität dadurch, dass eine Routine die Rückgabewerte eines Subprozesses für weitere Berechnungen benötigt. In diesem Fall blockiert die Routine, bis der Prozess beendet ist. Hat ein Prozess keine Rückgabewerte oder werden diese nicht benutzt, so kann auf den Prozess mit der Funktion *sync* gewartet werden. *sync* wird zusammen mit der Bezeichnung für die Prozessinstanz aufgerufen. Ist der Prozess bereits terminiert, so hat *sync* keine blockierende Wirkung. *sync* kann auch auf mehrere Prozesse gleichzeitig warten, wie in dem folgenden Codebeispiel gezeigt.

```
1 void main() {
2    procA() @ inst1;
3    procB() @ inst2;
4    ...
5    sync inst1, inst2;
6    ...
7 }
```

Prozesse ohne Rückgabewerte können direkt mit *sync* aufgerufen werden (`sync procA()`). In diesem Fall wird sofort auf das Beenden gewartet, die Prozesse verhalten sich dann wie eine Funktion.

Konstrukte wie Semaphore oder Mutexe sind im Prozessmodell nicht vorhanden. Zum Betreten eines kritischen Abschnittes wie einem gemeinsamen Speicherbereich muss daher auf Synchronisationsmechanismen wie den Algorithmen von Dekker[27] oder Peterson[86] zurückgegriffen werden, die die gemeinsame Nutzung einer Ressource zwischen zwei Prozessen steuern. In generalisierter Form können sie auch mit mehr als zwei Prozessen umgehen [71]. Die in diesen Algorithmen verwendeten Variablen können mit Hilfe von gemeinsamem Speicher oder gemeinsamen Datenstrukturen, die sich die Prozesse teilen, implementiert werden.

3.6 Guards

In CCS werden Guards mit Hilfe des "+"-Operators implementiert, wie im folgenden Programmausschnitt dargestellt.

```
1 P = a.b.c + x.y.z
```

Wenn `a` und `x` Kommunikationskanäle sind, so werden entweder die Anweisungen `b.c` oder `y.z` ausgeführt, je nachdem, welcher Kanal zuerst benutzt werden kann. Dies gilt sowohl für Ein- als auch für Ausgaben. Wie erwähnt unterstützt CSP nur Guards für Eingaben, in p-Nets sind solche Mechanismen nicht direkt vorhanden, können aber mittels State-Sampling implementiert werden. Nach [60] ist die Umsetzung des CSP-Modells mit den Input-Guards bereits nicht trivial, eine Erweiterung auf Output-Guards würde einen erheblichen Mehraufwand mit sich führen, was der Forderung eines geringen Protokollaufwands im TransC-Modell widerspricht. Da es sich hier aber um eine wichtige Eigenschaft handelt und auch die Implementierung von zusätzlichen Kommunikationswegen zum Erreichen dieser Funktionalität einen höheren Ressourcenaufwand bedeutet, sollen diese Konstrukte unterstützt werden. Ein guter Kompromiss stellt die Möglichkeit dar, den Status von synchronen Streams abfragen zu können, um herauszufinden, ob der nächste lesende bzw. schreibende Zugriff blockiert. Durch die mit synchronen Streams verbundene Prozesssynchronisation sind implizit bereits Handshake-Signale vorhanden, die noch explizit im Modell sichtbar sein müssen. Der Status von synchronen Streams soll mit der Funktion *block* abgefragt werden. *block* gibt *true* zurück, wenn der nächste Zugriff eine blockierende Wirkung hat. Durch die Verbindung mit if-Anweisungen kann eine ähnliche Funktionalität wie in dem Beispiel mit den CCS-Ausdrücken erreicht werden, wie der folgende Quellcode zeigt.

```
1 if(!a.block()) {
2     ...
3 }
4 else if(!b.block()) {
5     ...
6 }
```

`a` und `b` sind synchrone Streams, deren Zustand abgefragt wird. Wenn Daten zum Lesen vorhanden sind bzw. Platz für zu schreibende Daten verfügbar ist, so gibt *block* $false$ zurück. Durch das Ausrufezeichen wird das Ergebnis invertiert, so dass der Code in den geschweiften Klammern dann ausgeführt wird. Der Unterschied zu CCS ist, dass die Abfragen der beiden Kommunikationskanäle nicht parallel geschehen und ggf. ein Lesen oder Schreiben des Streams auch erst zu einem späteren Zeitpunkt als der Ausführung der *block*-Funktion stattfindet. Dafür ist die Implementierung dieser Funktionalität in Hardware ohne Mehraufwand möglich.

3.7 Link-Typen

Wie in den vorigen Abschnitten deutlich wurde, können Streams verschiedene Arten von Quellen und Senken haben. Sie können innerhalb von Ausdrücken gelesen oder geschrieben bzw. an andere Streams von Sub- oder Superprozessen geleitet werden. *Link* wird in dieser Arbeit als Oberbegriff für Quelle oder Senke benutzt. Wenn ein Stream von einem arithmetischen Ausdruck in einem Prozess gelesen wird, so besitzt er eine Senke vom Typ *EXP_ RD*, bei Schreiboperationen eine Quelle vom Typ *EXP_ WR*. Wenn ein Stream mit einem Ausgabe- oder Eingabestream eines Subprozesses verbunden ist, so wird sein Link als *SUB_ WR* bzw. *SUB_ RD* bezeichnet. Wird ein Stream hingegen nach außen zum Superprozess weitergeleitet, so werden Quelle und Senke *SUPER_ WR* oder *SUPER_ RD* genannt. Links vom Typ *SUPER_ WR* und *SUPER_ RD* können nur bei Ein- und Ausgabestreams auftreten. Wie die gesamte Prozessstruktur sind auch die einem Stream zugeordneten Linktypen statisch und können zur Laufzeit nicht verändert werden.

Listing 3.3: Beispiel für einen Prozessaufruf mit Streams

```
1  process int calc(ostream<int> tt, istream<int> yy) {
2     ...
3  }
4  void main(ostream<int> str, istream<int> inp) {
5     nstream<int> val;
6     int a, b;
7     str = 5;
8     a = calc(val, inp);
9     ...
10    b = val;
11    ...
12 }
```

Auflistung 3.3 zeigt Beispiele für die verschiedenen Linktypen. Die Hauptfunktion ruft einen Subprozess `calc` auf, der einen asynchronen Ausgabestream `tt` und Eingabestream `yy` in seiner Schnittstelle besitzt. In der `main`-Funktion wird ein Verbindungsstream `val` definiert, mit dem Ausgabestream von `calc` verbunden und von einer Operation in Zeile 10 gelesen. Dementsprechend besitzt `val` eine Quelle vom Typ *SUB_ WR* und eine Senke vom Typ *EXP_ RD*. In der Argumentenliste besitzt `main` einen Eingabestream *inp*, der direkt mit *yy* von *calc* verbunden ist. Da `inp` von einem Superprozess geschrieben und von einem Subprozess gelesen wird, besitzt er die Linktypen *SUPER_ WR* und *SUB_ RD*.

Tabelle 3.1: Maximale Anzahl von Sendern und Empfängern

	sync istr.	async istr.	sync ostr.	async ostr.	sync nstr.	async nstr.
wr	0	0	$<= 1$	$<= 1$	$<= 1$	$<= 1$
rd	$<= 1$	∞	0	∞	∞	∞

Die Anzahl der maximal möglichen Quellen und Senken ist je nach Streamtyp begrenzt. Es macht beispielsweise keinen Sinn, in einen Eingabestream, der Daten liefert, zu schreiben oder zwei verschiedene Instanzen von Subprozessen als Quelle mit einem Stream zu verbinden. In letzterem Fall ist es besser, in einem weiteren Prozess genau zu spezifizieren wie die Streams verschmolzen werden sollen. Tabelle 3.1 gibt Auskunft über die erlaubte Anzahl von Sendern (`wr`) und Empfängern (`rd`), wobei `wr` und `rd` folgendermaßen definiert sind:

$\texttt{wr} = \min\{1, \texttt{exp}_{\texttt{wr}}\} + \texttt{sub}_{\texttt{wr}}$,

$\texttt{rd} = \min\{1, \texttt{exp}_{\texttt{rd}}\} + \texttt{sub}_{\texttt{rd}}$.

$\texttt{exp}_{\texttt{wr}}$ ist die Anzahl der Sender vom Typ *EXP_ WR* usw. Die Linktypen *SUPER_ WR* und *SUPER_ RD* sind weggelassen worden, da diese nur bei Ein- und Ausgabestreams relevant sind. Diese Streams haben immer genau einen Link, der vom Superprozess benutzt wird. Die genaue Anzahl der Links vom Typ *EXP_ RD* bzw. *EXP_ RD* ist irrelevant. Da sie nur von Ausdrücken benutzt werden und die Ausdrücke so abgearbeitet werden können, dass die Zugriffe nicht gleichzeitig stattfinden, ist lediglich wichtig, ob es überhaupt Links von diesem Typ gibt. Somit kann eine Zahl von Zugriffen, die größer als 0 ist, immer auf 1 reduziert werden. Synchrone Streams dürfen generell nicht mehr als einen Sender oder Empfänger haben, da sonst Multiplexer zur Umschaltung benötigt werden würden, wodurch die Effizienz sinkt. Die einzige Ausnahme sind Verbindungsstreams, da dort für jeden Empfänger eine eigene FIFO generiert werden kann und somit der Sender blockiert, wenn mindestens eine der Empfänger-FIFOs voll ist. Synchrone Verbindungsstreams und asynchrone Streams generell können eine beliebige Anzahl von Senken haben, selbst asynchrone Ausgabestreams. Hier wird der zuletzt geschriebene Wert dann wieder gelesen. Wenn ein synchroner Eingabestream von zwei Ausdrücken gelesen wird und an einen empfangenden Subprozess weitergeleitet wird, berechnet sich `rd` wie folgt: $\texttt{rd} = \min\{1, 2\} + 1 = 2$. Nach Tabelle 3.1 überschreitet `rd` hier die maximal erlaubte Linkanzahl von 1.

Es können fast alle Quellen- und Senkentypen miteinander kombiniert werden. Tabelle 3.2 zeigt die Möglichkeiten und auf welche Streamtypen diese Kombinationen angewendet werden können. Dabei kann man wieder erkennen, dass Ausgabestreams immer dann benötigt werden, wenn Daten zum Superprozess geschickt werden, und Eingabestreams, wenn Daten vom Superprozess kommen. Sobald Streams von Ausdrücken innerhalb eines Prozesses benutzt werden oder Subprozesse miteinander kommunizieren sollen, werden Verbindungsstreams verwendet.

Tabelle 3.2: Kombination von Link-Typen

	EXP_RD	SUPER_RD	SUB_RD
EXP_WR	nstream	ostream	nstream
SUPER_WR	istream	-	istream
SUB_WR	nstream	ostream	nstream

3.8 Zeitmodell

Das in TransC verwendete Zeitmodell ist ähnlich zu dem von p-Nets. Während in p-Nets alle Operationen zwischen den $$-Operatoren simultan ausgeführt werden, laufen in TransC die Operationen innerhalb eines Grundblocks gleichzeitig ab, Folgeblöcke werden einen Zeitschritt später ausgeführt. Dies ist ein theoretischer Wert, da real die Ausführung aller Operationen länger dauern kann. Bei der Implementierung werden daher die Operationen so angeordnet und nach Möglichkeit parallelisiert, um die Ausführungszeit minimal zu halten. In Relation zu einem Block bb_n wird der Nachfolgeblock bb_{n+1} somit immer mindestens einen Zeitschritt später ausgeführt.

Die Streamkommunikation zwischen zwei Prozessen benötigt ebenfalls einen Zeitschritt. Der Empfänger erhält den Wert somit einen Takt nach der Sendeoperation. Asynchrone Streams werden alle gleichzeitig gelesen und aktualisiert, so dass zwei Lesezugriffe auf denselben Stream in einem Block immer dasselbe Ergebnis abholen und bei zwei aufeinanderfolgenden Schreibzugriffen der erste ignoriert wird, da er vom zweiten überschrieben wird. Beim Lesen und Schreiben von synchronen Streams liegt zwischen zwei Zugriffen ein infinitesimal kleiner Zeitabstand. Hier müssen alle Zugriffe berücksichtigt werden, da Schreibzugriffe ggf. eine FIFO füllen und Lesezugriffe diese leeren. Außerdem besitzen alle Zugriffe eine blockierende Wirkung.

Alle Operationen in einem Grundblock werden zwar simultan in einem Zeitschritt ausgeführt, jedoch steht nicht immer fest, welche Operationen letztendlich in einem Grundblock zusammengefasst sind und welche nicht. Dies kann sich auch durch Graphentransformationen und Optimierungen ändern. Wenn zwei Operationen nicht gleichzeitig ausgeführt werden sollen, werden sie durch einen `sync`-Operator getrennt. `sync` verhält sich wie eine Barriere und verhindert die parallele Ausführung von Operationen. Alle Anweisungen hinter dem `sync` werden in den nächsten Block und somit in den nächsten Zeitschritt verschoben. Grundblöcke, die durch `sync` voneinander getrennt wurden, können durch Optimierungen nicht mehr miteinander verschmolzen werden, da sich sonst das spezifizierte Zeitverhalten ändern würde. Da der `sync`-Operator quasi einen Wartezyklus von einem Zeitschritt darstellt, können mit dessen Hilfe Wartebefehle erzeugt werden. Mit folgenden Anweisungen z.B. wird mindestens 100 Zeitschritte gewartet:

```
1 for(i=0; i<100; ++i) sync;
```

Wie erwähnt kann durch zusätzliche Berechnungen die Wartezeit in tatsächlichen Implementierungen länger sein, da dort die Ausführungszeit von Blöcken von einem Zeitschritt nicht eingehalten werden kann. Die Ausführungsdauer von einem Zeitschritt für jeden Block ist somit ein Mindestwert, der außer bei kleinen Datenflussgraphen leicht überschritten werden kann.

Genaue Zeitangaben werden mit dem `time`-Operator erreicht. Mit `time` werden Funktionen instantiiert und ausgeführt. Jede Instanz besitzt intern einen eigenen Zähler, der jeden Zeitschritt inkrementiert wird. `time(1)` setzt den Zähler zurück, `time(0)` gibt den

aktuellen Wert. Da `time` wie eine Funktion behandelt wird, werden alle Anweisungen vor und nach dem Aufruf voneinander getrennt. Außerdem können mit Hilfe des @-Operators mehrere Instanzen erzeugt werden, wodurch die Benutzung von verschiedenen Zählern möglich ist. Die Schleife in den folgenden Anweisungen hat eine Ausführungsdauer von 100 Zeiteinheiten:

```
1 time(1);
2 while(time(0)<100){}
```

`time` ist vergleichbar mit dem $$-Operator in p-Nets, wobei die Anweisungen

```
1 time(1);
2 A
3 while(time(0)<d){}
4 B
```

äquivalent sind zu den p-Nets Ausdrücken

```
1 A
2 $$t+d
3 B
```

`A` und `B` sind dabei Anweisungsblöcke, die durch Zeitoperationen voneinander getrennt werden. Zeitbedingungen können mit Hilfe von `assert` gesetzt werden. Durch

```
1 time(1);
2 A
3 assert(time(0)<100);
```

wird festgelegt, dass der Anweisungsblock `A` nicht länger als 100 Zeiteinheiten zur Ausführung benötigen darf.

3.9 Zusammenfassung

Ein TransC-Prozess ist die Gesamtheit aller Operationen und ihren Datenabhängigkeiten, die einem Kontrollfluss zugehören. Diese Operationen beinhalten das Aufrufen von weiteren Subprozessen, wodurch eine hierarchische Prozessstruktur entsteht und der Start eines Subprozesses abhängig von dem Kontrollfluss des Superprozesses wird. Die Prozessstruktur ist statisch und bereits beim Programmstart bekannt. Im Gegensatz zu Funktionen laufen Subprozesse parallel zueinander und zum aufrufenden Prozess, wobei durch spezielle `sync`-Konstrukte oder durch Datenabhängigkeiten eine sequentielle Ausführung zwischen Prozessen erzwungen werden kann. Prozessaufrufe können explizit bestimmten Prozessinstanzen zugeordnet werden.

Die Prozesse kommunizieren über synchrone und asynchrone Kanäle, die als Streams bezeichnet werden. Ein Sendeprozess schreibt Daten in den Stream, die vom Empfangsprozess gelesen werden. Synchrone Streams haben eine blockierende Wirkung, die den Prozess anhalten, wenn keine Daten geschrieben bzw. gelesen werden können. Mit Hilfe der synchronen Kanäle können neben Daten auch lediglich Ereignisse übertragen werden. Wie

die Prozessstruktur ist auch die Kommunikationsstruktur statisch. Die Kommunikationsstruktur wird zur Erhöhung der Flexibilität nicht bei der Implementierung eines Prozesses bestimmt, sondern erst bei dessen Instantiierung, so dass bei der Implementierung noch nicht bekannt ist, über welche Kanäle oder mit welchen Prozessen Daten ausgetauscht werden. Ein Prozess besitzt eine Schnittstelle, die u.a. Eingabe- und Ausgabestreams enthält. Um diese Ein- und Ausgabestreams miteinander zu verbinden und auch von Ausdrücken in Superprozessen verwenden zu können, werden Verbindungsstreams benutzt. Die Streams besitzen bestimmte Quellen- und Senkentypen, die als Links bezeichnet werden. Zur Pufferung von Daten können die Verbindungsstreams FIFO-Zwischenspeicher enthalten. Neben der Kommunikation über Streams können auch über Argumente und Rückgabewerte Daten ausgetauscht werden, die ein Prozess beim Aufruf erhält und beim Beenden zurückgibt.

Abbildung 3.9.1 zeigt ein Beispiel für eine Prozess- und Kommunikationsstruktur. Die großen Rechtecke stellen einzelne Prozesse dar, die einen sequentiellen Kontrollfluss enthalten. Von diesem Kontrollfluss aus können weitere Subprozesse gestartet werden. So sind Prozess p1 und p2 Subprozesse von p0 und Prozess p3 ein Subprozess von p1. Prozess p0 schickt über einen Stream Daten an p1, wenn er bestimmte Zustände im Kontrollfluss erreicht. Außerdem empfängt Prozess p0 Daten von p3. Da Prozess p3 Subprozess von p1 ist, muss letzterer die Daten weiterleiten, wofür er eine zusätzliche Schnittstelle bereitstellt. Die einzelnen Streams sind von $s0$ bis $s4$ durchnummeriert. $s0$ wird z.B. in mehreren Ausdrücken in $p0$ geschrieben und an $p1$ weitergeleitet, wodurch er ein Verbindungsstream mit den Links EXP_WR und SUB_RD ist. $s3$ wird von einem Prozess geschrieben und zum Superprozess $p1$ weitergeleitet, wodurch er einen Ausgabestream mit den Links SUB_WR und $SUPER\_RD$ darstellt.

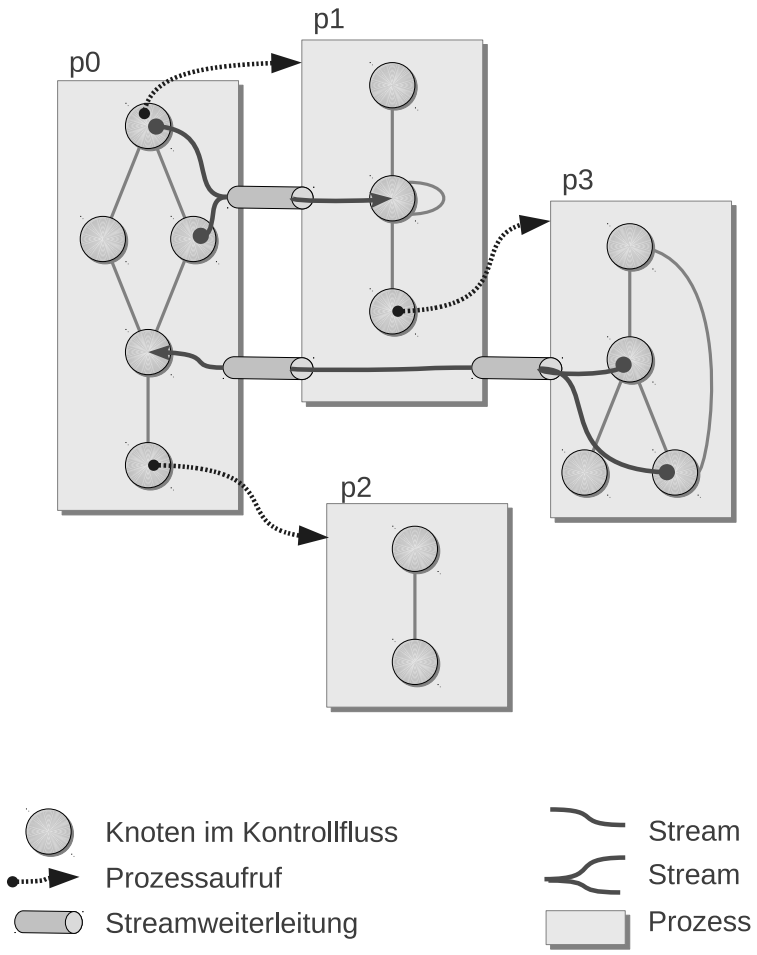


Abbildung 3.9.1: Prozess- und Kommunikationsstruktur eines TransC-Programms

Kapitel 4

TransC-Sprache

Das im vorigen Kapitel vorgestellte Prozessmodell wird zu einer Programmiersprache erweitert, mit der die Automatennetzwerke implementiert werden können. Die Notation zur Darstellung des Prozessmodells ist bereits an die Programmiersprache C angelehnt, Notationen für Prozessdefinitionen und Prozessaufrufe sowie Kommunikationsprimitiven sind vorhanden und werden direkt in die Sprache übernommen. Benötigt werden noch Grunddatentypen, Basisoperationen sowie Strukturen zur Definition des Kontrollflusses innerhalb eines Prozesses. Die Sprache soll in der Lage sein, imperative Paradigmen ausdrücken zu können. Objektorientierte Konstrukte werden nicht verwendet. Auch hierfür wird die Syntax der Programmiersprache C übernommen, da diese sehr weit verbreitet ist und auch andere Sprachen wie C++ oder Java an diese angelehnt sind [96].

Die folgenden Abschnitte stellen die Sprachkonstrukte mit ihren Restriktionen und Erweiterungen zu ANSI-C vor, abschließend werden die Mechanismen zur Prozessdefinition und Interprozesskommunikation in TransC mit anderen C-basierten Sprachen zur Hardwarebeschreibung verglichen. Die parallelen Konstrukte werden zudem der imperativen Programmiersprache Go [1] gegenübergestellt. Go wurde zur effizienten Programmierung von Mehrkernprozessoren und vernetzten Rechnerverbünden entwickelt und besitzt damit eine Vielzahl von Mechanismen zur konkurrenten Programmierung.

4.1 Sprachkonstrukte

Die TransC-Sprache besteht aus einer Untermenge der Programmiersprache C, die mit bestimmten Programmierkonzepten erweitert ist. Die wichtigsten Erweiterungen sind Konzepte zur Erstellung von parallelen Prozessen und der Kommunikation zwischen diesen. Die folgende Liste zeigt die Erweiterungen sowie die Einschränkungen im Vergleich zu C:

- Erweiterungen
  - Definition von mehreren Prozessen möglich
  - Synchrone sowie asynchrone Kommunikationskanäle zwischen Prozessen
  - bitgenaue und Boolesche Datentypen

  - mehrere Rückgabewerte bei Funktionen

- Restriktionen
  - Pointer
  - Fließkomma-Datentypen
  - C-Bibliotheksfunktionen
  - rekursive Funktionsaufrufe

In den folgenden Abschnitten werden die wichtigsten Spracheigenschaften vorgestellt, die sich von ANSI-C unterscheiden.

4.1.1 Erweiterungen und Restriktionen zu C

Prozesse und Funktionen

Die Definition von Prozessen und Funktionen orientiert sich, wie in der Beschreibung des Prozessmodells dargestellt, an der Programmiersprache C. Prozesse bekommen zusätzlich das Schlüsselwort `process` am Anfang, um sie von Funktionen unterscheiden zu können. Im Vergleich zu C sind jedoch mehrere Rückgabewerte möglich, die vor dem Funktionsnamen aufgelistet und durch Kommata getrennt werden:

```
1 int func1();
2 (int, int) func2();
```

Dementsprechend werden auch die `return`-Anweisungen sowie die Aufrufe dargestellt, wobei nicht alle Rückgabewerte verwendet werden müssen. Innerhalb von arithmetischen Ausdrücken können nur Funktionen oder Prozesse mit einem Rückgabewert verwendet werden:

```
1 a = func1();
2 (b) = func1();
3 (c,d) = func2();
4 (e, ) = func2();
5 return 7, 3;
```

Datentypen

Da TransC zur Erzeugung von digitalen Schaltungen entwickelt wurde, werden bitgenaue Datentypen unterstützt. So gibt es neben dem 32-Bit großen Integer-Typen auch bitgenaue Integer wie z.B. `int8` oder `int7`. Zusätzlich kann die Breite in spitzen Klammern angegeben werden, wodurch sich auch anpassbare Breiten definieren lassen. Allerdings muss die Breite immer zur Kompilierzeit bekannt sein.

```
1 const int ttt = 3;
2 const int width = 17+ttt;
```

```
3 void main() {
4     int3 a;
5     int<3> b;
6     int<width> c;
7     ...
8 }
```

Unterstützt werden, wie bereits gezeigt, Integer beliebiger Breite (`int`), vorzeichenlose Integer beliebiger Breite (`uint`) sowie der Boolesche Datentyp (`bool`). Zur Kompatibilität zu ANSI-C sind auch die dort verwendeten Integertypen wie `char` oder `unsigned` möglich. Bei Zuweisungen zwischen grundlegenden Datentypen wie `uint`, `int` und `bool` werden automatisch Casts durchgeführt. Bei der Zuweisung etwa von einem `int7` an einen `int10` wird der Bitvektor unter Beachtung des aktuellen Vorzeichens vergrößert, bei kleineren Datentypen wie etwa `int4` hingegen verkleinert. Auch `int` und `uint` werden ineinander konvertiert nach den selben Regeln wie in der C-Programmiersprache. Bei der Zuweisung an eine Boolesche Variable wird ermittelt, ob der Bitvektor ungleich null. Bei den Basisoperationen können alle Datentypen als Operanden verwendet werden, auch in gemischter Form.

Neben den einfachen Datentypen können mehrdimensionale Arrays und Datenstrukturen deklariert werden. Auch bei Arrays muss die angegebene Größe entweder zur Kompilierzeit berechenbar sein oder vollständig weggelassen werden (`int a[];`). In diesem Fall erhält das Array eine Standardgröße mit einer Adresstiefe von 16-Bit. Datenstrukturen können aus allen definierbaren Datentypen bestehen und bis zu einer beliebigen Tiefe verschachtelt sein. Die Definition und Benutzung ist identisch zu ANSI-C, allerdings kann das Schlüsselwort `struct` bei der Instantiierung weggelassen werden:

```
1 struct A {
2    int a;
3    int buf[13];
4 };
5 void main() {
6     struct A myStruct1;
7     A myStruct2;
8     ...
9 }
```

Arrays werden bei Funktionsaufrufen immer per Referenz übergeben. Da keine Pointer möglich sind, können Datenstrukturen nicht als solche in der Argumentliste von Prozessen stehen. Stattdessen wird hier der Referenzoperator wie in C++ verwendet:

```
1 process void calc(A &a);
```

Über `typedef` kann ein Alias von bereits bestehenden Datentypen angelegt werden. Neben den grundlegenden Typen wie Integer und Boolean ist auch die Ableitung von Datenstrukturen und Arrays möglich. Im Vergleich zur C-Sprache sind die Alias-Typen allerdings strikter typisiert. Bei Funktionsaufrufen können z.B. keine Integer übergeben

werden, wenn in der Argumentenliste ein vom Integer abgeleiteter Typ deklariert ist. In diesem Fall muss vorher ein Cast durchgeführt werden.

Fließkommatypen sind in der Sprache nicht vorhanden, da Fließkommaarithmetik komplexer als Integer- oder Festkommaarithmetik ist und wesentlich mehr Hardwareressourcen benötigt. In vielen Fällen können Fließkommatypen vermieden bzw. durch Festkommatypen ersetzt werden. Werden Fließkommatypen dennoch benötigt, so können diese über `typedef` definiert und entsprechende Funktionen für die arithmetischen Operationen bereitgestellt werden. Eine Spracherweiterung sähe dann wie folgt aus, wobei die Prozesse in TransC nur als Deklaration vorliegen, die eigentliche Implementierung jedoch als handoptimiertes Hardwaremodul vorhanden ist. Handelt es sich bei dem Prozess um eine gepipelinte Schaltung und nicht um ein Modul, das sich mittels Handshaking synchronisiert, so wird dies dem Compiler über ein Pipeline-Pragma mitgeteilt, wie beim Prozess `f32_mul` oder `f32_add` gezeigt.

```
1  typedef uint32 float;
2
3  process float f32_gen(int32 m, uint32 n);
4
5  #pragma pipeline 4
6  process float f32_mul(float a, float b);
7
8  #pragma pipeline 6
9  process float f32_add(float a, float b);
10
11 process bool f32_less(float a, float b);
12 process bool f32_equal(float a, float b);
13 ...
14
15
16 double a,b,c;
17 b = f32_gen(-45, 1448);
18 c = f32_mul(a,b);
19 ...
```

Für spätere Compilerversionen ist wie in C++ eine Operatorüberladung geplant, so dass Operatoren an Stelle von Prozessaufrufen verwendet werden können. Die Fließkommaeinheiten stehen nur dem Prozess zur Verfügung, in dem sie instantiiert worden sind. Sollen die Einheiten von mehreren Prozessen aus angesprochen werden, müssen zusätzliche Kommunikationsstrukturen, z.B. in Form von synchronen Streams bereitgestellt werden. Im Ausblick in Abschnitt 9 werden kurz Sprachkonstrukte beschrieben, die Funktionen in Objekte einbetten, welche dann weiteren Funktionen übergeben werden können. Dies wäre auch auf die hier beschriebenen Module anwendbar.

Pointer

Die Verwendung von generischen Pointern, die auf beliebige Arrays eines Programms zeigen können, würde einen gemeinsamen Speicherbereich für solche Arrays voraussetzen, was sich negativ auf die Effizienz der erzeugten Schaltungen auswirkt. Die meisten Pointer können allerdings durch mehrere Rückgabewerte bei Funktionsaufrufen sowie die Verwendung von Arrayreferenz und Index ersetzt werden.

Folgender Code zeigt eine Funktion mit Pointern:

```
1 int func(int *val, int index, int *array) {
2     array[index] = *val;
3     *val = array[2];
4     return 3;
5 }
```

Diese kann wie unten aufgeführt modifiziert werden, indem bei Pointern auf ein Array immer ein Index mitgeführt wird und Pointer auf einfache Variablen als Rückgabewert geschrieben werden:

```
1 (int,int) func(int val, int index,
2                 int array[], int i_array) {
3     array[i_array+index] = val;
4     return 3, array[i_array+2];
5 }
```

In weiteren Ausbaustufen der Sprache könnten die oben aufgeführten einfachen Pointer, die immer nur ein Array referenzieren, erlaubt sein, wobei die gezeigte Transformation automatisch geschieht.

Operationen

In TransC werden alle arithmetischen und logischen Operationen wie in C [12] unterstützt. Dabei werden die Operationen so durchgeführt, dass keine Stellen des Ergebnisses verloren gehen. Zahlen besitzen immer die minimal erforderliche Breite, um ihren Wert zu speichern. So benötigt eine 8 beispielsweise 4-Bit oder eine -3 3-Bit. Zusätzlich zur eigentlichen Zahl wird der Typ gespeichert, also Integer oder Unsigned Integer. Bei der Addition oder Subtraktion von zwei 16-Bit Integern kann das Ergebnis einem 17-Bit Integer zugewiesen werden. Im MSB ist dann der Übertrag gespeichert. Bei der Multiplikation erzeugen zwei Operanden der Größe n und m ein Ergebnis von $n + m$-Bit. Eine Ausnahme ist hier die Linksschiebeoperation. Hier richtet sich die Ergebnisbreite immer nach der Breite des ersten Operanden. Wenn der erste Operand eine Zahl ist, so muss dieser auf die entsprechende Breite gecastet werden. Da Zahlen lediglich die minimal benötigte Breite besitzen, würden sonst Stellen verloren gehen. Bei der Operation

```
1 x = 1<<y;
```

hat der erste Operand und damit das Ergebnis eine Breite von einem Bit, weshalb eine Cast-Operation benötigt wird:

```
1 x = (int)1<<y;
```

Globale Variablen und Felder

Die Deklaration von globalen Variablen und Feldern ist möglich. Im Prinzip widerspricht sich dies mit den Vorgaben des TransC-Prozessmodells, dass alle Effekte von Prozessen in der Deklaration erkennbar sein müssen. Aus diesem Grund werden die Variablen und Felder automatisch beim Kompilieren mit in die Argumentenliste des Prozesses und aller Superprozesse mit aufgenommen und die Deklaration der Variable letztendlich in den obersten Prozess verschoben, in dem sie benutzt wird.

4.1.2 Instanzen

Wenn ein Prozess mehrmals aufgerufen wird, gibt es für die Ausführung verschiedene Möglichkeiten.

```
1 int run(...) {
2     ...
3     a = calc(x,y);
4     b = calc(l,m);
5     ...
6 }
```

Es können entweder zwei Prozess-Instanzen erzeugt und beide Aufrufe parallel abgearbeitet werden oder nur eine Instanz verwendet werden, was zu einem sequentiellen Ablauf führt. Man kann Aufrufen explizit Instanzen zuweisen mit Hilfe des @-Operators, dem ein Name für die Instanz folgt. Für Instanzen gelten die gleichen Namenskonventionen wie für Variablen. Das nächste Beispiel erzwingt die Wiederverwendung derselben Instanz, wodurch die Berechnung von a und b nicht mehr parallel ablaufen kann.

```
1 int run(...) {
2     ...
3     a = calc(x,y)@i1;
4     b = calc(l,m)@i1;
5     ...
6 }
```

Umgekehrt kann man auch unterschiedliche Instanzen erzwingen, indem man unterschiedliche Bezeichner wählt.

```
1 int run(...) {
2     ...
3     a = calc(x,y)@i1;
4     b = calc(l,m)@i2;
5     ...
6 }
```

Ohne die Bezeichner ist die Zuordnung der Aufrufe auf die jeweilige Instanz nicht fest vorgegeben, so dass der Compiler diese Entscheidung übernimmt. Dies geschieht nach Durchführung aller Optimierungen und Transformationen vor der Codegenerierung. Dabei werden zwei Aufrufen unterschiedlichen Instanzen nur zugeordnet, wenn diese im selben Grundblock stehen.

Ein Prozess kann noch weiterlaufen, selbst wenn die aufrufende Funktion schon beendet ist. Mit Hilfe der `sync`-Funktion wird allerdings gewartet, bis der gekennzeichnete Prozess beendet ist.

```
1 int run(...) {
2     ...
3     calc(x,y)@i1;
4     ...
5     sync i1;
6 }
```

Wenn ein Prozess keine Rückgabewerte hat, kann er direkt mit dem Schlüsselwort `sync` gestartet werden. In diesem Fall wartet die aufrufende Funktion, bis der Prozess beendet ist.

4.1.3 sync-Anweisung

Neben der Synchronisierung von Prozessinstanzen werden mit `sync` Anweisungsfolgen voneinander getrennt. Sequentielle Anweisungen, die untereinander keine Datenabhängigkeiten aufweisen, können je nach Art der Ablaufplanung parallel ausgeführt werden. So benötigen die beiden Operationen

```
1 //ggf. parallele Ausführung der Addition und Multiplikation
2 a=b+c;
3 x=y*z;
```

zur Berechnung nur einen Zeitschritt, wenn Multiplikation und Addition gleichzeitig ausgeführt werden. Durch die *sync*-Anweisung zwischen den Operationen wird eine serielle Ausführung erzwungen, wodurch die Berechnung zwei Zeitschritte dauert, da die Operationen hinter dem *sync* erst ausgeführt werden, wenn die vorigen Operationen abgeschlossen sind.

```
1 //sequentielle Ausführung der Addition und Multiplikation
2 a=b+c;
3 sync;
4 x=y*z;
```

Dies kann z.B. bei asynchronen Streams ausgenutzt werden. Wird in einen asynchronen Stream zweimal hintereinander geschrieben, so hat der erste Schreibvorgang keine Auswirkungen, da der Stream sofort danach den Wert vom zweiten Schreibzugriff annimmt. Durch die Trennung der beiden Schreiboperationen durch ein *sync* liegt der erste Wert einen Zeitschritt an, bevor er durch den zweiten ersetzt wird.

4.1.4 Zeitverhalten

Es gibt eine vordefinierte Funktion `time`, mit deren Hilfe Echtzeitverhalten implementiert werden kann. `time` zählt die Anzahl der verstrichenen Takte, dementsprechend kann sie entweder ausgelesen oder zurückgesetzt werden, was mit Hilfe von Parametern bestimmt wird. Intern wird der Wert in einem 25-Bit Register gespeichert, wodurch bis etwa 33 Mio. gezählt werden kann. Mit Hilfe des @-Operators kann man beliebig viele Instanzen verwenden, die unabhängig voneinander Takte zählen und auch unabhängig voneinander zurückgesetzt werden können. Das folgende Beispiel zeigt eine Wartefunktion.

```
1  const int clockMHz = 25;
2  const int timeRST = 1;
3  const int timeRD = 0;
4
5  void delay(int ms) {
6     time(timeRST);
7     ms = clockMHz * 1000 * ms;
8     do {
9        sync;
10    } while(time(timeRD)<ms);
11 }
```

Der Aufruf `time(timeRST)` setzt die Zeit auf 0 zurück, während `time(timeRD)` den Zähler ausliest. Das `sync` in der `do...while`-Schleife kann auch weggelassen werden, hiermit verhindert man aber, dass die Schleife auch in zukünftigen Compilerversionen nicht wegoptimiert wird.

4.2 Vergleich mit anderen Sprachen

In den folgenden Abschnitten wird TransC mit drei Sprachen verglichen, die verschiedene Einsatzgebiete abdecken: Impulse C, Handel-C und Go. Während Handel-C und das ANSI-C kompatible Impulse C wie TransC zur Generierung von Hardware entwickelt wurden, ist Go eine Sprache zur Systemprogrammierung auf Mikroprozessoren, die allerdings nativ ein Prozessmodell besitzt und stream-ähnliche Konstrukte zur Interprozesskommunikation bereitstellt. Im Gegensatz zu Impulse C ist Handel-C lediglich an C angelehnt, setzt allerdings auf einer niedrigeren Sprachebene auf, da Parallelität und das Scheduling explizit angegeben werden müssen.

4.2.1 Impulse C

Impulse C [53] stammt von der Firma Impulse Accelerated Technologies und stellt eine Untermenge der C-Sprache dar, die zusätzlich eine Bibliothek mit C-kompatiblen Datentypen und API-Funktionen anbietet. Die Sprache dient zur Entwicklung von Applikationen für FPGAs. Im Vergleich zu TransC besitzt Impulse C keine erweiterte Syntax und ist somit kompatibel zum ANSI-C Standard, so dass die Programme mit Softwarecom-

pilern übersetzt und auf Mikroprozessoren ausgeführt und getestet werden können. Mit speziellen Impulse C Compilern werden Programme in Hardwarebeschreibungssprachen auf Registertransferebene synthetisiert. Die Impulse C Bibliothek unterstützt die parallele Programmierung durch Konstrukte zur Prozesserzeugung oder zur Interprozesskommunikation, die sich am CSP-Modell (siehe 2.5.2) orientiert. Außerdem können Speicherblöcke, Register und Signale explizit generiert werden. Zur Erzeugung von Prozessen wird der `co_process_create`-Aufruf verwendet, Streams werden mit Hilfe von `co_stream_create` erschaffen:

```
1 void config_helloworld(void *arg) {
2    co_stream s1,s2;
3    co_process producer, consumer;
4
5    s1=co_stream_create("Stream1",CHAR_TYPE,5);
6    s2=co_stream_create("Stream2",UINT_TYPE(32),5);
7    producer=co_process_create("Producer",(co_function) Producer, 2, s1, 1);
8    consumer=co_process_create("Consumer",(co_function) Consumer, 1, s2);
9 }
```

Die Streams erhalten einen Namen sowie einen Datentyp und die Größe des Zwischenpuffers. Benutzt ein Prozess diese zur Kommunikation, so werden sie dem Prozess bei der Erzeugung neben weiteren Parametern übergeben.

Die Streamkommunikation ist synchron und kann über einen Zwischenspeicher gepuffert werden. Da die Syntax ANSI-C kompatibel ist, müssen die Streams in spezielle Funktionsaufrufe eingebunden sein. Das folgende Beispiel zeigt einen Prozess, der einen Datenblock von einem Stream liest und in einen zweiten Stream schreibt:

```
1  void inc_proc(co_stream s_in, co_stream s_out) {
2     int i,buf;
3     do {
4        co_stream_open(s_in, O_RDONLY, INT_TYPE(32));
5        co_stream_open(s_out, O_WRONLY, INT_TYPE(32));
6        for(i=0; i<5; ++i) {
7           co_stream_read(s_in, &buf, sizeof(int));
8           co_stream_write(s_out, &buf, sizeof(int));
9        }
10       co_stream_close(s_in);
11       co_stream_close(s_out);
12    } while(1);
13 }
```

Man kann erkennen, dass der Stream für die Zugriffe ähnlich wie eine Datei geöffnet und geschlossen werden muss, da er während dieser Zeit für andere Prozesse gesperrt ist. Hier ist erst beim Öffnen der Datentyp sowie die Kommunikationsrichtung anzugeben und nicht schon bei der Deklaration.

Da die Streams synchron sind, eignen sie sich auch zur Prozesssynchronisation. Die Funktionalität der asynchronen Streams in TransC erreicht man mit Hilfe von gemein-

samen Variablen, die sich mehrere Prozesse teilen. Neben synchronen Streams kann die Interprozesskommunikation ebenfalls über einen gemeinsamen Speicher geschehen. Die dadurch auftretenden kritischen Bereiche werden durch Semaphore geschützt, die in der Impulse C Bibliothek vorhanden sind. Dafür stehen die Funktionen `co_semaphore_create`, `co_semaphore_release` und `co_semaphore_wait` zur Verfügung. Das Erzeugen von Speicher sowie das Kopieren von globalen Speicherbereichen in lokale Arrays einzelner Prozesse wird ebenfalls nativ mit Hilfe von bestimmten Funktionen realisiert.

Ein weiteres Mittel zur Prozesssynchronisation in Impulse C stellen Signale dar. Signale sind Datentypen, die aufgerufenen Prozessen als Argument übergeben werden und über die der Prozess anderen Prozessen bestimmte Zustände signalisiert. Die anderen Prozesse können dieses Signal abfragen und z.B. warten, bis es einen bestimmten Zustand erreicht. Im folgenden Beispiel fragt der Prozess `calc` das Signal `start` ab, bevor er einen kritischen Bereich betritt. Das Verlassen dieses Bereichs signalisiert er mit Hilfe von `done`.

```
1  void calc(co_signal start, co_signal done, ...) {
2     double A[ARRAYSIZE]; double B[ARRAYSIZE];
3     int32 status;
4     int32 offset = 0;
5
6     do {
7        co_signal_wait(start, (int32*)&status);
8        //kritischer Bereich
9        ....
10       co_signal_post(done, 0);
11    } while(1);
12 }
```

Alternativ kann dies auch wie in anderen Sprachen mit Hilfe von Streams bzw. Kommunikationskanälen geschehen. Allerdings ist bei den hier verwendeten Signalen der Programmieraufwand und der Overhead geringer.

4.2.2 Handel-C

Handel-C[40] ist eine Hochsprache, die zur Generierung von Hardware benutzt wird. Wie auch TransC ist sie zwar an C angelehnt, besitzt aber Erweiterungen, die nicht zum ANSI-Standard kompatibel sind. Dazu gehören u.a. die Definition von nebenläufigen Konstrukten oder die Möglichkeit, die Instantiierung von Hardwarekomponenten zu steuern. Handel-C orientiert sich ebenfalls am statischen CSP-Prozessmodell, allerdings wird die Parallelität nicht mit Prozessen, sondern über parallel laufende Kontrollzweige erreicht. Mit Hilfe des `par`-Operators spaltet sich der Kontrollfluss auf und die einzelnen Zweige werden simultan ausgeführt, wobei die Zweige wiederum sequentielle Konstrukte, Funktionsaufrufe usw. enthalten können. Abbildung 4.2.1 zeigt ein Codebeispiel und den zugehörigen Kontrollflussgraphen. Die in geschweiften Klammern eingefassten Blöcke werden parallel ausgeführt, während die darin enthaltenen Instruktionen wieder sequentiell sind.

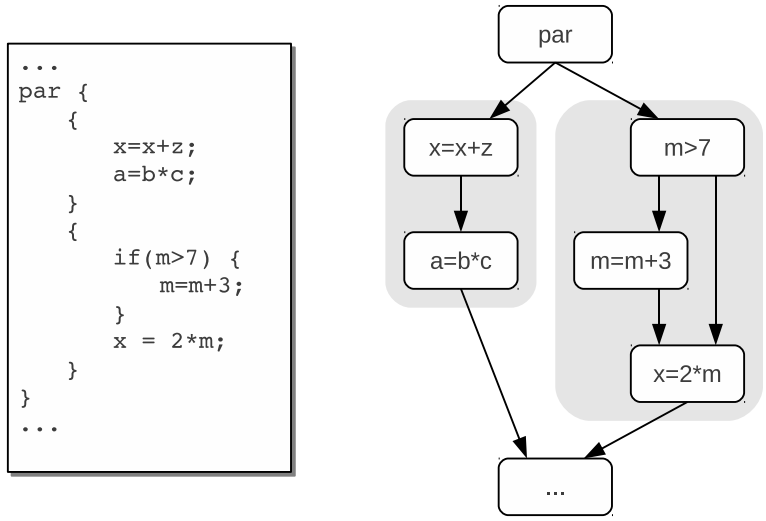


Abbildung 4.2.1: Parallele Kontrollflüsse in Handel-C

Auf diese Weise kann Parallelität sehr feingranular auf Instruktionsebene dargestellt werden, allerdings werden nicht explizit parallelisierte Anweisungen immer sequentiell ausgeführt. In TransC entscheidet der Scheduler anhand der Datenabhängigkeiten und gegebenen Randbedingungen, welche Instruktionen parallel ausgeführt werden können.

Die Kommunikation zwischen parallelen Anweisungen geschieht über *Channels*. Das Schreiben bzw. Lesen von Channels erfolgt synchron über spezielle Anweisungen, die durch "!"- bzw. "?"-Operatoren gekennzeichnet sind. Im folgenden Beispiel wird ein Channel `chin` gelesen, das Ergebnis in der Variablen `a` zwischengespeichert und an einen Channel `chout` ausgegeben:

```
1 chin ? a;
2 chout ! a
```

Zwischen Sender und Empfänger können keine Zwischenspeicher instantiiert werden, so dass der Datenaustausch simultan erfolgen muss. Mit Hilfe der Channels können parallele Anweisungsblöcke somit auch synchronisiert werden. In einen Channel dürfen mehrere Werte nicht gleichzeitig gelesen bzw. geschrieben werden, was z.B. durch folgendes Codesegment möglich wäre:

```
1 par {
2     chin ? a;
3     chin ? b; //nicht erlaubt
4 }
```

Datentypen können wie in TransC bitgenau definiert werden, indem die Bitbreite hinter dem Typen angegeben wird. Im Vergleich zu TransC kann diese undefiniert sein, so dass der Compiler anhand der Anweisungen die optimale Bitbreite ermittelt. Zudem können in Variablen einzelne Bits direkt manipuliert oder konkateniert werden, was in TransC nicht möglich ist:

```
1 int 17 a, b, c;
```

```
2 int undefined x, y;
3 ...
4
5 if(a[4])
6    ...
```

Die `if`-Bedingung greift auf das fünfte Bit der Variable `a` zu.

In TransC gibt es Optimierungen, die die Breite von Variablen, die innerhalb von Funktionen definiert sind, reduzieren. Fließkommadatentypen werden ebenfalls nicht nativ unterstützt, können aber durch entsprechende Bibliotheken implementiert werden. Wie in TransC gibt es mehrdimensionale Arrays, auf Pointer wurde ebenfalls verzichtet, da dies zu ineffizienterer Hardware führt.

Handel-C besitzt eine `delay`-Anweisung, die keine Auswirkungen hat, außer die nachfolgenden Instruktionen um einen Taktzyklus zu verzögern. Dies wird verwendet, um durch explizite Sequentialisierung Ressourcenkonflikte bei Speicher- oder Channelzugriffen zu vermeiden oder um ein bestimmtes Zeitverhalten zu erzielen. In TransC wird für die entsprechende Funktionalität der `sync`-Operator verwendet.

Im Zeitmodell von Handel-C dauert die `wait`-Anweisung und die Variablenzuweisung einen Taktzyklus, alle sonstigen arithmetischen Operationen, auch komplexere Ausdrücke, werden im selben Kontrollschritt wie die Zuweisung ausgeführt. Somit hat der Programmierer direkten Einfluss auf das Timing. Durch das Aufsplitten von Berechnungen durch mehrere Variablenzuweisungen wird die Zahl der benötigten Takte erhöht, dafür steigt aufgrund der geringeren Schaltungstiefe zwischen Registern ggf. die Taktfrequenz. In TransC hingegen bestimmt wie bei der Parallelisierung von Instruktionen lediglich der Scheduler die Kontrollschritte. Durch verkettete Zuweisungen wird die Ausführungszeit nicht erhöht.

4.2.3 Go

Go[1] ist eine nebenläufige Programmiersprache, die von der Firma Google zur Systemprogrammierung entwickelt wurde und somit auch die Möglichkeit zur hardwarenahen Beschreibung auf Mikroprozessoren bereitstellt. Wie die Programmiersprache C, die sich ebenfalls zur Systemprogrammierung eignet, ist auch Go imperativ und orientiert sich teilweise an deren Syntax. Dennoch wird auf fehleranfällige Sprachfeatures von C verzichtet. Durch die nebenläufigen Konstrukte ist die Sprache zur Programmierung von verteilten Systemen wie Mehrkernprozessoren oder Rechnerverbünden geeignet.

Im Vergleich zu Impulse C und Handel-C dient Go nicht zur Synthese von Hardware. Da aber das Hauptaugenmerk von TransC ebenfalls auf der Beschreibung von nebenläufigen Prozessen liegt, wird Go zum Vergleich herangezogen, dabei werden die verschiedenen Sprachfeatures zur Implementierung von Prozessen und der Interprozesskommunikation analysiert.

Im Gegensatz zu bisherigen nebenläufigen Programmiersprachen oder Threadbibliotheken unterstützt Go zusätzlich zu Konstrukten wie Mutexen, Bedingungsvariablen, o.ä. auch höhersprachliche Schnittstellen zur IPC und Prozesssynchronisation. Go orientiert

sich an dem in Abschnitt 2.5.2 beschriebenen CSP-Prozessmodell sowie an den Sprachen Occam[57] und Erlang[10]. Die Kommunikation zwischen Prozessen findet u.a. über sogenannte Channels statt. Die gesamte Prozess- und Kommunikationsstruktur ist dynamisch und wird erst zur Laufzeit festgelegt. In einigen Bereichen hat TransC aufgrund des Fehlens dieser dynamischen Eigenschaften Nachteile, dabei muss jedoch berücksichtigt werden, dass TransC zur effizienten Erzeugung von Hardware entwickelt wurde und nicht zur Programmierung von Systemen auf Mikroprozessoren.

Prozesse

In Go gibt es Prozesse und Funktionen, wobei die Prozesse *Goroutinen* genannt werden. Wie Prozesse in TransC laufen auch die Goroutinen parallel zu anderen Goroutinen und zum Aufrufer, während bei Funktionen immer auf das Ende gewartet wird. In Go gibt es bei der Definition keinen Unterschied zwischen Goroutinen und Funktionen, erst beim Aufruf entscheidet sich, ob die Routine parallel ausgeführt oder ob auf diese gewartet werden soll. Das folgende Programm zeigt die Definition von zwei Funktion und deren Aufruf als Goroutine:

```
1 //Go
2 func calc1(){ ... }
3 func calc2(){ ... }
4 func main() {
5    go calc1() //Aufruf als Goroutine, auf das Ende wird nicht gewartet
6    go calc2()
7 }
```

Ließe man das Schlüsselwort `go` beim Aufruf weg, so verhielten sich `calc1` und `calc2` wie normale Funktionen. Dies ist gegensätzlich zu TransC, wo als Prozess deklarierte Routinen mit dem Schlüsselwort `sync` auch als Funktionen aufgerufen werden können, wenn sie keine Rückgabewerte besitzen:

```
1 //TransC
2 process void calc1(){ ... }
3 process void calc2(){ ... }
4 void main() {
5    sync calc1(); //eigentlich Prozess, Aufruf aber als Funktion
6    sync calc2();
7 }
```

Die Goroutinen sind dynamisch und die einzelnen Instanzen sind nicht wie bei TransC schon beim Programmstart bekannt. Das Starten einer Routine z.B. in einer Schleife mit zehn Iterationen würde auch zehn Goroutinen erzeugen. In TransC würde immer wieder dieselbe Prozessinstanz aufgerufen werden.

Der Grund für die Festlegung bei der Definition für eine parallele oder serielle Ausführung in TransC liegt darin, dass bestimmte Operationen, wie z.B. eine aufwändige Fließkommamultiplikation oder -Addition, einfacher in den Code zu integrieren sind. Seien `mul`

und `add` Prozesse, von denen lediglich die Schnittstelle und die Latenz bekannt sind. Mit einem Aufruf wie "`a = add( mul(a,b), mul(c,d) )`" können die Multiplikationen bereits vom Compiler automatisch parallel bzw. überlappend gescheduled werden.

Interprozesskommunikation und Synchronisation

Neben globalen Speichern können in Go Channels zur Interprozesskommunikation verwendet werden. Ein Channel verhält sich wie ein synchroner Stream in TransC. Die Daten können ungepuffert übertragen werden, so dass das Senden und der Empfang gleichzeitig stattfinden müssen, die Angabe einer Puffergröße ist ebenfalls möglich. Allerdings können Channels nicht nur statisch, sondern auch dynamisch zur Laufzeit erzeugt werden. Im folgenden Beispiel wird eine einfache Kommunikation zwischen einer lesenden und einer schreibenden Goroutine gezeigt, die über einen Channel `ci` vom Typ `int` mit einer Puffergröße von 4 Daten austauschen:

```
1 //Go
2 ci := make(chan int, 4)
3 func send(){ ... ci<-3 ... }
4 func recv(){ ... a = <-ci ... }
5 void main() {
6    go send()
7    go recv()
8 }
```

Dabei erkennt man im Vergleich zu TransC die Kommunikation nicht an der Schnittstelle, da der Channel als global definierte Variable vorliegt. Die äquivalente Implementierung in TransC sieht wie folgt aus, der global definierte Channel `ci` liegt hier lokal in der aufrufenden Funktion:

```
1 //TransC
2 process void send(sostream<int> str){ ... str = 3; ... }
3 process void recv(sistream<int> str){ ... a = str; ... }
4 void main() {
5    snstream<int> ci[4];
6    send(ci);
7    recv(ci);
8 }
```

In TransC werden für die Streams keine globalen Variablen verwendet, weil man sonst nicht die Seiteneffekte des Prozesses bzw. seine Hardware-Schnittstelle nicht an der Deklaration erkennen kann. Allerdings ist es auch in Go möglich, Channels über Argumente an Goroutinen zu übergeben (`func filter(data chan int, prime int)`), wobei die Rolle des Senders und des Empfängers nicht so restriktiv gehandhabt wird. Neben der Übergabe an Funktionen können Channels auch über weitere Channels verschickt werden, was u.a die einfache Implementierung von parallelen Demultiplexern ermöglicht. Somit ist auch wie im pi-Kalkül die Kommunikationsstruktur dynamisch.

Synchronisation

Wie in TransC können auch in Go Channels zur Prozesssynchronisation verwendet werden. Da sie global definiert werden können, können sie auch als Mutex oder Semaphore eingesetzt werden:

```
1  //Go
2  var global int = 0
3  var m = make(chan int, 1)
4  func f1() { <-m;  global = 1; m <- 1 }
5  func f2() { <-m;  global = 2; m <- 1 }
6  func main() {
7     m <- 1
8     go f1()
9     go f2()
10 }
```

Im oben gezeigten Beispiel ist die Zuweisung an die Variable `global` ein kritischer Bereich, der erst betreten werden kann, wenn im Channel `m` ein Datum liegt. In TransC ist diese Implementierung nicht möglich. Zur Erzeugung vom Mutexen oder Semaphoren muss beispielsweise der Algorithmus von Peterson[86] eingesetzt werden, was wesentlich aufwändiger ist.

Kapitel 5

Zwischencode

Das folgende Kapitel beschreibt den Zwischencode, der nach Analyse der Eingabesprache entsteht und auf dem die Optimierungen durchgeführt werden. Zusätzlich zur generellen Codestruktur werden Algorithmen und Konstrukte erläutert, die zur Erkennung von Schleifen oder zur Realisierung von komplexeren Operationen dienen. Statt der Entwicklung eines spezifischen Zwischenformats hätten auch bestehende Bibliotheken verwendet und erweitert werden können. Zu den bekanntesten Beispielen gehören hier LibFirm[68], LLVM[67], gcc[52] oder SUIF2[101]. Die Eigenentwicklung wurde bevorzugt, da auf diese Weise spezielle Eigenschaften und Operationen, auch zur Unterstützung des in Kapitel 3 vorgestellten Prozessmodells, leichter integriert werden können und keine Einschränkungen u.a. bei der Darstellung der Interprozesskommunikation bestehen.

Das Zwischenformat soll möglichst einfach gehalten werden und nur aus einem CDFG mit Kontroll- und Datenflussknoten bestehen, wie in Abschnitt 2.2.4 beschrieben. Zwischenformate, die z.B. in [37] oder [36] benutzt werden, besitzen attributierte Kontroll-Datenfluss-Graphen, dessen Kontrollflusselemente eine Vielzahl von Eigenschaften aufweisen, um z.B. Schleifen, *case*- oder *if-else*-Anweisungen zu modellieren. Bei dem hier verwendeten Zwischenformat sollen solche Konstrukte lediglich aus der Knotenstruktur des Kontrollflusses erkannt werden, da sich auf diese Weise wegen der geringeren Anzahl von zu beachtenden Sonderfällen Analysen und Graphentransformationen vereinfachen.

5.1 CDFG

Jede Funktion der imperativen Eingabesprache wird durch einen spezifischen Kontroll-Datenflussgraphen (CDFG) dargestellt, ähnlich wie in 2.2.4 beschrieben:

$$CDFG = (CFG, D, E_{DD}).$$

CFG ist ein Kontrollflussgraph, der definiert ist als

$$CFG = (BB, T).$$

BB ist die Menge aller Grundblöcke, wobei $T \subseteq BB \times BB$ die Menge aller Kanten des Kontrollflusses darstellt. Eine Kante steht für eine Transition $t = (bb_i, bb_j) \in T$, die die beiden Blöcke bb_i und bb_j mit $0 \leq i, j < |BB|$ verbindet, wobei bb_i vor bb_j ausgeführt wird. bb kann entweder ein Startblock $bb_{Start} = bb_0$, ein Endblock bb_{End} oder ein anderer Block sein. Ein CFG enthält exakt einen Start-, einen Endblock und eine beliebige Zahl von weiteren Blöcken. Es gibt mindestens einen Pfad vom Startblock zu jedem anderen Block, aber aufgrund von Zyklen im CFG nicht unbedingt einen Pfad von jedem beliebigen Block zum Endblock. Um den Kontrollfluss einfach zu halten, wurde die Zahl von möglichen Nachfolgern eines Blocks auf höchstens zwei beschränkt.

Jedem Block in einem Kontrollfluss wird ein Ordnungsindex (ID) zugewiesen, an dem sich der relative Ausführungszeitpunkt erkennen lässt. Da der Startblock immer zuerst ausgeführt wird, erhält er die Nummer 0 ($ID(bb_{Start}) = 0$). Die übrigen Blöcke werden mit Hilfe einer Breitensuche durchnummeriert, sodass die Nachfolger eines Blockes einen größeren Index erhalten. Durch Zyklen im Graphen können Nachfolger bereits einen Index besitzen, in diesem Fall wird ihnen kein neuer zugeteilt. Auf diese Weise erhalten später ausgeführte Blöcke immer eine höhere ID, eine Ausnahme bilden hierbei Schleifen.

Grundblöcke werden mit der Eigenschaft *sync* markiert, wenn der Kontrollfluss im Programm durch einen `sync`-Befehl oder einen Funktionsaufruf unterbrochen wird. Diese Markierung verhindert bei Optimierungen und Zwischencodetransformationen die Verschmelzung des Blocks mit dem Nachfolger, wodurch die Befehlsreihenfolge im Zwischencode erhalten bleibt. Werden z.B. zwei Funktionen hintereinander aufgerufen, so sind diese in zwei separaten und mit *sync* markierten Blöcken gespeichert.

D ist eine Menge von Datenflussgraphen DFG:

$$D = \{d^0, d^1, ..., d^{|D|-1}\}.$$

Jeder Block bb_i ist genau einem DFG d^i zugeordnet, so dass $|BB| = |D|$. Ein DFG ist ein gerichteter azyklischer Graph, der aus Knoten N und Kanten E besteht:

$$DFG = (N, E).$$

Jeder Knoten $n \in N$ entspricht einer Operation, wobei Knoten durch Kanten $e \in E \subseteq N \times N$ miteinander verbunden sind, die die Datenabhängigkeiten repräsentieren. Die Knoten in einem DFG müssen nicht unbedingt zusammenhängend sein.

Die Menge der Nachfolger eines Datenflussknotens n bzw. eines Kontrollflussknotens bb wird mit $Next(n)$ oder $Next(bb)$ bezeichnet, dementsprechend trägt die Menge der Vorgänger den Namen $Prev(n)$ bzw. $Prev(bb)$. Es gibt Knoten mit bestimmten Eigenschaften, die mit entsprechenden Kennungen versehen sind, wie zum Beispiel Bedingungsknoten n_{cond}. Wenn ein Block zwei Nachfolger hat, bestimmt der Boolesche Wert von n_{cond} zur Laufzeit, welcher Nachfolger genommen werden soll. Zudem existieren spezifische Eingabeknoten $n_{in} \in N_{in}$ zum Lesen von Eingabewerten und Ausgabeknoten $n_{out} \in N_{out}$ zum Schreiben von Ausgabewerten. Eingabeknoten erhalten ihren Operanden von Ausgabekno-

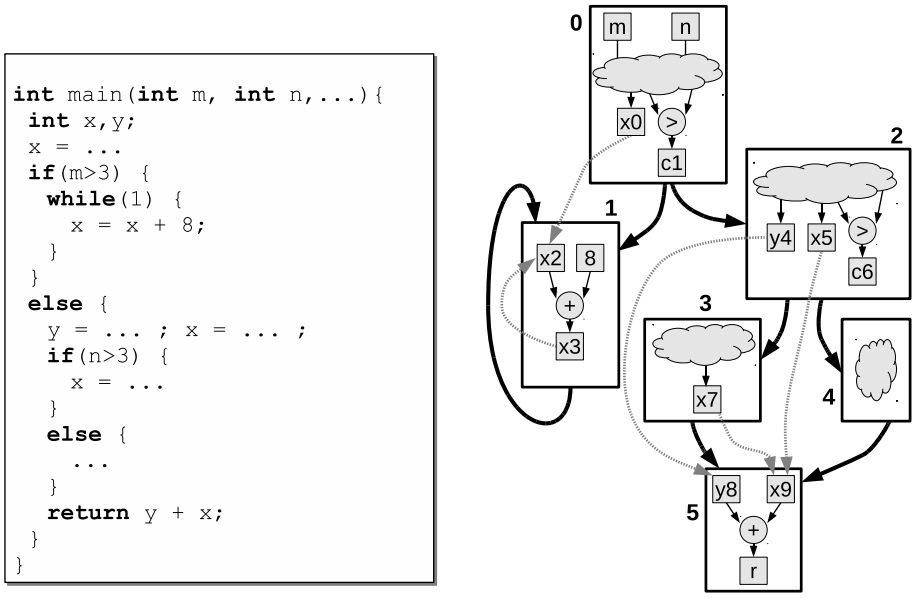


Abbildung 5.1.1: C-Code und zugehöriger CDFG

ten aus Datenflussgraphen, die Bestandteil von zuvor ausgeführten Blöcken sind, so dass Eingabeknoten keine direkten Vorgänger innerhalb ihres Grundblocks besitzen. Die Ergebnisse von Ausgabeknoten können auch im selben DFG referenziert werden, so dass direkte Nachfolger von Ausgabeknoten möglich sind. Knoten, die die Endresultate eines CDFGs speichern, werden als Return-Knoten bezeichnet und sind in einer Menge N_{ret} enthalten. Knoten, die Funktionsargumente übergeben bekommen, sind Bestandteil der Menge N_{arg}. Wenn Knoten eine konstante Zahl zurückgeben, sind sie in der Menge N_{num}. Um Datenflussanalysen zu vereinfachen, darf nur der Startblock bb_0 Argumentknoten enthalten. Der Endblock bb_{End} eines CDFGs hat immer einen leeren DFG. Abbildung 5.1.1 zeigt einen Programmausschnitt mit dem zugehörigen Datenflussgraphen. Die Knoten m und n sind Argumentknoten, r ist ein Return-Knoten, während $c1$ und $c6$ Sprungbedingungen darstellen.

E_{DD} enthält Kanten, die Ausgabe- mit Eingabeknoten N^i_{out} und N^j_{in} von zwei DFGs d^i und d^j miteinander verbinden.

$$E_{DD} \subseteq N^i_{out} \times N^j_{in}.$$

Im Vergleich zu der Kante $e^i \in E^i$, die eine Datenabhängigkeit innerhalb eines Blockes bb_i repräsentiert, steht eine Kante

$$e^{ij}_{DD} = (n^i_{out}, n^j_{in}) \in E_{DD}$$

für eine Datenabhängigkeit zwischen Grundblock bb_i und bb_j.

Diese Kanten resultieren von Datenflussanalysen, wie später in Abschnitt 5.3.6 beschrieben wird und sind als gepunktete graue Linien in Abbildung 5.1.1 dargestellt. Durch Zyklen im Kontrollfluss können Kanten entstehen, die entgegengesetzt des Datenflusses

verlaufen (siehe Abbildung 5.1.1, bb_1). Verzweigungen im Kontrollfluss führen zu Ausgabeknoten mit mehreren E_{DD}-Kanten, wenn sie von mehreren Knoten innerhalb der Zweige referenziert werden. Analog dazu können Eingabeknoten hinter Vereinigungen auch Ergebnisse aus mehreren Zweigen referenzieren, was zu mehreren eingehenden E_{DD}-Kanten führt, wie in Abb. 5.1.1, bb_5 gezeigt. Um die Menge aller Knoten darzustellen, die mit einem Knoten n durch E_{DD}-Kanten verbunden sind, wird die Notation $N_{DD}(n)$ benutzt.

5.2 Schleifen

Wie im vorigen Abschnitt beschrieben, erhalten Grundblöcke einen Index, wobei der von später ausgeführten Blöcken höher ist. Für eine Transition (bb_m, bb_n) ist somit $m < n$. Dies gilt nicht für Schleifen im Kontrollfluss aufgrund ihrer Rückwärtskanten $t = (bb_i, bb_j)$, $i \geq j$, wie man z.B. an Kante (bb_1, bb_1) in Abb. 5.1.1 erkennen kann. Transitionen mit dieser Eigenschaft werden als Schleifentupel bezeichnet und sind in der Menge $L \subseteq T$ enthalten. Da eine Schleife eindeutig durch solch eine Rückwärtskante charakterisiert ist, wird bb_i auch als der Endblock bb_{LEnd} und bb_j als der Startblock bb_{LStart} der Schleife bezeichnet. Eine Schleife l wird somit durch die beiden Blöcke $bb_{LStart,l}$ und $bb_{LEnd,l}$ eingegrenzt. Die Menge BB_l enthält alle Grundblöcke, die zur Schleife l gehören, d.h., deren Index größer gleich $ID(bb_{Start,l})$, aber kleiner gleich $ID(bb_{End,l})$ ist:

$$BB_l = \{bb_i\}, i >= ID(bb_{LStart}) \wedge i <= ID(bb_{LEnd}).$$

Normalerweise dominiert bb_{LStart} alle Blöcke in BB_l. Ist dies nicht der Fall, so besitzt die Schleife mehrere Eintrittspunkte, was durch strukturierte Programmierung [23] allerdings nicht vorkommt. Durch zusätzliche `break`- oder `continue`-Anweisungen im Schleifenrumpf kann es auch mehrere Austrittspunkte geben, also Blöcke ungleich bb_{LEnd}, dessen Nachfolger nicht mehr in BB_l enthalten sind. Besitzt eine Schleife keinen Austrittspunkt, so handelt es sich um eine Endlosschleife. Einfache Endlosschleifen lassen sich daran erkennen, dass bb_{LEnd} nur einen Nachfolger bb_{LStart} besitzt. Wenn bb_{LEnd} zwei Nachfolger besitzt, wobei einer bb_{LStart} und der andere einen Block mit einem höheren Index als bb_{LEnd} referenziert, so wird die Schleife als nicht endlos angenommen, da sie theoretisch über diese Vorwärtskante verlassen werden kann. Wenn beide Nachfolger Rückwärtskanten sind, d.h. der Index ist kleiner gleich dem Index von bb_{LEnd}, so ist der Nachfolger mit dem kleineren Index der Startblock der Endlosschleife. In diesem Fall handelt es sich um zwei verschachtelte Schleifen, wobei die äußere nicht verlassen werden kann und somit endlos ist. Alle Endlosschleifen eines CDFGs sind Element der Menge $L_{Inf} \subseteq L$.[1]

Allgemein ist eine Transition $t = (bb_m, bb_n)$ Element von L_{Inf}, wenn

1. alle Nachfolger von bb_m einen Index kleiner gleich m haben und
2. bb_n den kleinsten Index aller Nachfolger besitzt.

[1] Eine weitere Voraussetzung für Endlosschleifen ist, dass es neben bb_{LEnd} keine weiteren Austrittspunkte in BB_l geben darf.

5.3 Erstellung von CDFGs

Die nächsten Abschnitte beschreiben die Vorgänge, um aus einem Syntaxbaum einen Kontroll-Datenflussgraphen aufzubauen, wie in 5.1 beschrieben. Dabei stehen vor allem die Eingabe- und Ausgabeknoten im Mittelpunkt und wie sie aus den Anweisungen im Quellcode generiert werden. Obwohl sich die Datenflussknoten stark an den Variablen im Quellcode orientieren und teilweise identische Namen tragen, werden die Operationen nicht auf Variablen, sondern auf Datenflussknoten durchgeführt. Die Variablen dienen als Hilfskonstrukte, um passende Datenflussknoten miteinander zu verbinden.

Zunächst werden einige Techniken zur Verschmelzung und zum Vergleich von Datenflussgraphen erläutert, die für die Generierung des CDFGs notwendig sind. Danach folgt die Beschreibung vom Aufbau des Kontrollpfades, des Datenpfades sowie der blocküber-greifenden Datenabhängigkeiten.

5.3.1 Datenflussgraphen

Alle Bestandteile der Anweisungen in der Eingangssprache werden direkt in Datenflussknoten umgewandelt. Dabei werden auch Schreib- und Lesezugriffe auf Variablen berücksichtigt. Abbildung 5.3.1 zeigt die Zuweisung '`a=b`', aus der ein DFG mit zwei Knoten generiert wird. Der obere Knoten stellt die Leseoperation der Variablen dar und der untere die Schreiboperation, was durch das "="-Symbol vor der Variable gekennzeichnet wird. Diese Darstellung mit Hilfe der ursprünglichen Variablen bringt Vorteile, auch wenn durch die Darstellung des gesamten Datenflusses mit Knoten und Kanten die Variablen nicht mehr benötigt werden. Zum einen erhöht sich die Lesbarkeit, da die Zuordnung zum ursprünglichen Quellcode nicht verloren geht. Außerdem behält man auf einfache Weise die vom Programmierer vorgegebenen Datentypen und Typumwandlungen. Daher werden diese Schreibknoten auch für Variablen erzeugt, die nur für die temporäre Zwischenspeicherung von Ergebnissen dienen. Wenn `a` einen anderen Typ als `b` hat, stellt der Schreibknoten auch gleichzeitig eine Castoperation dar. Auf die Erstellung eines separaten Knotens mit einer Castoperation wird hier verzichtet, da die Castoperation bis auf die fehlende Angabe des Variablennamens identisch zu einer Variablenzuweisung ist.

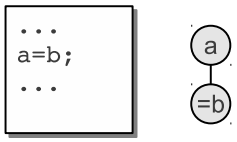


Abbildung 5.3.1: DFG einer einfachen Variablenzuweisung

5.3.2 Verschmelzung von Datenflussgraphen

Wenn zum Beispiel ein Grundblock lediglich die Operation '`a=2*b`' enthält und diese in einen DFG umgewandelt wird, wie in Abb. 5.3.2a links oben gezeigt, so handelt es sich bei

dem Knoten `b` um einen Eingabeknoten. `b` holt seine Operanden aus anderen Datenflussgraphen, da er im selben DFG keinen Vorgänger besitzt. `a` ist ein Ausgabeknoten, da dieser von Eingabeknoten aus anderen DFGs referenziert werden kann. Die Multiplikation wird durch einen eigenen Knoten dargestellt, der mit den Operandenknoten und der Zuweisung verbunden ist.

Die Anweisungsfolge '`a=2*b; c=a;`' kann in Form von zwei sequentiell ausgeführten Datenflussgraphen betrachtet werden, die wie in Abb. 5.3.2a gezeigt, zu einem kombiniert werden können. Dafür müssen die Ausgabeknoten des ersten DFG mit den Eingabeknoten des zweiten DFG verschmolzen werden, wobei alle Ausgabeknoten erhalten bleiben. Wenn die Ausgabeknoten von zwei zu kombinierenden DFGs dieselbe Variable referenzieren, so fällt der Ausgabeknoten des ersten Datenflussgraphen weg (Abb. 5.3.2b). Für die Verschmelzung von zwei sequentiell ausgeführten DFGs d_0 und d_1 wird der "+"-Operator verwendet ($d_s = d_0 + d_1$).

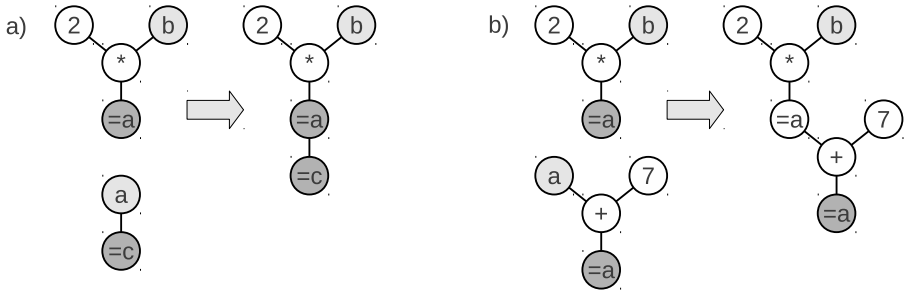


Abbildung 5.3.2: Verschmelzung von sequentiell ausgeführten DFGs. Die hellgrauen Datenflussknoten sind Eingabeknoten, die dunkelgrauen Ausgabeknoten

Neben der sequentiellen Verschmelzung gibt es auch die Verschmelzung von zwei parallel ausgeführten Datenflussgraphen d_0 und d_1, was als parallele Verschmelzung bezeichnet wird. Hier dürfen sich die Ausgabeknoten von d_0 und d_1 nicht auf dieselben Variablen beziehen, da Ausgabeknoten innerhalb eines DFGs eindeutig sein müssen. In diesem Fall werden nur die Eingabeknoten miteinander verschmolzen, wie in Abb. 5.3.3 gezeigt. Für die parallele Verschmelzung von zwei DFGs d_0 und d_1 wird der |-Operator verwendet ($d_p = d_0|d_1$).

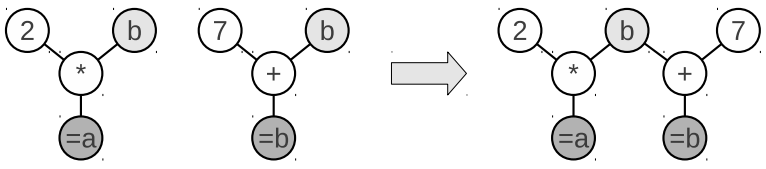


Abbildung 5.3.3: Verschmelzung von parallel ausgeführten DFGs. Die hellgrauen Datenflussknoten sind Eingabeknoten, die dunkelgrauen Ausgabeknoten

5.3.3 Gleichheit von DFG-Knoten

Wenn zwei DFG-Knoten gleich sind, führen sie die gleiche Operation aus und produzieren auch das gleiche Ergebnis, so dass einer der beiden Knoten eliminiert werden kann und dessen Nachfolger den erhaltenen Knoten referenzieren. Die Eliminierung von bestimmten gleichen Knoten ist nicht nur eine Optimierung, sondern für den korrekten Aufbau der Datenflussgraphen innerhalb des Grundblocks notwendig. Wenn der Teilgraph links unten in Abbildung 5.3.2a zwei verschiedene Eingabeknoten hätte, die von `a` lesen, so gäbe es bei der Verschmelzung mit dem darüber liegenden Teilgraphen Probleme, da der Ausgabeknoten `=a` mit beiden Knoten verbunden werden müsste. Zwei DFG-Knoten sind gleich, wenn

1. sie demselben Grundblock zugeordnet sind,

2. ihre Operation gleich ist,

3. ihre Vorgänger ebenfalls gleich sind.

Aus Effizienzgründen könnte man statt der Gleichheit der Vorgänger auch fordern, dass sie dieselben Vorgänger referenzieren müssen. So muss man den Graphen nicht jedes Mal rekursiv über alle Vorgänger durchlaufen, um gleiche Knoten zu erkennen. Wenn von zwei gleichen Knoten immer einer eliminiert wird, dann sind nach mehreren Anwendung der Gleichheitsprüfung in Verbindung mit den Eliminierungen gleiche Vorgänger eines Knotens auch identisch, wie in Abb. 5.3.4 gezeigt. Hier sind zwei Wiederholungen notwendig, um bei der Funktion '`c=(a+b)*(a+b)`' alle gleichen Knoten zu entfernen. Wenn zwei Knoten die gleichen Operationen ausführen und auch dieselben Operanden haben, diese Operationen aber Seiteneffekte aufweisen, dann dürfen die Knoten nicht gelöscht werden. In diesem Fall muss aber der Vergleichsoperator der Operation dafür sorgen, dass diese nicht als gleich betrachtet werden können, da sie andere Auswirkungen haben. Die Entfernung aller gleichen Knoten ist identisch mit der Eliminierung gemeinsamer Teilausdrücke. Die Gleichheit von Knoten innerhalb eines Blocks wird später u.a. genutzt, um auch global über Grundblöcke hinweg gemeinsame Teilausdrücke zu finden.

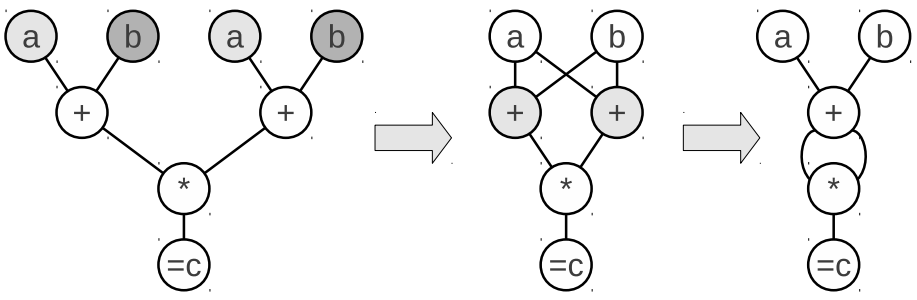


Abbildung 5.3.4: Eliminierung gleicher Knoten eines DFGs

5.3.4 Erstellung des CFGs

Kontrollkonstrukte wie Schleifen und Verzweigungen werden in einen Kontrollflussgraphen umgewandelt. Dabei gibt es, wie in 5.1 beschrieben, nur eine Knotenart für sämtliche Konstrukte im Kontrollfluss, so dass man nicht mehr direkt anhand eines Knotens auf die ursprüngliche Anweisung im Quellcode (z.B. `for`, `while`, `if`) schließen kann. Gemäß dem Prozessmodell erhält jede Funktion und jeder Prozess einen eigenen Kontrollflussgraphen. Bei der Erstellung des CFG gibt es immer einen aktuellen Grundblock, der als bb_A bezeichnet wird. Wenn der Syntaxbaum einer Funktionsdefinition eingelesen wird, wird bereits vor dem Parsen der ersten Anweisungen ein neuer Block erzeugt und als bb_A markiert. Alle dann eingelesenen Datenflussanweisungen werden in Form eines DFGs in diesem gespeichert. Nach dem Einlesen einer Kontrollanweisung wird ein neuer Block erzeugt und in bb_A als Nachfolger gesetzt. Anschließend wird der neue Block als bb_A markiert.

Beim Einlesen eines *if*-Konstrukts wird bb_A auf einem Stack gespeichert. Nach der Verarbeitung der letzten Anweisung des *if*-Zweigs wird ein neuer Nachfolgeblock erzeugt und mit dem letzten Block im *if*-Zweig verknüpft. Anschließend erhält der oberste Block auf dem Stack den neuen Block als alternativen Nachfolger und wird vom Stack genommen. Dieses Stack wird ebenfalls bei *if-else*-Anweisungen oder Schleifen genutzt, so dass beliebig verschachtelte Kontrollanweisungen umgesetzt werden können. Auch bei der Umsetzung von Schleifen oder *case*-Anweisungen finden solche Stacks Verwendung.

5.3.5 Einfügen von DFGs in Grundblöcke

Ausgangspunkt bei der Erstellung des Datenflussgraphen ist immer der aktuelle Grundblock bb_A. Beim Einlesen von einer Datenflussanweisungen wird diese zunächst in einen Datenflussgraphen d_a umgewandelt und anschließend nach folgenden Schritten dem bereits im Block vorhandenen DFG d_b angefügt, der resultierende Graph im Block ist dann d_r:

1. Elimination gleicher Eingabeknoten in d_a nach 5.3.4
2. $d_r = d_b + d_a$
3. Elimination gleicher Eingabeknoten in d_r nach 5.3.4
4. Wenn d_a Funktionsaufrufe enthält, so werden die DFGs nicht miteinander verschmolzen, damit die Ausführungsreihenfolge erhalten bleibt. In diesem Fall wird nach dem Anfügen von d_a ein neuer Block bb_N erzeugt, der bb_A als Vorgänger hat. Anschließend wird bb_N als aktuell markiert, so dass darauf eingelesene Datenflussgraphen in diesem eingefügt werden.

Die Eliminierung gleicher Eingabeknoten hat den Sinn, dass für die Verschmelzung mit Ausgabeknoten nur ein Eingabeknoten betrachtet werden muss. Dies gilt auch für Datenflussanalysen und Optimierungen, die auf den DFGs durchgeführt werden. Abb. 5.3.5 zeigt ein Beispiel für den Aufbau eines DFGs im Grundblock aus einer Anweisungsfolge. Dabei ist `calc` ein nicht weiter spezifizierter Prozess. Die dunkelgrau markierten DFG-Knoten

sind Ausgabeknoten, die mit den entsprechenden Variablenleseknoten aus den einzufügenden DFGs verbunden werden.

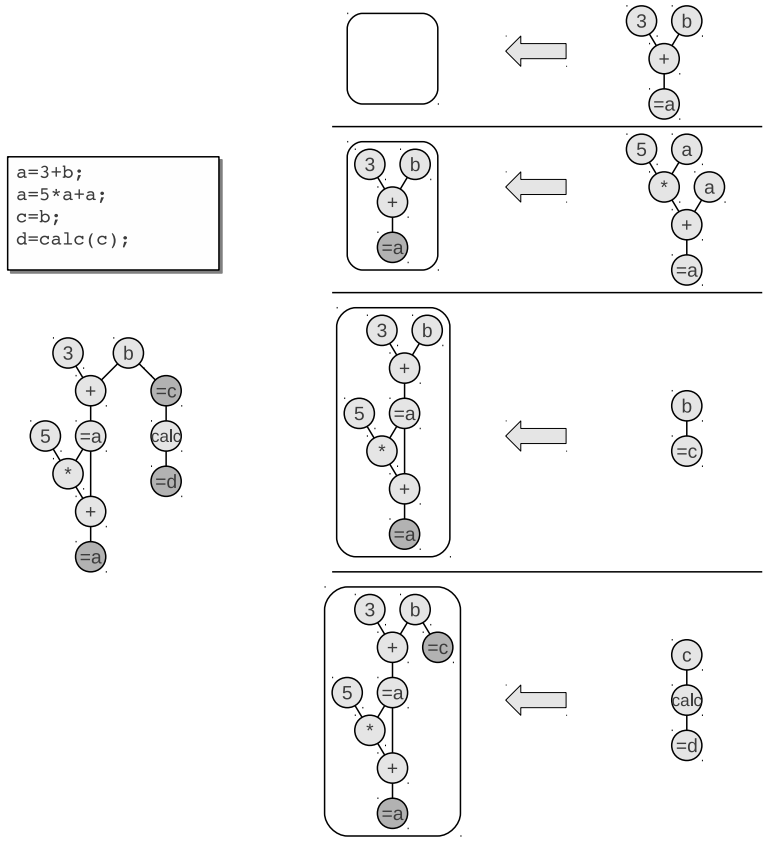


Abbildung 5.3.5: Anfügen von Anweisungen in den DFG eines Grundblocks. Die dunkelgrauen Knoten sind Ausgabeknoten.

Wäre `calc` eine Funktion, so würde der Datenflussgraph für den Aufruf noch am DFG des vorhandenen Blocks angefügt werden und der Block mit *sync* markiert. Alle anschließend geparsten Anweisungen kämen in einen neuen Block, der Nachfolger des vorhandenen Blocks wäre.

Nach der Erstellung des CDFGs sind die einzelnen DFGs der Grundblöcke noch nicht blockübergreifend miteinander verknüpft, was aber Voraussetzung für bestimmte Datenflussanalysen und Optimierungen ist. Dies wird mit einer Lebenszeitanalyse auf Grundblockebene erreicht.

5.3.6 Lebenszeitanalyse

Bei der Lebenszeitanalyse[79] (engl. *liveness analysis*) werden für jeden Programmpunkt die lebendigen Variablen berechnet. Diese Informationen können z.B. für Optimierungen oder für die Registervergabe bei der Softwaresynthese verwendet werden. Eine Variable ist lebendig, wenn ihr Wert zu einem späteren Zeitpunkt benutzt wird. Dazu wird ein Programm nach seinen Anweisungen in einen Kontrollflussgraphen nach 2.2.2 aufgeteilt und für jede Anweisung n die Menge der gelesenen Variablen $use(n)$, der geschriebenen

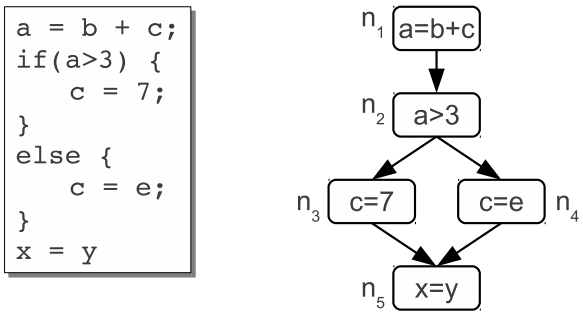


Abbildung 5.3.6: Kontrollflussgraph für Lebenszeit-Analyse

Variablen $def(n)$ sowie die Menge der Folgeanweisungen $Next(n)$ bestimmt. Abbildung 5.3.6 zeigt eine Anweisungsfolge mit dem zugehörigen Kontrollflussgraphen. $use(n_1)$ sind beispielsweise die Variablen b, c und $def(n_1)$ ist a. Die Nachfolger $Next(n_2)$ sind die Knoten n_3, n_4, da hier eine Verzweigung auftritt. Das Ziel der Lebenszeitanalyse ist die Bestimmung der Mengen $in(n)$ und $out(n)$, die die lebendigen Variablen vor und nach Ausführung von Knoten n enthalten. Eine Variable i ist Element von $in(n)$, wenn $i \in use(n)$ oder wenn es einen Pfad von n zu einem Knoten gibt, der i benutzt, i auf diesem Pfad aber nicht neu definiert wurde. Eine Variable i ist Element von $out(n)$, wenn $i \in in(s)$ für $s \in Next(n)$. Damit ergeben sich folgende Gleichungen:

- $in(n) = use(n) \cup (out[n] \setminus def[n])$
- $out(n) = \bigcup_{s \in Next(n)} in(s)$

Zur Berechnung sind am Anfang die Mengen $in(n)$ und $out(n)$ für jeden Knoten leer. Die Gleichungen werden wie in Algorithmus 4 gezeigt, iterativ so lange auf die Kontrollflussknoten angewendet, bis sich die Mengen nicht mehr ändern. Der Algorithmus konvergiert schneller, wenn die CFG-Knoten in umgekehrter Reihenfolge (engl. *reverse postorder*) durchlaufen werden. Für das Beispiel in Abb. 5.3.6 wäre die Reihenfolge n_5, n_4, n_3, n_2, n_1. Für CFGs ohne Schleifen würden sich die Mengen schon nach einem Durchlauf nicht mehr ändern.

Auf dem hier verwendeten CDFG kann die Lebenszeitanalyse nicht auf Anweisungen durchgeführt werden, sondern auf den Ein- und Ausgabeknoten der jeweiligen Grundblöcke. Ein Eingabeknoten referenziert den Ausgabeknoten eines Vorgängerblocks, wenn sich beide auf dieselbe Variable beziehen. Damit ist $use(n)$ die Menge der in Block n von Eingabeknoten und $def(n)$ die Menge der von Ausgabeknoten referenzierten Variablen. Die Berechnung von $in(n)$ und $out(n)$ bleibt identisch. Dadurch erhält man zu jedem Grundblock die Menge der lebendigen Variablen vor und nach dessen Ausführung. Durch $in(n)$ kann dann die Menge der lebendigen Eingabeknoten und durch $out(n)$ die lebendigen Ausgabeknoten von Block n ermittelt werden, die man für die Bestimmung der E_{DD}-Kanten eines CDFGs benötigt.

Algorithm 4: Lebenszeitanalyse

Data: Knotenmenge N, use, def
Result: in, out
begin
  foreach $n \in N$ **do**
    $in(n) \leftarrow \emptyset$
    $out(n) \leftarrow \emptyset$
  repeat
    $in'(n) = in(n)$
    $out'(n) = out(n)$
    $in(n) = use(n) \cup (out[n] \setminus def[n])$
    $out(n) = \bigcup_{s \in Next(n)} in(s)$
  until $in'(n) = in(n) \wedge out'(n) = out(n) \forall n \in N$;
end

5.3.7 Erstellung der E_{DD}-Kanten

Wie in Abschnitt 5.1 erwähnt, dienen die E_{DD}-Kanten zur Darstellung des blockübergreifenden Datenflusses zwischen Aus- und Eingabeknoten. Nach der Erstellung des Kontrollflussgraphen und der DFGs in den einzelnen Grundblöcken sind noch keine E_{DD}-Kanten vorhanden. In Abb. 5.3.7 sind drei Grundblöcke mit ihrem Datenfluss dargestellt. In Block bb_0 wird in einem Ausgabeknoten n_{a0} in eine Variable a geschrieben. bb_0 besitzt zwei direkte Nachfolger bb_1 und bb_2, dort wird a von den Eingabeknoten n_{a1} bzw. n_{a2} gelesen. Hier werden zwei E_{DD}-Kanten benötigt, um den Datenfluss von n_{a0} nach n_{a1} und n_{a0} nach n_{a2} darzustellen. Die Ein- und Ausgabeknoten müssen nach Erstellung der E_{DD}-Kanten nicht mehr notwendigerweise dieselben Variablen referenzieren, eigentlich müssen lediglich Informationen über den benutzten Datentypen erhalten bleiben. Dennoch wird gefordert, dass die Knoten weiterhin Variablen referenzieren und dass alle durch E_{DD}-Kanten verbundenen Knoten auch dieselbe Variable referenzieren müssen. Bestimmte Transformationen und Optimierung vereinfachen sich dadurch, da hier die E_{DD}-Kanten abgebaut, die Graphentransformationen durchgeführt und die E_{DD}-Kanten anschließend wieder aufgebaut werden können.

Die Verwendung dieser Kantenart für blockübergreifenden Datenfluss anstelle von normalen DFG-Kanten hat den Grund, dass sonst Zyklen im DFG bei Schleifen entstehen, was zusätzlichen Aufwand vor allem bei der rekursiven Traversierung bedeutet. Außerdem sind durch diese speziellen Kanten die Datenflüsse in den einzelnen Blöcken stärker voneinander getrennt, so dass bei lokalen DFG-Transformationen beim Traversieren nicht überprüft werden muss, ob man sich noch im aktuellen Block befindet. Für die Erstellung der E_{DD}-Kanten muss die im vorigen Abschnitt vorgestellte Lebenszeitanalyse erweitert werden. Nach der Lebenszeitanalyse kennt man vor und hinter jedem Block die lebendigen Variablen. Jedoch ist der Ausgabeknoten, der eine lebendige Variable in $in(n)$ zuletzt modifiziert hat, unbekannt. Genau dieser letzte Ausgabeknoten ist aber für die E_{DD}-Kanten

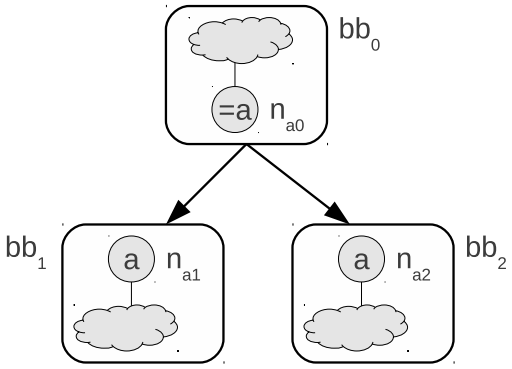


Abbildung 5.3.7: CDFG mit Ein- und Ausgabeknoten. Zwischen $n_{a0} \rightarrow n_{a1}$ und $n_{a0} \rightarrow n_{a2}$ müssen noch E_{DD}-Kanten hergestellt werden.

relevant. Algorithmus 5 zeigt die notwendigen Schritte, um diese Knoten zu ermitteln und die E_{DD}-Kanten zu setzen. Dabei wird zunächst mit Hilfe der bei der Lebenszeitanalyse ermittelten Variablen für jeden Grundblock *bb* im CDFG die Menge der lebendigen Ausgabeknoten *outNodes* ermittelt. Anschließend werden diese Mengen nach und nach mit lebendigen Knoten aus Vorgängerblöcken ergänzt, wenn die entsprechende Variable nicht in einem neuen Ausgabeknoten neu geschrieben wurde. Wird in einem Block eine Variable von einem Eingabeknoten referenziert und kommt gleichzeitig bei einem lebendigen Ausgabeknoten aus einem Vorgängerblock vor, so wird eine E_{DD}-Kante von diesem Ausgabeknoten zum Eingabeknoten gesetzt. *outNodes* wird so lange aktualisiert und die entsprechende Kanten erstellt, bis sich *outNodes* nicht mehr ändert.

Algorithm 5: *SetEDD* - Setzen aller E_{DD}-Kanten im CDFG

```
Data: CDFG, out
begin
    outNodes speichert für jeden Block bb im CDFG die lebendigen Ausgabeknoten.
    Füge für jeden Block bb im CDFG die Ausgabeknoten in outNodes[bb] ein,
    dessen Variable in out(bb) enthalten ist.
    repeat
        outNodes' ← outNodes
        foreach bb im CDFG do
            foreach s ∈ Next(bb) do
                outNodes[s] ← UpdateEDD(outNodes, bb, s)
    until outNodes ≠ outNodes' ;
end
```

Algorithm 6: *UpdateEDD* - Einzelne E_{DD}-Kanten von Block *bb* nach *s* setzen und aliveNodes[s] aktualisieren

Data: $aliveNodes, bb, s$
Result: $newAliveNodes[s]$
begin
  $redefinedNodes \leftarrow \emptyset$
  foreach $n_{bb} \in aliveNodes[bb]$ **do**
    foreach n_{in} *in* $s \wedge n_{in} \in N_{in}$ **do**
      if $Var(n_{bb}) = Var(n_{in})$ **then**
        Erstelle E_{DD}-Kante von n_{bb} nach n_{in}
    foreach n_{out} *in* $s \wedge n_{out} \in N_{out}$ **do**
      if $Var(n_{bb}) = Var(n_{out})$ **then**
        $redefinedNodes \leftarrow redefinedNodes \cup n_{out}$
  foreach $n_{bb} \in aliveNodes[bb]$ **do**
    if $n_{bb} \notin redefinedNodes$ **then**
      $aliveNodes[s] \leftarrow aliveNodes[s] \cup n_{bb}$
  return aliveNodes[s]
end

5.4 Operationen

```
(x,y) = div(a,b);
m = y - 4;
```

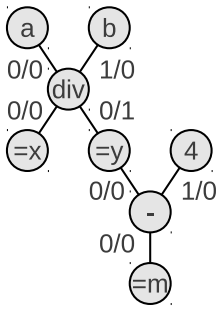


Abbildung 5.4.1: Datenflussgraph mit Indizes, die Operanden und Ergebnisse referenzieren

Eine Operation kann mehrere Operanden besitzen und auch mehrere Ergebnisse liefern. Dementsprechend besitzen die Operanden zwei Indizes. Der erste bestimmt die Position des Operanden in der Operation, der zweite gibt an, auf welches Ergebnis des Operanden zugegriffen wird. `div` sei eine Operation, die den Quotienten sowie den Rest von zwei Zahlen berechnet. Abb. 5.4.1 zeigt zwei arithmetische Ausdrücke sowie den zugehörigen Datenflussgraphen mit den Indizes an den Kanten. Der Knoten `"=y"` z.B. greift mit seinem Operanden auf das zweite Ergebnis der Operation `div` zu, dementsprechend lauten die beiden Indizes (0,1). Da "`-`" mit seinem zweiten Operanden das Ergebnis vom Knoten "`4`"

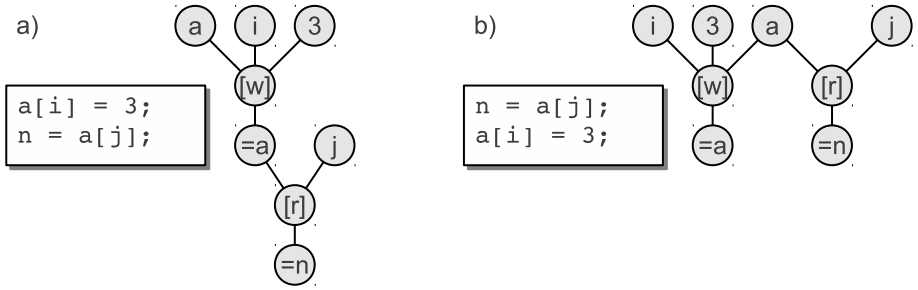


Abbildung 5.4.2: Beispiele für verschiedene Arrayoperationen

referenziert, sind dort die Indizes (1,0).

Die folgenden Abschnitte zeigen Operationen, die spezielle Eigenschaften besitzen und nicht nur durch einen einzigen DFG-Knoten dargestellt werden können. Diese setzen sich aus mehreren Knoten zusammen und können sich Knoten mit anderen komlexen Operationen teilen.

Arrayoperationen

Arrayoperationen sind Anweisungen zur Modifikation und zum Auslesen von Feldern und bestehen, wie in Abbildung 5.4.2 gezeigt, aus mehreren Datenflussknoten. Der zentrale Knoten kennzeichnet die eigentliche Operation, die vom Typ Array lesen "[r]" oder Array schreiben "[w]" sein kann. Der erste Operand ist ein Knoten, der über eine Variable eine primäre Referenz auf das Array darstellt, der zweite Operand ist der Arrayindex. Das Resultat von Leseoperationen ist der gelesene Wert selber. Schreiboperationen besitzen noch mit dem zu schreibenden Wert einen dritten Operanden und ebenfalls als Nachfolger einen Resultatknoten. Damit unterscheidet sich der hier verwendete Zwischencode zu Datenflussgraphen, die z.B. in [83] verwendet werden, wo Schreiboperationen eine Senke darstellen und keine Nachfolger haben, bzw. wo der Nachfolger der geschriebene Wert ist. Der Resultatknoten beinhaltet eine sekundäre Referenz auf dieselbe Arrayvariable. Wie in Abbildung 5.4.2a dargestellt, benutzen darauf folgende Arrayoperationen diesen Knoten wieder als primäre Referenz. Dies hat den Vorteil, dass die Reihenfolge von aufeinanderfolgenden Arrayoperationen direkt im Datenfluss erhalten bleibt. Arrayzugriffe, die einer Leseoperation folgen, benutzen dieselbe primäre Arrayreferenz erneut (Abb. 5.4.2b). Operationen, die das Array modifizieren, erstellen quasi ein neues Array, das von allen Folgeoperationen verwendet wird.

Für ein Array ist die erste primäre Referenz in einem Grundblock immer ein Eingabeknoten, die letzte sekundäre Referenz immer ein Ausgabeknoten, so dass auch hier blockübergreifende Datenabhängigkeiten mit E_{DD}-Kanten dargestellt werden können. Durch die Verwendung von primären und sekundären Referenzen können auch parallele Schreib- und Leseoperationen dargestellt werden, wie in Abbildung 5.4.3 gezeigt. Hier beziehen sich mehrere Zugriffe auf dieselben primären und sekundären Arrayknoten. Sequentielle Lesezugriffe sind immer parallel (5.4.3a), Schreiboperationen können sowohl sequentiell als

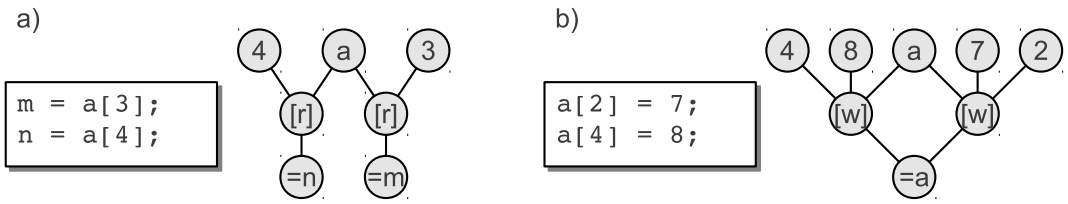


Abbildung 5.4.3: Beispiele für parallele Arrayzugriffe

auch parallel (5.4.3b) dargestellt werden, je nachdem, auf welche Arrayreferenz sie sich beziehen. Allerdings dürfen sich bei der parallelen Darstellung die Arrayindizes niemals überschneiden, so dass gewährleistet ist, dass die Ausführungsreihenfolge der parallelen Arrayoperationen keine Auswirkungen auf den Arrayinhalt besitzt. TransC unterstützt die Definition von mehrdimensionalen Arrays. Zugriffe auf diese Arrays besitzen dann statt einem mehrere Indexoperanden und dementsprechend auch weitere Vorgängerknoten im DFG.

Im Gegensatz zu anderen Zwischenformaten wie etwa in Spark[44] besitzen die Arrayzugriffe neben den Arrayreferenzen, Schreibdaten und Indices noch einen weiteren Operanden mit einem Booleschen Wert. Nur wenn dieser wahr ist, wird die Operation ausgeführt. Dieses Konzept wird u.a. bei Mikroprozessoren verwendet, wo die Ausführung eines Befehl von einem Statusflag abhängt [3]. Auf diese Weise können Arrayzugriffe effizienter implementiert werden, da bedingte Zugriffe direkt in den Datenpfad integriert werden können und nicht zusätzliche Verzweigungen im Kontrollfluss benötigen. Als Folge des einfachen Kontrollflusses können Grundblöcke zusammengefasst werden, wodurch sich weitere Optimierungsmöglichkeiten eröffnen. Bedingte Zugriffe scheinen nur bei den Schreiboperationen sinnvoll zu sein, da diese Seiteneffekte aufweisen. Aber auch bei Leseoperationen bringt diese Bedingung Vorteile, da auf diese Weise nach der Implementierung in Hardware und der Abbildung von Arrays auf Speicher die Speicherports nur verwendet werden, wenn man Daten tatsächlich benötigt. Wenn sich mehrere Prozesse dasselbe Array teilen, entstehen so weniger Zugriffe, was die Wahrscheinlichkeit von auftretenden Wartezyklen verringert. Wartezyklen sind notwendig, wenn zwei Prozesse gleichzeitig einen Speicherport benutzen wollen. Abbildung 5.4.4 zeigt den Vergleich der Arrayoperationen mit und ohne Zugriffsbedingung.

Strukturoperationen

Die TransC-Sprache unterstützt ähnlich wie C benutzerdefinierte Datenstrukturen (*struct*), dementsprechend müssen Operationen zur Modifikation und zum Auslesen im Zwischencode vorhanden sein. Wie Arrayoperationen bestehen Strukturoperationen aus mehreren Knoten, deren Zusammenstellung denen der Arrayoperationen sehr ähnlich ist. Variablen, die Datenstrukturen repräsentieren, enthalten eine Liste von weiteren Variablen, die die einzelnen Elemente der Datenstruktur darstellen. Diese Elemente können sich auch aus Arrays oder weiteren Strukturvariablen zusammensetzen, wodurch verschachtelte Daten-

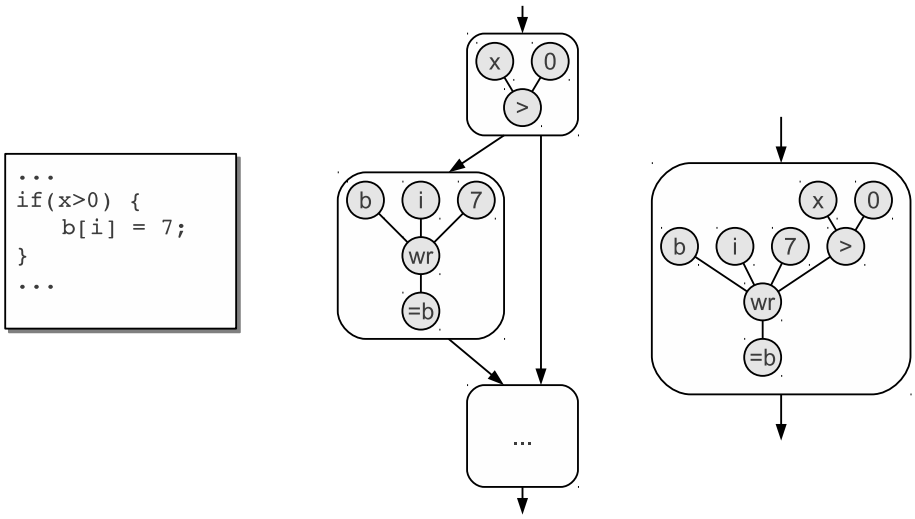


Abbildung 5.4.4: Arrayzugriffe mit und ohne zusätzlichen Bedingungsoperanden. Mit dem zusätzlichen Operanden lassen sich Grundblöcke weiter zusammenfassen.

strukturen entstehen. Der zentrale Knoten bei Strukturoperationen stellt eine Lese- bzw. Schreiboperation dar, dessen erster Operand ein Knoten ist, der die Variable der Datenstruktur referenziert. Somit besitzen Strukturoperationen ähnlich wie Arrayoperationen primäre Referenzen. Der zweite Operand ist der Index des Elements, das geschrieben bzw. gelesen werden soll. Bei verschachtelten Strukturen gibt es mehrere Indizes. Zeigt der erste Index beispielsweise auf ein Array innerhalb der Datenstruktur, so zeigt der zweite Index auf das Arrayelement. Bei Schreiboperationen gibt es mit dem zu schreibenden Wert einen weiteren Operanden. Dieser zu schreibende Wert kann ein Basisdatentyp wie Integer oder Boolean sein, aber auch wiederum eine komplette Datenstruktur, wenn das Element auch eine Struktur desselben Typs ist, auf das der Index zeigt. Bei Leseoperationen ist der Ergebnisknoten der aus der Datenstruktur gelesene Wert. Schreiboperationen haben, wie Arrayoperationen, als Nachfolger eine sekundäre Referenz auf die Strukturvariable. Abb. 5.4.5 zeigt Beispielcode für den Zugriff auf Datenstrukturen mit dem zugehörigen Datenflussgraph. Die Knoten, die für die Indizierung der Elemente verwendet werden, sind in weiß dargestellt.

Neben einfachen Datenstrukturen können auch Arrays von Datenstrukturen definiert werden. In diesem Fall handelt es sich um eine Arrayoperation, die mehrere Operanden zur Indexierung besitzt. Der erste Indexoperand ist der Arrayindex, mit den anderen Indexoperanden greift man dann auf die Einträge der Datenstruktur zu.

Streamoperationen

Sowohl bei den synchronen als auch bei den asynchronen Streams gibt es Lese- und Schreiboperationen. Die asynchronen Streams werden wie gewöhnliche Variablenoperationen behandelt, da es sich hier letztendlich um das Lesen einer Variable aus einem anderen Prozess handelt. Auch die Vorgehensweise beim Einfügen in den Grundblock ist identisch. Abbil-

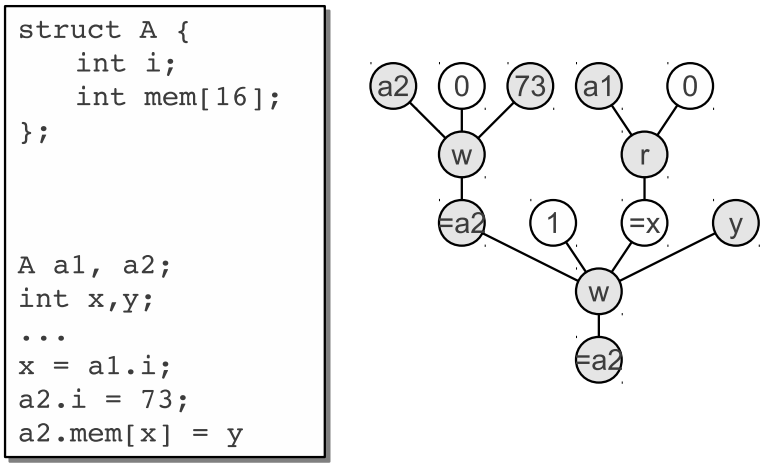


Abbildung 5.4.5: Beispiel für Strukturoperationen

dung 5.4.6a zeigt den resultierenden DFG von zwei Leseoperationen. Die beiden Zugriffe werden zusammengefasst, wenn es keine zeitlichen Beschränkungen gibt. Genau so verhält es sich bei Schreiboperationen, wie in Abb. 5.4.6b dargestellt wird. Wenn keine Zeitbedingungen vorhanden sind, kann der zeitliche Abstand zwischen den beiden Zugriffen als infinitesimal klein gesehen werden. Dadurch hat die erste Schreiboperation keine Auswirkungen, da der Stream sofort überschrieben wird. Bei zwei Leseoperationen würde sich der Wert des Streams in diesem kurzen Zeitraum nicht ändern, weshalb eine Leseoperation ausreicht. Bei Zeitbedingungen oder *sync*-Befehlen zwischen zwei Streamoperationen würden diese in verschiedenen Grundblöcken stehen, was ein Zusammenfassen verhindert. Die Zeitbedingungen werden in Form von Funktionsaufrufen modelliert, wobei alle Anweisungen nach dem Aufruf in einen neuen Grundblock eingefügt werden (siehe nächsten Abschnitt).

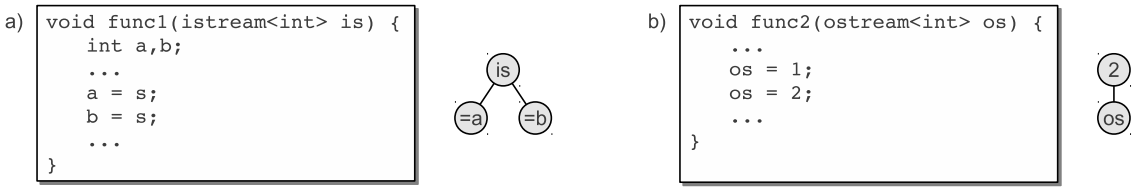


Abbildung 5.4.6: Asynchrone Streamoperationen

Diese Zusammenfassung kann bei synchronen Streams nicht durchgeführt werden, da jede Schreib- bzw. Leseoperation den Programmablauf beeinflusst und entweder die FIFO zwischen den zu kommunizierenden Prozessen ändert oder zumindest eine synchronisierende Wirkung hat. Wie bei Arrayoperationen auch muss die Reihenfolge beibehalten werden. Operationen auf synchronen Streams setzen sich aus mehreren DFG-Knoten zusammen. Der zentrale Knoten ist die eigentliche Schreib- oder Leseoperation, wobei der erste Operand wie bei den Arrayoperationen eine primäre Referenz auf die Streamvariable besitzt. Bei Schreiboperationen ist der zweite Operand der zu schreibende Wert. Weitere Operanden kommen nicht vor, da hier im Vergleich zu Feldern oder Datenstrukturen

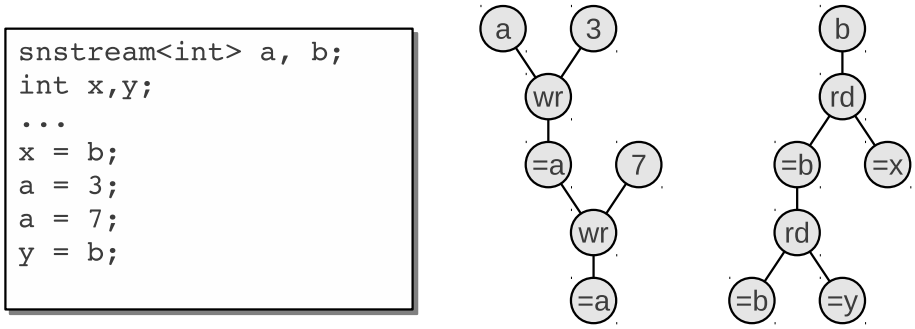


Abbildung 5.4.7: Synchrone Streamoperationen

nicht indiziert wird. Da sowohl Lese- als auch Schreiboperationen den synchronen Stream verändern, also Daten entnehmen oder zufügen, besitzt der zentrale Knoten bei beiden Operationen einen festen Nachfolgeknoten, der eine sekundäre Referenz auf die Streamvariable darstellt. Leseoperationen erzeugen noch mit dem zu lesenden Wert ein weiteres Resultat. Aufeinanderfolgende Operationen, die auf denselben synchronen Stream zugreifen, besitzen als primäre Referenz die sekundäre Referenz der Vorgängeroperation, ähnlich wie bei Array- und Strukturoperationen. Auf diese Weise bleibt die Reihenfolge direkt im Datenfluss erhalten, gleichzeitig erkennt man bei unterschiedlichen Streams Operationsfolgen, die parallel abgearbeitet werden können. Abb. 5.4.7 zeigt ein Programm mit Zugriffen auf synchrone Streams und den zugehörigen Datenflussgraph. Die einzelnen Graphen können parallel abgearbeitet werden, was aus der reinen Anweisungsfolge nicht ohne weiteres erkennbar ist.

Prozessaufrufe

Prozessaufrufe und Funktionsaufrufe besitzen einen *call*-Knoten im DFG, der die Operation darstellt. In diesem Knoten ist der CDFG des auszuführenden Prozesses bzw. der Funktion referenziert. Ob es sich um einen Prozess- bzw. Funktionsaufruf handelt, ist nicht in dem DFG-Knoten, sondern im aufgerufenen Objekt gespeichert. Argumente werden wie in Abb. 5.4.8 gezeigt, als Operandenknoten repräsentiert. Der gesamte Aufruf ist im normalen Datenfluss des DFG integriert.

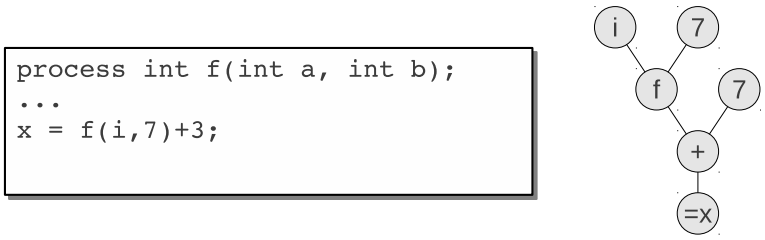


Abbildung 5.4.8: Prozessaufruf

Auch Arrays, Datenstrukturen und Streams in der Argumentenliste des Prozesses werden in Form von Operandenknoten übergeben, die die Variable des jeweiligen Konstrukts

```
process int g(
    istream<int> a,
    sostream<int> b,
    int c[]);
...
g(x,y,z);
```

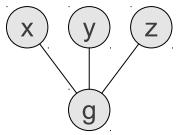


Abbildung 5.4.9: Prozessaufruf mit Streams und Arrays

als primäre Referenz beinhalten. Abb. 5.4.9 zeigt einen Prozess, dem ein asynchroner Stream, ein synchroner Stream und ein Array übergeben wird. Da das erste Argument ein asynchroner Stream ist, wird der erste Operand im DFG als Streamübergabe interpretiert. Wäre das erste Funktionsargument eine normale Integervariable, so würde der erste Operandenknoten als Streamlese-Operation behandelt werden, wobei der aus dem Stream entnommene Wert übergeben wird. Im Datenfluss selber ist dieser Unterschied bei den asynchronen Streams nicht erkennbar. Da normale Zugriffe auf synchrone Streams hingegen immer eine Streamlese- bzw. Schreiboperation besitzen (Seite 108), kann man bereits im Datenfluss die Übergabe erkennen. Genauso verhält es sich bei Arrayübergaben.

Obwohl die Funktionen und Prozesse die übergebenen Streams und Arrays modifizieren können, besitzen sie keine sekundären Referenzen auf die Konstrukte. Dies hat den Grund, dass die Parallelität hier nicht eingeschränkt werden soll, wie es beispielsweise bei mehreren sequentiellen Arrayoperationen geschieht. Zwei hintereinander aufgerufene Prozesse, die auf denselben Stream oder dasselbe Array zugreifen, sollen weiterhin parallel ablaufen, wie in Abb. 5.4.10a gezeigt. Würde `p1` eine sekundäre Referenz erzeugen, so würde sich `p2` auf diese beziehen. Prozesse und Funktionen werden beim Aufruf gleich behandelt, daher besitzen auch Funktionen keine sekundären Referenzen. Wären `p1` und `p2` Funktionen, so kämen sie in unterschiedliche Grundblöcke (Abb. 5.4.10b). Da im Datenflussgraphen die Aufruffolge von Funktionen nicht gespeichert wird, wird wie in vorigen Abschnitten beschrieben, nach dem Einlesen eines Funktionsaufrufs der aktuelle Grundblock abgeschlossen, mit *sync* markiert und ein Folgeblock erzeugt, in den die danach kommenden Anweisungen eingefügt werden. Auf diese Weise bleibt die ursprüngliche Ausführungsreihenfolge erhalten. Da bei Funktionsaufrufen der Programmablauf des aufrufenden Prozesses immer angehalten wird und auf das Ende gewartet wird, laufen die Funktionen auch nicht parallel ab. Somit vereinfachen sich Graphentransformationen und auch die Codegenerierung im Backend, da Prozesse und Funktionen immer gleich behandelt werden können. Bei Prozessen, die innerhalb eines Blocks gestartet werden, ist die Ausführungsreihenfolge irrelevant. Nach dem TransC-Prozessmodell laufen alle Prozesse parallel zueinander ab. Muss die Ausführungsreihenfolge beibehalten werden, so müssen sie explizit mit dem `sync`-Befehl (Abschnitt 3.5) in verschiedene Grundblöcke gesetzt werden.

Beim expliziten Warten auf das Ende eines Prozesses gibt es eine spezielle Operation (siehe Abschnitt 3.5), da Aufruf und das Warten in unterschiedlichen Blöcken vorkommen kann. Dies geschieht mit Hilfe eines *call-end*-Knotens, der intern den Aufruf-Knoten refe-

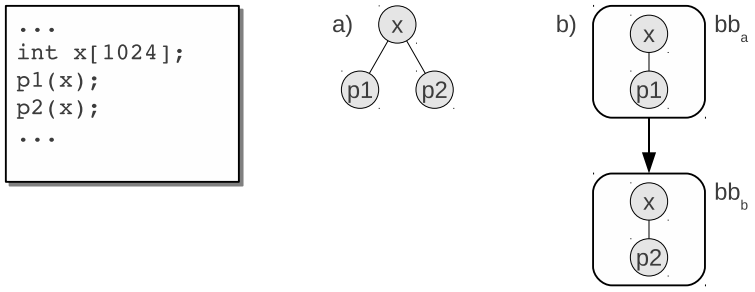


Abbildung 5.4.10: Unterschied zwischen Prozessaufrufen (a) und Funktionsaufrufen (b). Prozesse werden parallel ausgeführt, bei Funktionen wird durch die Trennung in aufeinanderfolgende Grundblöcke eine sequentielle Ausführung erzwungen.

renziert. Bei dieser Operation stoppt der Programmablauf wie einer Funktion und es wird auf das Ende des Prozesses gewartet.

Kapitel 6

Optimierungen

Dieses Kapitel zeigt Optimierungen auf, die von bestehenden Verfahren abgeleitet sind und auf den verwendeten Zwischencode angepasst wurden. Aufgrund der Graphenstrukturen sind viele der aufgezeigten Transformationen relativ einfach anwendbar. Außerdem werden Optimierungen gezeigt, die auf einer neu entwickelten Variante der partiellen Funktionsauswertung basieren und auf einfache Weise Bitbreitenreduktion oder die Parallelisierung von Arrayzugriffen ermöglichen.

Transformationen, die zwei aufeinanderfolgende Blöcke miteinander verschmelzen sind nur erlaubt, wenn der erste Block nicht mit einem *sync* markiert ist. Bei der Verschmelzung von zwei Blöcken, bei dem der zweite diese Eigenschaft besitzt, ist der entstehende Block ebenfalls mit *sync* markiert.

6.1 Optimierungen auf der CDFG-Struktur

Konstantenfaltung und -propagation

Aufgrund der CDFG-Struktur wird zwischen Konstantenfaltung und -propagation nicht mehr unterschieden. Statt in Anweisungen Variablen durch den ihnen zuvor zugewiesenen Wert zu ersetzen, können Konstanten innerhalb eines Datenflussgraphen entlang der Kanten propagiert werden. Sobald alle Operanden eines Knotens bekannt sind, kann dieser durch das Ergebnis seiner Operation ersetzt werden. Abbildung 6.1.1a zeigt die Anwendung der ursprünglichen Konstantenfaltung, 6.1.1b die Konstantenpropagation auf dem Graphen. Konstante Knoten können nicht nur innerhalb eines DFGs, sondern auch grundblockübergreifend berechnet werden. Wenn ein Eingabeknoten nur eine einzige E_{DD}-Kante besitzt und der damit verbundene Ausgabeknoten eine Konstante als Operand hat, so kann der Eingabeknoten unter Berücksichtigung der Datentypen ebenfalls durch diese Konstante ersetzt werden.

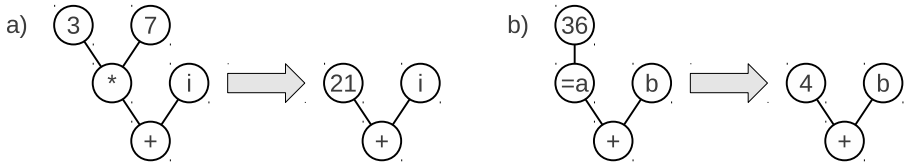


Abbildung 6.1.1: Faltung von konstanten Knoten, als Operation (a) und als Zuweisung (b). Variable `a` sei vom Typ Integer mit einer Breite von 5-Bit. Dies muss beim Ersetzen berücksichtigt werden.

Elimination unbenutzter Berechnungen

Bei der Elimination unbenutzter Berechnungen werden Operationen entfernt, die das Ergebnis an eine nicht mehr lebendige Variable zuweisen. Eine vorherige Datenflussanalyse ist somit notwendig. Bei dem hier verwendeten Zwischenformat haben die ursprünglichen Variablen keine Relevanz, da alle Datenabhängigkeiten durch DFG-Kanten bzw. die in Abschnitt 5 beschriebenen E_{DD}-Kanten dargestellt werden. Um die E_{DD}-Kanten zu erzeugen, ist die in Abschnitt 5.3.6 dargestellte Datenflussanalyse notwendig. Einen Knoten, dessen Ergebnis nicht mehr benötigt wird, kann man dann an fehlenden ausgehenden Kanten erkennen. Ein Knoten kann somit aus dem Datenflussgraphen entfernt werden, wenn er

- keine Nachfolgeknoten im DFG besitzt,
- keine ausgehenden E_{DD}-Kanten hat und
- seine Operation keine Seiteneffekte aufweist.

Durch das Entfernen eines Knotens erfüllen ggf. dessen Vorgänger auch die o.a. Kriterien, sodass auch diese aus dem Graphen entnommen werden können. Auf diese Weise ist es möglich, alle nicht benötigten Berechnungen zu erkennen und zu eliminieren. Eine Ausnahme bilden Schleifen, da sie über die E_{DD}-Kanten Zyklen erzeugen, die vorher identifiziert und aufgelöst werden müssen. Abb. 6.1.2 zeigt einen CDFG mit E_{DD}-Kanten. Knoten, die gelöscht werden können, sind dunkelgrau markiert. Der erste Knoten, auf den die oben genannten Bedingungen zutreffen, ist n_0.

Assoziative Umformungen

Assoziative Umformungen werden durchgeführt, um den Parallelitätsgrad im Datenfluss zu erhöhen oder Konstantenfaltungen zu ermöglichen, wodurch Berechnungen zur Laufzeit wegfallen. In diesem Projekt werden assoziative Umformungen auf Knoten mit Additionen, Multiplikationen und bestimmten Booleschen Operationen angewendet. Bei Datentypen wie Fließkomma können sich durch die Umformungen aufgrund von Auslöschungen und Ungenauigkeiten beim Runden die Ergebnisse verändern, weshalb die Berechnungsreihenfolge dort nicht modifiziert werden darf, bzw. nur wenn dies gewünscht ist.

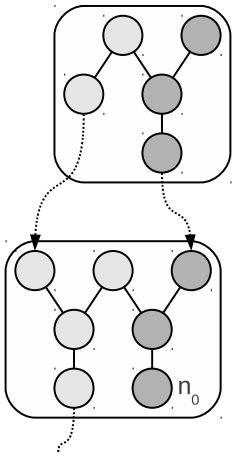


Abbildung 6.1.2: CDFG mit Knoten, die entfernt werden können

Wenn mehrere Datenflussknoten mit den gleichen geeigneten Operationen einen zusammenhängenden Teilgraphen bilden, so sind dies Kandidaten für eine Umformung. Abb. 6.1.3a zeigt ein Beispiel für solch einen Teilgraphen.

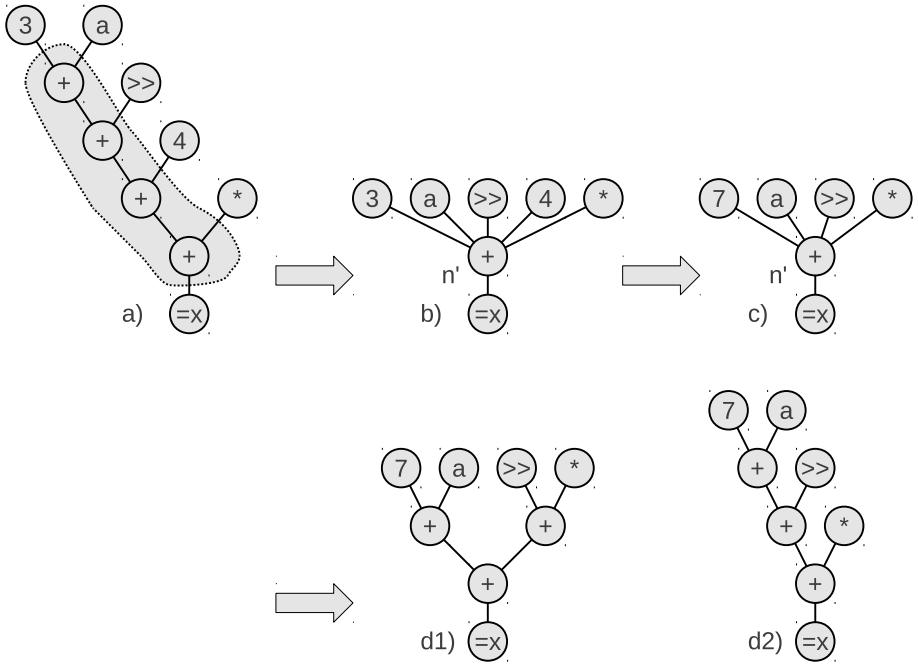


Abbildung 6.1.3: Assoziative Umformung zur Konstantenfaltung bzw. Erhöhung der Parallelität

Die Knoten im Teilgraphen werden zusammengefasst und durch einen einzigen Knoten n' ersetzt (Abb.6.1.3b). Jetzt können alle konstanten Operanden von n' berechnet werden, so dass nur noch ein konstanter Vorgänger übrig bleibt (Abb.6.1.3c). Anschließend wird n' nach verschiedenen Kriterien wieder expandiert. Entweder kann der Graph in die Tiefe oder zur Erhöhung der Parallelität in die Breite gebaut werden (Abb.6.1.3d1 bzw.

6.1.3d2). Wenn man von bestimmten Operanden weiß, dass sie schleifeninvariant sind (siehe Abschnitt 2.4.1), können diese separat aufgebaut werden, so dass sie sich nicht eine Operation mit schleifenvarianten Operanden teilen. Auf diese Weise können bei der *Loop Invariant Code Motion* bessere Ergebnisse erzielt werden, wenn diese Knoten zusammen mit der Operation dann vor die Schleife verschoben werden.

Globale Eliminierung gemeinsamer Teilausdrücke

In dieser Arbeit werden für die globale Eliminierung gemeinsamer Teilausdrücke (engl. *Global Common Subexpression Elimination, GCSE*) immer zwei Grundblöcke betrachtet und analysiert, ob sie redundante Operationen besitzen. Werden solche Ausdrücke gefunden, können sie entfernt werden, so dass sie nur einmal berechnet werden müssen. Die globale Eliminierung wird u.a. in [20] beschrieben.

In dieser Arbeit kann die lokale Eliminierung von gemeinsamen Ausdrücken (*LCSE*) innerhalb eines Blockes bereits mit dem in Abschnitt 5.3.3 vorgestellten Algorithmus ausgeführt werden, auf dem die globale Eliminierung aufbaut. Hier wird zwischen zwei Methoden der globalen Eliminierung unterschieden. Bei der ersten besitzen zwei Grundblöcke b_0 und b_1 gemeinsame Ausdrücke, wobei b_1 von b_0 dominiert wird. Bei der zweiten Methode ist diese Dominanz nicht vorhanden. Der DFG von b_0 wird mit d_0 bezeichnet, wobei alle seine Knoten in der Menge N_0 zusammengefasst sind. Mit b_1 wird entsprechend verfahren. Zwei Blöcke b_0 und b_1 sind Kandidaten für GCSE, wenn sie gleiche Teilgraphen aufweisen.

Wenn b_0 b_1 dominiert, so kann d_1 aus b_1 entfernt und sequentiell an d_0 angefügt werden ($d0{+}d1$, siehe Abschnitt 5.3.2). Durch LCSE können jetzt gemeinsame Teilausdrücke eliminiert werden, wodurch Knoten wegfallen. Dabei muss immer der Knoten aus N_1 gelöscht werden und der Knoten aus N_0 erhalten bleiben, dementsprechend sind die gelöschten Knoten aus N_1 herauszunehmen.

Dort, wo zwei miteinander verbundene Knoten nicht denselben Knotenmengen angehören, wird die Kante aufgetrennt und durch einen Ausgabe- und einen Eingabeknoten ersetzt. Der Ausgabeknoten wird N_0 und der Eingabeknoten N_1 zugeordnet, sodass sie derselben Knotenmenge wie dessen Vorgänger bzw. Nachfolger angehören. Anschließend werden Eingabe- und Ausgabeknoten durch eine E_{DD}-Kante verbunden. Kanten hinter Variablen- bzw. Konstantenleseoperationen sollten nicht aufgetrennt werden, da diese Operationen trivial sind und das Auftrennen weiterer Leseoperationen mit sich führt. Stattdessen sollten diese dupliziert werden, sodass die eine Operation ihr Ergebnis an einen Knoten aus N_0 und die andere an einen Knoten aus N_1 weiterleitet. Das Duplikat muss der entsprechenden Knotenmenge ebenfalls zugeordnet werden. Jetzt sind die Knoten aus N_0 nicht mehr direkt mit den Knoten aus N_1 verbunden, sodass die Knoten aus N_1 wieder in b_1 verschoben werden können. Die erzeugten E_{DD}-Kanten verlaufen jetzt von b_0 nach b_1. Abbildung 6.1.4 zeigt die Schritte einer GCSE, wenn ein Block b_0 den anderen Block b_1 dominiert. Die Knoten, die in N_0 enthalten sind, sind weiß, die anderen grau gefärbt. In 6.1.4b werden beide DFGs in d_0 parallel zusammengefügt und eine lokale Eliminierung

der gemeinsamen Teilausdrücke vorgenommen, die Knoten aus N_0 bleiben dabei erhalten. Anschließend werden die Knoten mit den trivialen Operationen kopiert (6.1.4c). Danach werden die entsprechenden Kanten aufgetrennt (6.1.4d) und zum Schluss die Knoten, die in N_1 enthalten sind, nach b_1 zurück verschoben (6.1.4e).

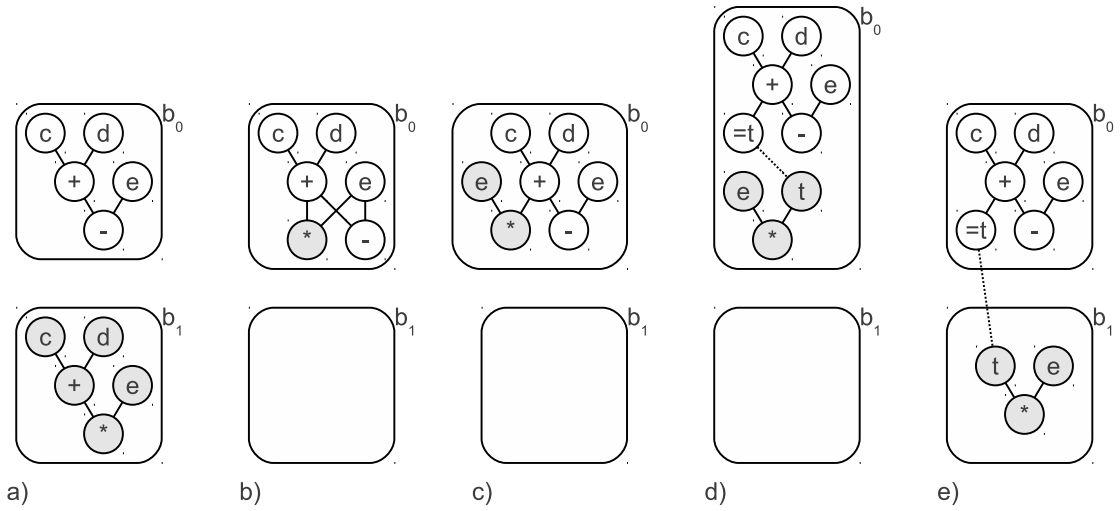


Abbildung 6.1.4: Schritte der GCSE, wenn Grundblock b_0 den Block b_1 dominiert.

Wenn zwischen zwei Blöcken gemeinsame Ausdrücke entfernt werden sollen, jedoch keine Dominanzbeziehung zwischen beiden vorhanden ist, so wird die Eliminierung etwas aufwändiger. Die gemeinsamen Ausdrücke müssen in einen neuen Block b_n verschoben werden, der sowohl b_0 als auch b_1 dominiert und sich in den blockübergreifenden Datenfluss integrieren lässt. Dazu werden zunächst sowohl d_0 als auch d_1 aus den Blöcken herausgenommen und parallel in b_n eingefügt $(d_0|d_1)$. Dort wird ebenfalls die lokale CSE ausgeführt, wobei es jetzt bei der Entfernung der Knoten irrelevant ist, aus welchen Graphen sie kommen. Dort wo zwei miteinander verbundene Knoten unterschiedlichen Knotenmengen angehören, wird die entsprechende Kante wieder aufgetrennt, Eingabe- und Ausgabeknoten eingefügt sowie durch eine E_{DD}-Kante verbunden. Allerdings werden jetzt der Ausgabeknoten und alle seine Vorgänger einer Menge N_n zugeordnet und entsprechend aus N_0 und N_1 entfernt. Variablen- und Konstantenleseoperationen können hier ebenfalls vorher dupliziert und derselben Menge wie ihr Nachfolger zugeordnet werden. Dadurch, dass auch Vorgängerknoten N_n zugefügt worden sind, entstehen zusätzliche Kanten, die Knoten aus verschiedenen Mengen verbinden. Auch diese neuen Kanten müssen aufgelöst werden. Sind die drei Graphen, dessen Knoten aus N_0, N_1 bzw. N_n stammen, nicht mehr zusammenhängend, können die Knoten aus N_0 nach b_0 und aus N_1 wieder nach b_1 verschoben werden. b_n ist so in den Kontrollfluss einzufügen, dass er sowohl b_0 als auch b_1 dominiert, dort kann er ggf. wie in Abschnitt 2.4.1 beschrieben mit seinem Vorgänger zusammengefügt werden. Abbildung 6.1.5 zeigt die Auslagerung von gemeinsamen Teilausdrücken in einen neuen Block. Die Knoten, die zur Menge N_n gehören, sind dunkelgrau. Sie entstehen erst bei der Auftrennung der Kanten in 6.1.5d und 6.1.5e. Wie in 6.1.5e und f zu erkennen ist, können

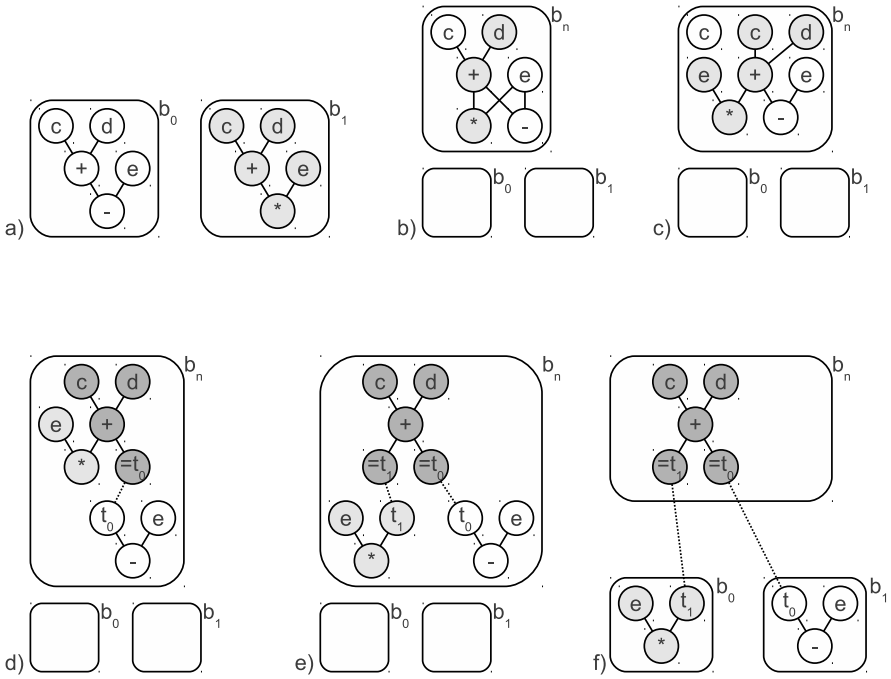


Abbildung 6.1.5: Schritte der GCSE ohne Dominierung

mehrere Ausgabeknoten entstehen, die denselben Vorgänger referenzieren. Um den Graphen zu vereinfachen, können diese in weiteren Optimierungsschritten zusammengefasst werden.

Die globale Eliminierung gemeinsamer Teilausdrücke lohnt sich nur, wenn sich die Laufzeit oder der Ressourcenverbrauch verringert. Dies kann durch Schedulingalgorithmen bzw. durch ein Abschätzen des Ressourcenverbrauchs herausgefunden werden. Da oftmals zusätzliche Zwischenergebnisse gehalten werden müssen, steigt die Anzahl der benötigten Register.

Verschieben schleifen-invarianter Knoten

Das in diesem Projekt implementierte Verschieben schleifen-invarianter Knoten ist abgeleitet von der in Abschnitt 2.4.1 beschriebenen Loop-Invariant-Code-Motion. Das hier verwendete Verfahren dient zum Auslagern von Knoten innerhalb des Schleifenrumpfes in einen Grundblock vor dem Schleifenkopf, der als *Pre-header* bezeichnet wird. Statt invariante Ausdrücke über ihre definierten und gelesenen Variablen sowie die in der Lebenszeitanalyse ermittelten *Live-In-* und *Live-Out*-Mengen zu finden, werden entsprechende Knoten im Datenflussgraphen anhand ihrer Kanten und E_{DD}-Kanten gesucht. Zur Vorbereitung müssen alle trivialen Operationen, d.h. Variablen- und Konstantenleseoperationen, die mehrere Nachfolger haben, aufgespalten werden, so dass diese jeweils nur einen Nachfolger besitzen (Abb. 6.1.6). Dadurch können nachher mehr Knoten als invariant markiert und somit ausgelagert werden.

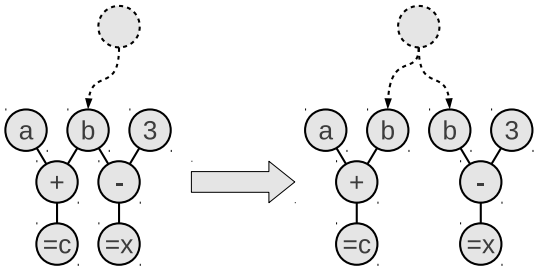


Abbildung 6.1.6: Duplizierung und Separierung der Variablen- und Konstantenleseoperationen. Auf diese Weise erhält man getrennte Graphen, was ggf. die Anzahl der invarianten Knoten erhöht. E_{DD}-Kanten, die durch gestrichelte Linien dargestellt sind, werden ebenfalls dupliziert.

Alle Schleifentupel L im CDFG (Abschnitt 5.2) werden in einer Liste gespeichert und so sortiert, dass bei geschachtelten Schleifen die innere vor den äußeren in der Liste steht. Anschließend werden alle Blöcke, die zu einer Schleife l gehören, in einer Menge BB_l zusammengefasst. Die Verschiebung der Knoten wird einzeln für jede Schleife unter Berücksichtigung der Reihenfolge durchgeführt. Auf diese Weise werden Knoten, die in mehreren Schleifen invariant sind, auch über mehrere Schleifengrenzen hinweg nach außen verschoben.

Zunächst werden alle Knoten innerhalb einer Schleife l als invariant angenommen und in einer Menge $N_{l,inv}$ zusammengefasst. Ein Knoten n wird aus $N_{l,inv}$ herausgenommen, wenn:

1. $\forall n \in N_{in}$
   $\exists n_e \in E_{dd}(n)$
   $BB(n_e) \in BB_l \wedge (ID(BB(n_e)) \geq ID(BB(n)) \vee n_e \notin N_{inv})$

2. $\exists n_p \in Prev(n)$
   $n_p \notin N_{inv}$

3. $\forall n \in N_{out}$
   $\exists n_e \in E_{DD}(n)$
   $\exists n_{ee} \in E_{DD}(n_e)$
   $BB(n_{ee}) \in BB_l \wedge n_{ee} \notin N_{inv}$

4. n hat eine Operation mit Seiteneffekten oder eine Operation, die veränderbare Daten liest, wie z.B. ein gemeinsamer Datenbereich oder ein Stream.

Aus der Menge $N_{l,inv}$ sind so lange Knoten zu entnehmen, bis sie sich nicht mehr verändert. 1 bis 3 decken die in Abschnitt 2.4.1 erläuterten Bedingungen für schleifen-invariante Ausdrücke 1 bis 3 ab. Der Unterschied hier ist, dass Knoten in einem Graphen analysiert werden und keine Ausdrücke mit Variablen. Bedingung 1 besagt, dass ein Eingabeknoten nicht von einem Ausgabeknoten lesen darf, der sich innerhalb der Schleife befindet ($BB(n_e) \in BB_l$) und dabei invariant ist ($n_e \notin N_{inv}$) oder zu einem späteren Zeitpunkt

ausgeführt wird ($ID(BB(n_e)) \geq ID(BB(n))$). In diesem Fall liest er Daten, die in der vorherigen Wiederholung modifiziert worden sind. Die zweite Bedingung besagt, dass von einem Knoten auch alle Vorgänger invariant sein müssen und die dritte Bedingung, dass alle Ausgabeknoten, die indirekt über zwei E_{DD}-Kanten mit einem Ausgabeknoten verbunden sind und sich innerhalb der Schleife befinden, ebenfalls invariant sein müssen.

Die in Abschnitt 2.4.1 erwähnte Voraussetzung für die Auslagerung, dass die Variable in der Schleife nur einmal definiert werden (Nummer 2) und nicht hinter dem Pre-header lebendig sein darf, muss hier nicht überprüft werden. Die Bedingung für die Dominanz (Nummer 1) muss hier allerdings auch erfüllt sein, damit ein Knoten auch ausgelagert werden darf. Daher werden zum Schluss alle Knoten aus $N_{l,inv}$ entfernt, die diese Bedingung nicht erfüllen.

An den Kanten, wo der eine Knoten in $N_{l,inv}$ integriert ist und der andere nicht, wird entsprechend ein Eingabe- und ein Ausgabeknoten eingefügt, die durch eine E_{DD}-Kante miteinander verbunden sind. Die Knoten in $N_{l,inv}$ werden zusammen mit den neu erzeugten Ausgabeknoten in einen neuen Block vor den Schleifenkopf verschoben. Nach dem Verlagern müssen die Konstanten- und Variablen-Leseknoten wieder zusammengefasst werden. Einzelne triviale Operationen wie Leseknoten sollten nicht verschoben werden, da durch das Verschieben wieder ein Ausgabe- und ein Eingabeknoten entsteht. Abb. 6.1.7 zeigt ein Beispiel, in dem alle invarianten Knoten dunkel markiert sind. Obwohl hier die Variable `z` mehrmals definiert wird, kann der erste Ausdruck ausgelagert werden. "`=t`" ist der künstlich erzeugte Ausgabeknoten mit der entsprechenden E_{DD}-Kante. Da sein Vorgänger "`=z`" ebenfalls ein Schreibknoten ist, hätte auch dieser für die E_{DD}-Kante verwendet werden können.

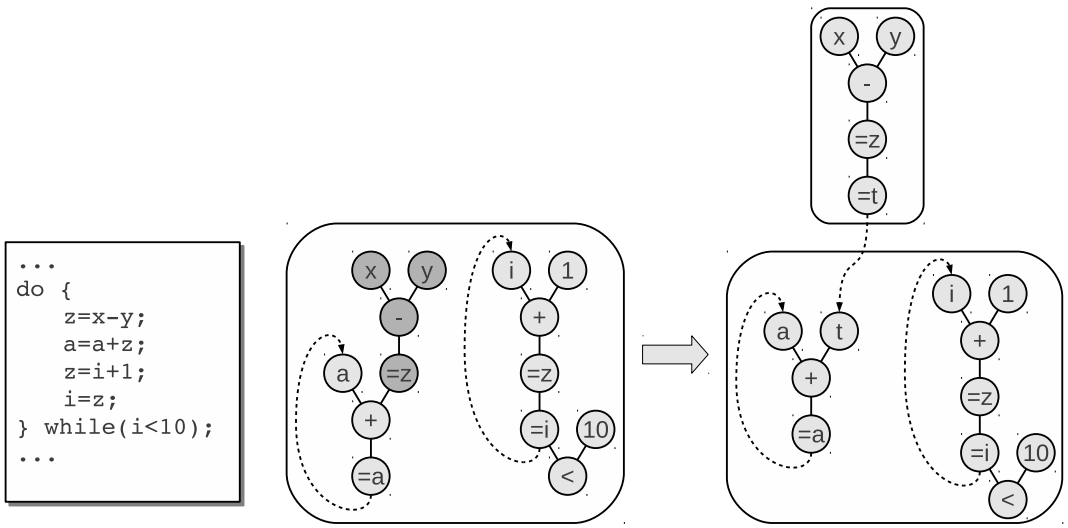


Abbildung 6.1.7: Schleifen-invariante Knoten und dessen Auslagerung in einen Pre-header

Umwandlung von Kontrollfluss-Verzweigungen in Datenfluss-Multiplexer

Die Umwandlung von Kontrollfluss in Datenfluss-Multiplexer funktioniert wie die im Abschnitt 2.4.3 beschriebene spekulative Berechnung. Dort werden einfache Verzweigungen, die durch *if*-Konstrukte entstehen, eliminiert und durch Selektoranweisungen ersetzt. Dadurch vereinfacht sich der Kontrollfluss und es entstehen zusätzliche Operationen im Datenfluss. Auf dem hier verwendeten Zwischencode können aufgrund der Bedingungsoperanden bei Speicherzugriffen (Abschnitt 5.4) auch Speicheroperationen integriert werden, was sonst aufgrund der Seiteneffekte sonst nicht möglich wäre. Auf diese Weise kann diese Optimierung nicht nur bei einfachen Berechnungen und Zuweisungen, sondern auch bei komplexeren Operationen angewendet werden.

6.2 Partielle Evaluierung

Partielle Auswertung (PE) ist die Berechnung einer Funktion, wobei lediglich ein Teil ihrer Eingangsdaten bekannt ist [33]. Die bekannten Eingangsdaten werden als statisch, die unbekannten als dynamisch bezeichnet. Mit Hilfe der statischen Eingangsdaten wird eine neue Funktion erzeugt (Residuum), die weniger Argumente als die Originalfunktion benötigt. Nur die Information, die nicht bei der partiellen Auswertung bekannt war, muss der neuen Funktion übergeben werden. Partielle Auswertung wird daher auch als Programmspezialisierung bezeichnet. Die Vorteile der Programmspezialisierung liegen in der Reduzierung der Laufzeit, da ein Teil der Ausdrücke schon zur Kompilierzeit berechnet werden kann. Nach der Notation, die u.a in [76] benutzt wird, erzeugt ein Programm s in der Sprache L während der Ausführung das Resultat res aus seinen Eingangsdaten d:

$$||s||_L[d] = res.$$

Ein Programmspezialisierer *peval* evaluiert s zusammen mit dem statischen Anteil der Eingangsdaten d_s, der dynamische Teil d_d wird nicht genutzt. Das Resultat s' ist die spezialisierte Version von s und muss das gleiche Resultat wie s für beliebige Eingangsdaten d_d erzeugen:

$$||peval||_L[s, d_s] = s'$$
$$||s'||_L[d_d] = ||s||_L[d_s, d_d].$$

Es gibt verschiedene Vorgehensweisen zur Erzeugung des Residuums, die z.B. in [76] oder [58] beschrieben sind. Die Methoden gehen von einem Programm aus, dessen Kontrollfluss aus Grundblöcken besteht, die durch Kontrollkanten miteinander verbunden sind. Das Programm wird ausgeführt und alle Ausdrücke, die auf statischen Variablen basieren, ausgewertet. Datenzustände werden bei der Spezialisierung umgewandelt in Zustände im Kontrollfluss. Bei der polyvarianten Spezialisierung etwa erhält man von jedem Grundblock beliebig viele neue Varianten, je nachdem, welche Werte die statischen Variablen beim Eintritt in den Block haben. Das neue Programm besitzt gewöhnlich viele triviale Zustandsübergänge im Kontrollfluss, die zusammengefasst werden können (*transition com-*

pressing). Der Vorteil der Programmspezialisierung liegt bei der Verkürzung der Laufzeit, da Ausdrücke und Sprünge schon zur Kompilierzeit berechnet werden. Nachteile gibt es bei großen Programmen mit vielen statischen Daten, da hier die Zahl der spezialisierten Grundblöcke sehr groß werden kann und die Residualprogramme dementsprechend lang werden. Wenn Sprungbedingungen von dynamischen Daten abhängen, halten sich die Optimierungsmöglichkeiten stark in Grenzen [76].

Neben der direkten Laufzeitverkürzung eröffnet die partielle Auswertung weitere Optimierungsmöglichkeiten für imperative Programme. In [25] etwa werden Methoden vorgestellt für Loop-Invariant Code Motion, Strength Reduction und weitere spezielle Optimierungen, die auf der partiellen Evaluierung beruhen. Auch in der Hardwaresynthese kommt die partielle Auswertung zum Einsatz. In [78] und [98] wird sie genutzt zur Reduzierung von konstanten Signalen an Schaltungskomponenten oder in [103] zur Generierung von Konstantenmultiplizierern.

In dieser Arbeit wird eine modifizierte Form der partiellen Auswertung genutzt, die nicht den Kontrollfluss modifiziert, sondern das Programm statisch analysiert und Zwischenergebnisse sammelt. Diese Ergebnisse werden genutzt, um FPGA-spezifische Optimierungen während der Architektursynthese in den Operationen für den Kontrollpfad durchzuführen. Zu den Optimierungen gehören die Reduktion der Datenbreite, die Erkennung von Konstanten sowie die Parallelisierung oder Eliminierung von Speicherzugriffen. [15] beschreibt einen anderen Algorithmus, um Speicherzugriffe zu parallelisieren. Im Gegensatz zu den hier vorgestellten Methoden basiert dieser auf Profiling oder der Analyse von Induktionsvariablen. Programme für die Architektursynthese besitzen häufig zyklische Prozesse, die mit Hilfe von Endlosschleifen dargestellt werden [73]. Werden Endlosschleifen gefunden, terminiert der Algorithmus nicht vorzeitig [38], sondern analysiert auch diese Schleifen so weit wie möglich.

Die hier verwendete partielle Evaluierung soll von der Annahme profitieren, dass in einem Programm die Berechnungen für die Steuerdaten - im Vergleich zu denen der Nutzerdaten - nicht von Eingabedaten abhängen und somit im Voraus bestimmbar sind. Die folgenden Formeln zeigen die Vorgehensweise der Algorithmen:

$$||peval||_L[s, d_s] = V \tag{6.2.1}$$

$$||opt||_L[s, V] = s' \tag{6.2.2}$$

$$||s||_L[d_s, d_d] = ||s'||_L[d_d]. \tag{6.2.3}$$

Der Hauptunterschied zu den Methoden, die in [76] oder [58] beschrieben sind, liegt darin, dass das Programm s erst einmal unverändert bleibt, die einzelnen Grundblöcke nicht spezialisiert werden und von ihnen nicht mehrere Varianten entstehen. Vielmehr wird s, das in Form des vorher beschriebenen CDFGs vorliegt, ausgeführt und alle Zwischenergebnisse der DFG-Knoten in der Datenstruktur V gesammelt (Formel 6.2.1). Sind Werte abhängig von erst später bekannten Eingabedaten der Funktion und somit nicht berechenbar, erhalten sie die Markierung ungültig. Berechenbar sind vor allem Schleifenzähler, Arrayadressen usw., da die Variablen, von denen sie abhängen, oft in Form von Konstanten an die Funk-

tion übergeben werden. Daher sind konstante Funktionsargumente mit zu berücksichtigen und wie Initialwerte für Argumente zu behandeln. Nach der Terminierung der partiellen Auswertung liegt für jeden berechenbaren Datenflussknoten eine Liste mit Werten vor, die die Zwischenergebnisse von jedem Zeitpunkt beinhaltet, an dem der Knoten ausgeführt wurde. Dies ist der Hauptunterschied zu polyvarianten Verfahren, wo Zwischenergebnisse zu verschiedenen Zeitpunkten implizit durch spezialisierte Blöcke dargestellt sind. In späteren Schritten werden die Wertelisten von Optimierungen *opt* genutzt, um das Programm zu modifizieren (Formel 6.2.2).

6.2.1 Algorithmus

Bei der hier entwickelten Variante der partiellen Evaluierung wird ein Tupel von Werten $V_n = (v_0, \ldots, v_L)$ für jeden Knoten n innerhalb einer aufgerufenen Funktion erzeugt. Jedesmal, wenn n ausgeführt wird, wird der berechnete Wert v an V_n angehängt. Wenn n nicht berechnet werden kann, wird v als ungültig markiert. Die partielle Evaluierung wird auf dem CDFG vom Startblock bb_{Start} bis zum Endblock bb_{End} angewendet. Jeder traversierte Grundblock wird an ein Tupel mit dem Namen BB_{Trav} angehängt.

Am Anfang ist V_n leer für jeden Knoten n. Wenn Funktionsargumente $n \in N_{arg}$ mit bekannten Initialwerten vorliegen, werden diese vor der Ausführung V_n zugefügt. Anschließend wird der eigentliche Algorithmus *evalBlocks* (Alg. 7) ausgeführt. Hierbei werden alle Blöcke entlang des Kontrollflusses von bb bis einschließlich bb_{Last} evaluiert. Die Menge N_{Ignore} in Alg. 7 beinhaltet Knoten, die nicht ausgewertet werden dürfen. Beim erstmaligen Aufruf ist $bb = bb_{Start}$ und $bb_{Last} = bb_{End}$, die Menge N_{Ignore} ist leer und wird während der Ausführung des Algorithmus mit Knoten gefüllt. Zuerst berechnet *evalBlocks* für jeden Knoten n in bb seinen aktuellen Wert mit der Funktion *calcDFG* und fügt ihn an V_n an. Dabei geht *calcDFG* wie folgt vor: Wenn $n \in N_{Ignore}$, wird n nicht berechnet und v ist ungültig. Wenn n Vorgänger besitzt, wird der Wert rekursiv auf Grundlage seiner Operanden berechnet. Wenn $n \in N_{arg}$, dann ist v äquivalent zum letzten Wert von V_n. Bei Eingabeknoten ($n \in N_{in}$) wird v von einem adäquaten Ausgabeknoten genommen, der von einer E_{DD}-Kante referenziert wird. Wenn n mehrere Ausgabeknoten referenziert, wird der Knoten genommen, der sich in einem Grundblock möglichst weit hinten von BB_{Trav} befindet. Damit wird sichergestellt, dass es sich hier um den aktuellen Ausgabeknoten handelt.

Nach der Berechnung aller Knoten wird bb an BB_{Trav} angefügt und der Nachfolgeblock s ermittelt. Wenn der Wert $values_L$ der Sprungbedingung n_{Cond} ungültig ist, wird s mit *evalBranches* bestimmt. Wenn die Kante (bb, s) Rückwärtskante einer Endlosschleife ist, wird *evalInfLoop* aufgerufen und anschließend der aktuelle Aufruf von *evalBlocks* beendet, ansonsten wird s ausgewertet wie zuvor bb. Wenn die Evaluierung z.B. aufgrund nicht detektierter Endlosschleifen oder sehr großer Iterationsfolgen von Schleifen nicht terminiert, kann der Algorithmus abgebrochen werden, wenn eine bestimmte Zahl von traversierten Grundblöcken überschritten wird oder *evalBlocks* eine gewisse Rekursionstiefe

Algorithm 7: **evalBlocks**: Wertet alle Blöcke von bb bis bb_{Last} aus

```
Data: bb, bb_Last, includeBB_Last, N_Ignore
Result: N_Ignore
begin
    while bb ≠ bb_End ∧ (bb ≠ bb_Last ∨ includeBB_Last) do
        calcDFG(bb, N_Ignore)
        Append bb to BB_Trav
        if bb = bb_Last then
            return N_Ignore
        if |successors(bb)| > 1 ∧ v_L^cond is invalid then
            (s, N_Ignore) ← evalBranches(bb, N_Ignore)
        else
            take an adequate successor s
        if (bb, s) ∈ L_Inf then
            evalInfLoop(s, bb)
            return N_Ignore
        bb ← s
    return N_Ignore
end
```

überschritten hat. In diesem Fall ist V_n zurückzusetzen und die Werte aller Knoten sind unbekannt.

evalBranches evaluiert Verzweigungen mit unbekannten Sprungbedingungen in Block bb, wobei jeder Zweig einzeln ausgewertet wird. Dazu wird *evalBlocks* für jeden Zweig ausgeführt. Da unbekannt ist, welcher Zweig tatsächlich genommen wird, müssen alle Eingabeknoten außerhalb der Zweige, die von Ausgabeknoten innerhalb der Zweige lesen, in die Menge der ungültigen Knoten N_{Ignore} aufgenommen werden. *evalBranches* gibt den Knoten zurück, an dem sich die Verzweigungen wieder vereinigen sowie die aktualisierte Menge N_{Ignore}.

evalInfLoop (Alg. 9) ermöglicht die partielle Evaluierung von Programmen mit Endlosschleifen. Zwischenergebnisse innerhalb solcher Schleifen werden für eine Iteration gesammelt, während Resultate, die von mehreren Wiederholungen abhängen, verworfen werden. *evalInfLoop* ruft *evalBlocks* mit dem Start- und dem Endblock der Schleife auf, so dass der CFG einmal traversiert wird. Vorher müssen alle Eingabeknoten innerhalb der Schleife, die Ausgabeknoten außerhalb der Schleife referenzieren, in N_{Ignore} eingefügt werden. Auf diese Weise werden nur Knoten ausgewertet, die Werte referenzieren, welche innerhalb der Endlosschleife initialisiert worden sind.

Die partielle Evaluierung kann auch Programme auswerten, die nicht nach den Kriterien der strukturierten Programmierung entstanden sind [23]. Solche Programme besitzen Schleifen mit mehreren Ausgängen oder Sprünge in Schleifenkörper bzw. beliebig miteinander verknüpfte Grundblöcke. Ab einer bestimmten Komplexität steigt bei solchen Programmen die Rekursionstiefe von *evalBlocks* sehr schnell an, wodurch der Algorithmus abgebrochen wird. Allerdings erhält man in diesem Fall keine auswertbaren Ergebnisse.

Algorithm 8: **evalBranches**: Wertet Verzweigungen in *bb* aus mit unbekannter Verzweigungsabedingung

Data: bb: branch-block, N_{Ignore}: Set of nodes to ignore
Result: bb_{Merge}: merge-block, N_{Ignore}: Updated set of nodes to ignore
begin
  Find the first block bb_{Merge} which merges the branches of bb. Backward edges, i.e., loops are also taken into account.
  If there is no block, then $bb_{Merge} = bb_{End}$
  foreach $s \in successors(bb) \wedge s \neq bb_{Merge} \wedge s \neq bb_{End}$ **do**
    $N_{Ignore} \leftarrow evalBlocks(s, bb_{Merge}, false, N_{Ignore})$
    Determine the set of all blocks BB_P, which reside on a path between s and bb_{Merge}
    $BB_P \leftarrow BB_P \cup \{s\}$
    Determine the set of all nodes N_P which are located in BB_P
    foreach n_P *in* $N_P \wedge n_P \in N_{out}$ **do**
      foreach $n_{dd} \in N_{DD}(n_P) \wedge Block(n_{dd}) \notin BB_P$ **do**
        $N_{Ignore} \leftarrow N_{Ignore} \cup \{n_{dd}\}$
    return (bb_{Merge}, N_{Ignore})
end

Algorithm 9: **evalInfLoop**: Evaluiert Endlosschleifen

Data: bb_S, bb_E: Start- and end-block of the loop
Result:
begin
  Determine the set of all blocks BB_L, which reside on a path between bb_S and bb_E
  $BB_L \leftarrow BB_L \cup \{bb_S, bb_E\}$
  $N_{unknown} \leftarrow \emptyset$
  foreach $n_L \in BB_L \wedge n_L \in N_{in}$ **do**
    if $n_{DD} \in N_{DD}(n_L) \wedge Block(n_{DD}) \notin BB_L$ **then**
      $N_{unknown} \leftarrow N_{unknown} \cup \{n_L\}$;
  $evalBlocks(bb_S, bb_E, true, N_{unknown})$;
end

Im Folgenden werden die oben beschriebenen Algorithmen auf dem CDFG von Abb. 5.1.1 angewendet. Zuerst berechnet *evalBlocks* die Knoten von bb_0. Da die Sprungbedingung unbekannt ist, wird *evalBranches* aufgerufen, welches versucht, den Vereinigungsblock zu bestimmen. Der Vereinigungsblock ist der Endblock bb_{End} des CDFG, da die beiden Zweige nicht mehr zusammentreffen. Anschließend wird *evalBlocks* rekursiv aufgerufen für bb_1 und bb_{End} sowie für bb_2 und bb_{End}. Der erste rekursive Aufruf von *evalBlocks* evaluiert bb_1 und erkennt eine Endlosschleife, so dass *evalInfLoops* ausgeführt wird. *evalInfLoop* fügt Knoten $x2$ in die Menge der unbekannten Knoten ein, da dieser Knoten $x0$ außerhalb der Schleife referenziert. Danach wird *evalBlocks* erneut aufgerufen und wertet bb_1 aus. Im Vergleich zur ersten Evaluierung von bb_1 sind einige Werte von $x2$ und von allen seinen Nachfolgern jetzt ungültig. Der zweite rekursive Aufruf von

evalBlocks evaluiert bb_2. Weil bb_2 ebenfalls eine unbekannte Bedingung beinhaltet, wird *evalBranches* und somit *evalBlocks* erneut für bb_3 bis bb_5 sowie bb_4 bis bb_5 aufgerufen. Der Vereinigungsblock bb_5 darf nicht in der Auswertung eingeschlossen werden. Anschließend aktualisiert *evalBranches* die Menge der zu ignorierenden Knoten N_{Ignore} und gibt bb_5 zurück. Da $x9$ einen Knoten innerhalb der Zweige referenziert, wird dieser in N_{Ignore} mit aufgenommen. Im letzten Schritt wird bb_5 evaluiert und alle rekursiven Aufrufe von *evalBlocks* terminieren.

6.2.2 Optimierungen

Die bei der Evaluierung entstanden Tupel V_n von Datenflussknoten n können für diverse Optimierungen genutzt werden, wenn diese gültige Werte besitzen. In dieser Arbeit wurden vier Typen wie die Detektierung von konstanten Knoten, die Reduktion der Datenpfadbreiten sowie die Parallelisierung und die Eliminierung von Arrayzugriffen implementiert. Diese Optimierungen können auch von anderen Methoden wie Konstantenpropagation (Abschnitt 6.1) oder abstrakter Interpretation [16] erreicht werden, die auf Knoten mit unbekannten Werten funktionieren. Wenn allerdings Werte von Knoten vollständig von der partiellen Evaluierung bestimmt werden können, so sind die Resultate zumindest gleichwertig, da bereits bei der Kompilierung die realen Daten zur Verfügung stehen, die auch bei der späteren Ausführung vorhanden sind.

Ein DFG-Knoten n ist konstant, wenn alle Werte v_i in V_n gültig sind und $v_0 = v_i,\ \forall\, i \in \{0, \ldots, |V_n|-1\}$. Die Operation wird durch die entsprechende Konstantenlese-Operation ersetzt und Vorgängerknoten können eliminiert werden. Wenn Sprungbedingungen konstant sind, kann der Kontrollfluss durch die Entfernung der Verzweigungsbedingungen stark vereinfacht werden (Abschnitt 2.4.1).

Eine andere Möglichkeit ist die Reduzierung der Datenpfadbreiten, wenn ausgehende Kanten und Knoten auf Busse und Funktionseinheiten in FPGAs abgebildet werden. Auf diese Weise kann der Ressourcenverbrauch stark gesenkt werden, da nur wirklich benötigte Bits berechnet und gespeichert werden müssen und nicht die gesamte Breite des vom Programmierer angegebenen Datentyps. Die minimale Breite des Datenpfades $w(n)$ für einen Knoten n ist gegeben durch $w(n) = \max\{w(v_i) \mid v_i \in V_n\}$ für alle i, wobei $w(v_i)$ die minimale Datenpfadbreite des Wertes $v_i \in V_n$ ist.

Normalerweise können RaW- oder WaW-Operationen bei Arrays nur sequentiell ausgeführt werden, da sich die zweite Operation auf das durch die erste Operation veränderte Array bezieht, wie in Abbildung 5.4.2a dargestellt ist. Um zwei Operationen n_1 und n_2 parallelisieren zu können, müssen die korrespondierenden Indexknoten k_1 und k_2 gültige und unterschiedliche Werte zu jedem Zeitpunkt besitzen, was der Fall ist, wenn $v_i \neq \tilde{v}_i$ mit $v_i \in V_{k1}$ und $\tilde{v}_i \in V_{k2}$ für alle i. Nach der Parallelisierung referenzieren die Operationen denselben primären Knoten, wie in Abb. 5.4.3b. gezeigt wird. Durch die parallele Darstellung im Zwischencode können dedizierte Speicher innerhalb FPGAs ausgenutzt werden, die standardmäßig mehrere Schreib- und Leseports zur Verfügung stellen.

Durch die Parallelisierung von Arrayoperationen erhält man wiederum mehr Möglichkeiten bei der Eliminierung von Knoten. Bei RaW-, WaW- oder RaR-Operationen kann die zweite Operation n_2 gelöscht werden, wenn beide Operationen immer den gleichen Arrayindex haben. In diesem Fall wird ein Wert in ein Array geschrieben und wieder gelesen oder auf dieselbe Position wird zweimal zugegriffen, ohne dass das Array zwischenzeitlich dort modifiziert wurde. Dies ist zum einen gegeben, wenn sich im DFG beide Operationen n_1 und n_2 auf denselben Indexknoten beziehen, also $k_1 = k_2$. Wenn $k_1 \neq k_2$ kann n_2 dennoch gelöscht werden, wenn $v_i^{k1} = v_i^{k2}$ mit $v_i^{k1} \in V_{k1}$ und $v_i^{k2} \in V_{k2}$. Wenn die zweite Arrayoperation n_2 den sekundären Knoten von mehreren parallelen Schreiboperationen $(n_{11}, n_{12}, ..., n_{1m})$ referenziert, kann n_2 gelöscht werden, wenn für alle i der Indexwert $v_i^{k2} \in \{v_i^{k11}, \ldots, v_i^{k1m}\}$ ist.

6.3 Funktionales Pipelining

Funktionales Pipelining ist mit dem Software-Pipelining verwandt, das bei Mikroprozessoren verwendet wird. Software-Pipelining ist eine Art der Out-of-order Execution, wo Maschinenbefehle umsortiert und in einer anderen Reihenfolge als im Assemblercode ausgeführt werden, um die Funktionseinheiten besser auszulasten. Voraussetzung dafür ist das Vorhandensein der entsprechenden Anzahl an Ausführungseinheiten und die Möglichkeit, gleichzeitig mehrere Befehle ausführen zu können, was z.B. bei VLIW-Architekturen gegeben ist.

Beim funktionalen Pipelining wird der Kontroll-Datenflussgraph von sich wiederholenden Programmen in Subgraphen unterteilt, die gleichzeitig ausgeführt werden. In dieser Arbeit wird funktionales Pipelining auf den DFG von Schleifenkörpern ohne Kontrollfluss angewendet und die darin enthaltenen Instruktionen so umgeordnet, dass diese mit einem höheren Parallelitätsgrad ausgeführt werden können, wodurch sich die Laufzeit reduziert. Neue Iterationsvorgänge der Schleife werden gestartet, bevor die vorherigen abgeschlossen sind, so dass mehrere Iterationen in unterschiedlichen Berechnungsstufen gleichzeitig aktiv sind.

Abbildung 6.3.1a zeigt eine Funktion für die Berechnung des Skalarprodukts. Wenn man den Schleifenzähler vernachlässigt, dauert eine Schleifeniteration drei Zeitschritte unter der Annahme, dass die Zugriffe auf `a` und `b` gleichzeitig in einem Zeitschritt ausgeführt werden können und die Multiplikation sowie die Akkumulation ebenfalls einen Zeitschritt dauert. Hier bestehen Datenabhängigkeiten zwischen den Operationen, was durch die schwarzen Pfeile dargestellt wird. Eine Anweisung benötigt die Resultate des Vorgängers, eine parallele Ausführung ist nicht möglich und die Funktionseinheiten werden nur jeden dritten Takt benutzt. Die grau gestrichelten Pfeile zeigen die Abhängigkeiten der Ergebnisse aus der vorherigen Schleifeniteration. Bis auf die letzte Anweisung bestehen keine Datenabhängigkeiten, die zwischen den Operationen der einzelnen Schleifeniterationen übergreifen. Daher können bei Durchführung der Multiplikation schon die nächsten Operanden aus den Arrays geholt werden. Auch bei der Akkumulation kann die nächste Multiplikation durchgeführt

a)

```
int dot(int a[], int b[], int n) {
    int i, res = 0;
    int x1,x2,y;
    for(i=0; i<n; ++i) {
        x1 = a[i]; x2 = b[i];
        y = x1 * x2;
        res = res + y;
    }
    return res;
}
```

b)

```
int dot(int a[], int b[], int n) {
    int i, res = 0;
    int x1,x2,y;

    x1 = a[0]; x2 = b[0];

    y = x1 * x2;
    x1 = a[1]; x2 = b[1];

    for(i=2; i<n; ++i) {
        res = res + y;
        y = x1 * x2;
        x1 = a[i]; x2 = b[i];
    }

    res = res + y;
    y = x1 * x2;

    res = res + y;

    return res;
}
```

Abbildung 6.3.1: Quellcode für die Berechnung des Skalarprodukts vor und nach dem Software-Pipelining

und die übernächsten Operanden geholt werden, wie in Abb. 6.3.1b gezeigt wird. Auf diese Weise gibt es keine Datenabhängigkeiten mehr zwischen den einzelnen Operationen und eine parallele Ausführung des gesamten Schleifenrumpfes ist möglich. Da im ersten Zeitschritt noch keine Daten zur Multiplikation und zur Akkumulation vorliegen, wird vor die Schleife ein Prolog gesetzt. Hier werden im ersten Zeitschritt die Operanden geholt und im zweiten Zeitschritt zusätzlich multipliziert. Somit kann die Schleife mit einem höheren Anfangsindex ausgeführt werden. Nach Ausführung der Schleife müssen keine Operanden mehr geholt werden, allerdings noch die restlichen Daten multipliziert und akkumuliert, weshalb hinter der Schleife ein Epilog ausgeführt wird. Verfahren zum Software-Pipelining sind u.a. in [65] oder [8] beschrieben.

Anhand der E_{DD}-Kanten des in Kapitel 5 vorgestellten Zwischencodes kann auf einfache Weise festgestellt werden, welche Operationen innerhalb eines Schleifenrumpfes parallelisiert werden können und welche nicht. Eine sequentielle Folge von Operationen kann immer parallel ausgeführt werden, sofern keine E_{DD}-Kanten innerhalb dieser Folge verlaufen. Die Parallelisierung setzt auch entsprechende Operationen vor und nach der Schleife, also im Epilog und im Prolog voraus. Der Schleifenrumpf selber darf keinen Kontrollfluss enthalten, muss also aus nur einem Grundblock bestehen. Enthält der Schleifenrumpf *if*-Anweisungen, so können diese in der Regel mit Hilfe der in Abschnitt 6.1 vorgestellten spekulativen Berechnung eliminiert werden.

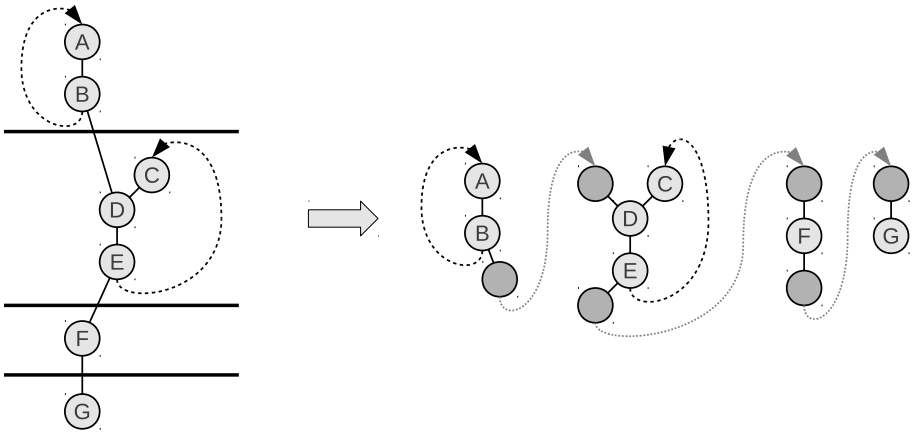


Abbildung 6.3.2: Unterteilung eines DFGs in Subgraphen, die parallel ausgeführt werden

Abb. 6.3.2 zeigt einen Datenflussgraphen mit E_{DD}-Kanten. Kanten, die keine durch E_{DD}-Kanten eingefasste Knotenfolge verbinden, können für eine parallele Ausführung aufgetrennt werden. Nach der Auftrennung müssen entsprechende Ein- und Ausgabeknoten an die Schnittstellen gesetzt und durch neue E_{DD}-Kanten verbunden werden. Diese E_{DD}-Kanten stellen den Datenfluss zwischen den einzelnen Schleifeniterationen dar, die parallelisierten Operationen beziehen sich somit auf die Ergebnisse der vorherigen Iteration. Hier ist lediglich der neu entstandene Schleifenkörper dargestellt, der Pro- und der Epilog wurde weggelassen.

Bei der hier verwendeten Implementierung muss für den DFG die Ablaufplanung bereits durchgeführt worden sein, bevor die Kanten aufgetrennt werden können. Dies ist ähnlich zu den in [7] beschriebenen Verfahren, wo ebenfalls angenommen wird, dass die Ablaufpläne bereits vorhanden sind und nicht mehr geändert werden können. Die Auftrennung des DFGs kann nur zwischen Operationen geschehen, die verschiedenen Kontrollschritten zugewiesen worden sind, was die mögliche Parallelisierung stark einschränkt. Die Operationen sind u.a. so anzuordnen, dass sich die rückwärts-gerichteten E_{DD}-Kanten nicht überschneiden. In 6.3.2 können (A, B) und (C, D, E) etwa nicht mehr parallelisiert werden, wenn B denselben Kontrollschritt wie C erhielte. Ein Nachteil des funktionales Pipelinings ist daher auch die Komplexität. Die Erstellung eines optimalen Ablaufplans ist NP-vollständig, weshalb auf Heuristiken zurückgegriffen wird. In dieser Arbeit wird Simulated-Annealing[62] verwendet. Benchmarks in Abschnitt 8.1.3 zeigen die Auswirkungen des Pipelinings auf verschiedene Testprogramme.

Kapitel 7

VHDL-Generierung

Der in Kapitel 5 beschriebene Zwischencode wird in RTL-VHDL-Code übersetzt. Die angewandten Synthesetechniken sind bis auf Prozessaufrufe und der Interprozesskommunikation ähnlich zu bestehenden Verfahren, die z.B. in [24] beschrieben sind und auch in Systemen wie Spark[44] oder Hybridthreads[95] eingesetzt werden. Die folgenden Abschnitte beschreiben detailliert, in was für Hardwarestrukturen einzelne Konstrukte des Prozessmodells und des Zwischencodes umgesetzt werden. Alle Optimierungen und Transformationen müssen bereits auf dem Zwischencode durchgeführt worden sein, da der CDFG direkt umgesetzt wird und auch Konstrukte wie leere Blöcke oder Berechnungen mit ungenutzten Ergebnissen berücksichtigt werden.

7.1 Prozesse

Jeder Prozess und jede Funktion erhält eine eigene VHDL-Entität. Bei der Implementierung wird nicht zwischen Prozessen und Funktionen unterschieden. Wie erwähnt unterscheiden sich Funktionen lediglich beim Aufruf, weil der Programmablauf in der aufrufenden Instanz angehalten wird, bis die Funktion beendet ist. Aus diesem Grund wird nachfolgend nur noch von Prozessen gesprochen. Wenn ausschließlich Funktionen gemeint sind, wird dies explizit erwähnt. Da ein Prozess die Gesamtheit aller Operationen darstellt, die zu einem Kontrollfluss gehören, wird der Kontrollfluss in Form eines einzigen Automaten implementiert. Jeder Grundblock wird dabei auf mindestens einen Zustand abgebildet. Den Operationen im Datenpfad innerhalb eines Grundblocks werden mit Hilfe des List-Schedulings Kontrollschritte zugeordnet, so dass ein Block so viele Kontrollschritte und somit Zustände besitzt wie die Latenz des Ablaufplans hoch ist. Leere Blöcke erhalten auch einen Zustand.

Im Vergleich zu Systemen wie Spark[44] wird der Automat im VHDL-Code nicht mit zwei, sondern drei VHDL-Prozessen beschrieben. Ein Prozess ist getaktet und die anderen beiden laufen asynchron. Der getaktete Prozess ist verantwortlich für Berechnungen und Signalzuweisungen, die von Zuständen im Automaten abhängen und in Registern gehalten werden sollen. Der erste asynchrone Prozess wird für Zustandsübergänge verwendet, während der andere Berechnungen und Signalzuweisungen durchführt, wobei die erzeug-

```
process int myProc(int x, int y);
```

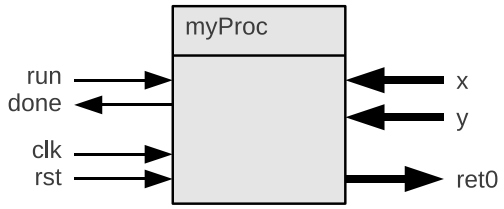


Abbildung 7.1.1: Prozessdeklaration und die erzeugte Hardwarekomponente mit Datenbussen, Takt und invertierten Reset-Signal sowie zusätzlichen Synchronisationssignalen `run` und `done`

ten Signale nicht in Registern gespeichert werden. Diese Signale können entweder nach außen geführt oder als Steuersignale für interne Komponenten verwendet werden. Außerdem dienen sie als Operanden für Berechnungen im getakteten Prozess, wodurch verkettete Operationen im selben Kontollschritt ausgeführt werden können (Chaining). Jedem Operationstypen wird u.a. abhängig von dessen Datenbreite eine bestimmte Latenz zugeordnet, die im Compiler gespeichert ist. Erst wenn die Ausführungsdauer von verketteten Operationen mit dem Hinzufügen einer weiteren Operation die Taktperiode übersteigt, wird die neue Operation in den nächsten Zeitschritt gelegt. (Sowohl die Latenz als auch die Zyklenzahl von Operationen und die Taktperiode sind anpassbar.)

Besitzen somit zwei voneinander abhängige Operationen denselben Kontrollschritt, sind sie direkt über Signale miteinander verbunden, bei verschiedenen Kontrollschritten wird das Ergebnis der ersten Operation in einem Register gehalten. In Spark wird Chaining über Variablen im getakteten FSM-Prozess gelöst. Allerdings können die Werte der Variablen dann nicht wie im ungetakteten Prozess direkt hinausgeführt und ohne Verzögerung in weiteren Komponenten innerhalb oder außerhalb der Entity genutzt werden.

Argumente des Prozesses sind Eingänge, während Rückgabewerte Ausgänge der Entity darstellen, wie in Abb. 7.1.1 gezeigt. Wenn Arrays dem Prozess übergeben werden, erhält die Entity Speicherbusse und die dazugehörigen Kontrollsignale in der Schnittstelle. Genau so verhält es sich bei asynchronen Streams, die Datensignale bei Eingabestreams sind Eingänge und Ausgänge bei Ausgabestreams. Synchrone Streams besitzen zusätzliche Steuersignale. Die Zurücksetzung des Automaten erfolgt durch ein *reset*-Signal. Nach dem *reset* befindet sich der Automat in einem Wartezustand, bis er durch einen entsprechenden Prozess- oder Funktionsaufruf aktiviert wird. Die Schnittstelle der Entity ist bereits vollständig durch den Prototypen des Prozesses definiert, d.h. alle Signale, die die Entity besitzt, sollen auch schon bei der Deklaration des Prozesses feststehen. Dies vereinfacht auch die Integration von bestehenden VHDL-Modulen. So kann die in Abb. 7.1.1 deklarierte Entity entweder durch TransC beschrieben worden sein oder es kann sich um handoptimierten VHDL-Code handeln. Für den Aufruf aus anderen TransC-Prozessen spielt dies keine Rolle, so lang der Prototyp bekannt ist.

7.2 Grundoperationen

Grundlegende arithmetische und logische Operationen wie z.B. die Ganzzahladdition oder Vergleiche auf Ganzzahlen werden direkt mit Standardoperatoren in VHDL dargestellt, die von VHDL direkt oder von der numerischen Standardbibliothek `NUMERIC_STD` [11] zur Verfügung gestellt werden. Eine Übersicht über die unterstützten Operatoren gibt u.a. [91]. Die Standardbibliothek kann sowohl vorzeichenbehaftete als auch nicht vorzeichenbehaftete Datentypen bearbeiten, wobei die Operanden nicht gemischt werden dürfen. Für jedes Resultat einer Operation muss die Breite des Datenbusses ermittelt werden, so dass diese mit der Breite der Operanden zusammenpasst, wie in Abschnitt 7.7 beschrieben. Vor der Implementierung der Operationen werden die grundlegenden Syntheseschritte wie die Allokation, Bindung und Ablaufplanung durchgeführt. Da die Operationen aber nicht in Form von Komponenten, sondern ebenfalls als VHDL-Operationen in die einzelnen Zustände der Automaten eingetragen werden, ist hier keine Bindung und keine Allokation notwendig, da diese auf Ebene der RTL-Synthese durchgeführt wird. Lediglich die Ablaufplanung ist relevant, da diese auch die Anzahl der Automatenzustände bestimmt und in welchen Zustand, d.h. in welchen Kontrollschritt, welche Operation ausgeführt wird. Ist bei einer Operation *op* mindestens eine Nachfolgeoperation im selben Kontrollschritt, so wird *op* im ungetakteten Prozess platziert (siehe Abschnitt 7.1), da es auf diese Weise direkt über ein Signal in die nächste Operation geleitet werden kann. Wenn das Ergebnis von *op* auch in späteren Kontrollschritten benötigt wird, so muss es in einem Register gehalten werden. Die Anweisung zur Speicherung im Register befindet sich im getakteten Prozess, wobei dann das Signal aus dem ungetakteten zusätzlich in ein Register geführt wird. Wird ein Ergebnis nur in späteren Kontrollschritten benötigt, so kann die Operation direkt im getakteten Prozess platziert werden. Anweisungsfolgen wie `a=b=c=d` werden in einem Takt abgearbeitet. Wird etwa das Ergebnis von `b` später nicht mehr benötigt, so erhält `b` auch kein Register. Für den Fall, dass `b` noch verwendet wird, wird zwar ein Register erzeugt, `a` erhält seinen Wert jedoch aus dem Bus vor dem Register, so dass für diese Zuweisung kein Taktzyklus verloren geht.

Für die Zwischenspeicherung von Signalen wird in TransC für jedes Ergebnis ein neues Register allokiert, bestehende Register werden nicht wiederverwendet. Auf diese Weise erhält das Tool, das den erzeugten VHDL-Code weiter synthetisiert, mehr Möglichkeiten und Freiheitsgrade bei der Optimierung.

Abb. 7.2.1 zeigt ein Beispiel für die Umsetzung von Grundoperationen. Hier ist ein Ausschnitt aus dem getakteten und dem ungetakteten Prozess dargestellt, die den Automaten implementieren. Die UND-Verknüpfung sowie die Schiebeoperation werden im selben Takt ausgeführt und durch die `and-` sowie die `srl`-Operation dargestellt. Die Ausführung findet im ungetakteten Prozess statt. Da die Ergebnisse, die `w0` und `w2` zugewiesen werden, im nächsten Takt von der Multiplikation benötigt werden, werden sie nochmals im getakteten Prozess in Registern gespeichert. Die *Resize*-Befehle und die Typkonvertierungen werden benutzt, um die entsprechenden Datentypen und -breiten zu erhalten, so dass die Breiten

der Operanden zu denen der Ergebnisse passen. Außerdem benötigt die Standardbibliothek Signale vom Typ `signed` oder `unsigned`, um die entsprechenden Funktionen ausführen zu können.

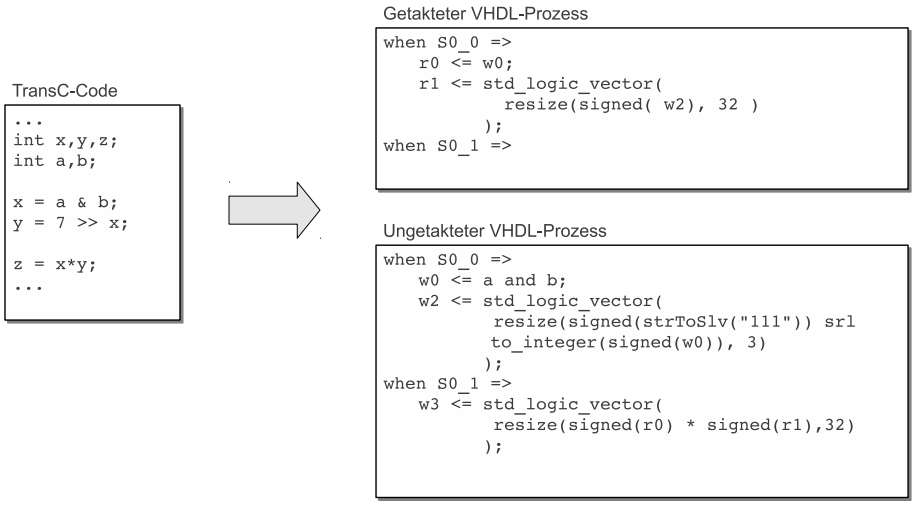


Abbildung 7.2.1: Umsetzung von Grundoperationen in VHDL

Operationen, die nicht von der numerischen Standardbibliothek unterstützt werden, müssen entweder vorher in äquivalente unterstützte Operationen umgewandelt oder durch Prozessaufrufe substituiert werden. Als Beispiel wäre hier etwa die Division zu nennen, die durch einen entsprechenden Prozessaufruf ersetzt wird. Prozessaufrufe eignen sich eher als Funktionsaufrufe, da im Prozessmodell von TransC Funktionsaufrufe immer sequentiell durchgeführt werden müssen, Prozesse aber auch parallel ablaufen können. Demnach sind die Grundoperationen Prozesse mit konstanter Ausführungszeit, da diese gleichzeitig gescheduled werden, wenn es die Datenabhängigkeiten und die Hardwareressourcen zulassen.

7.3 Prozessaufrufe

Durch die statische und hierarchische Prozessstruktur können Subprozesse direkt auf Instanzen einer Entity abgebildet werden. Die Prozesshierarchie, wie sie etwa in Abb. 3.4.1 auf Seite 66 dargestellt wird, ist somit identisch zur entsprechenden Modulstruktur in Hardware. Die Synchronisation mit dem aufrufenden Prozess wird mittels Handshaking durchgeführt.

Zusätzlich zu den Daten- und Steuerleitungen für Speicher und synchrone Streams erhält die Entity in ihrer Schnittstelle zwei Signale, `run` und `done`, um den Automaten zu steuern und mit der übergeordneten Entity zu kommunizieren (Abb. 7.1.1). Der Automat wird aus seinem Wartezustand mit einem hohen Pegel auf dem `run`-Signal geholt. Erst nachdem der Automat seine Ausführung beendet hat und wieder in den Wartezustand zurückkehrt, kann erneut `run` aktiviert werden. Ist der Automat im Wartezustand, wird `done` auf 1 gesetzt, um der übergeordneten Entity die Bereitschaft zu zeigen, ansonsten ist `done` 0. Auch zyklische Prozesse, die nicht terminieren, besitzen ein `done`-Signal. Auf

diese Weise wird die Schnittstelle konsistent gehalten. Beim Prozessaufruf wird das `run`-Signal der jeweiligen Unterkomponente im entsprechenden Kontrollschritt der FSM auf 1 gesetzt und die Operanden an den Eingängen angelegt. In dem Kontrollschritt, in dem die Rückgabewerte benötigt werden, wird so lange gewartet, bis `done` wieder auf 1 gesetzt ist. Erst dann sind auch die Ergebnisse an den Ausgängen gültig.

Prozesse, die nur aus einem Grundblock bestehen und keinen Kontrollfluss enthalten, kommen ohne Automaten aus, wenn bei diesen im Quellcode ein `pipeline`-Pragma gesetzt ist. Die Operationen im Datenpfad werden dann direkt implementiert und über Pipelinestufen miteinander verbunden. Allerdings steigt dadurch der Ressourcenverbrauch erheblich, da jede Operation im Datenfluss eine eigene Funktionseinheit erhält. Auf der anderen Seite können in jedem Takt neue Operanden angelegt werden und die Ausführungszeit ist konstant, so dass die Handshaking-Mechanismen hier nicht mehr benötigt werden. Aufgrund der Schnittstellenkompatibilität sind die Signale `run` und `done` dennoch vorhanden, obwohl sie nicht verwendet werden. Beim Scheduling müssen solche Prozessaufrufe wie Operationen mit einer festen Ausführungszeit behandelt werden.

Besitzt ein Prozess keinen Kontrollfluss und als einzige Operationen nur über Streams miteinander verbundene Subprozesse, so kann sein innerer Aufbau stark vereinfacht und Ressourcen eingespart werden. In diesem Fall weisen die Subprozesse keine Datenabhängigkeiten zueinander auf, wodurch es auch keine zeitliche Ausführungsreihenfolge gibt. Alle Unterprozesse müssen gleichzeitig gestartet werden und die Steuerung durch einen Automaten wird überflüssig. Die `run`-Signale der Unterentities werden direkt mit dem `run` der aufrufenden Entity verbunden, nach außen hin bleibt die Schnittstelle der Entity unverändert.

7.4 Streams

Asynchrone Ausgabestreams werden in Form von Registern implementiert, dessen Ausgänge aus der Schnittstelle der Entity hinausgeführt werden. Folglich werden die Signale im getakteten VHDL-Prozess des Automaten erzeugt. Eingabestreams sind Eingänge, deren Signale direkt genutzt und nicht über ein Register zwischengespeichert werden. Asynchrone Verbindungsstreams werden als Datenbusse implementiert, die Ein- und Ausgabestreams zwischen Subprozessen verbinden oder von Operationen im Datenpfad des Prozesses gelesen oder geschrieben werden.

Im Vergleich zu asynchronen Streams besitzen synchrone Streams keine Register in der Entity. Allerdings verwenden sie zum Datenaustausch mit FIFOs und anderen synchronen Streams zusätzliche Handshake-Signale. Bei der Implementierung von Ausgabestreams wird neben der Datenleitung ein `write`-Ausgang und ein `full`-Eingang in der Entityschnittstelle erzeugt, Eingabestreams erhalten einen `read`-Ausgang und ein `empty`-Eingangssignal, wie in Abb. 7.4.1 dargestellt. Ähnlich wie asynchrone Eingabe- und Ausgabestreams werden auch synchrone Streams mit Verbindungsstreams verknüpft. Wenn ein Verbindungsstream einen Zwischenspeicher besitzen soll, wird zwischen dem Ein- und Aus-

```
process void myProc(sistream<int> sstr_in,
                    sostream<int> sstr_out,
                    istream<int> async_in,
                    ostream<int> async_out,
);
```

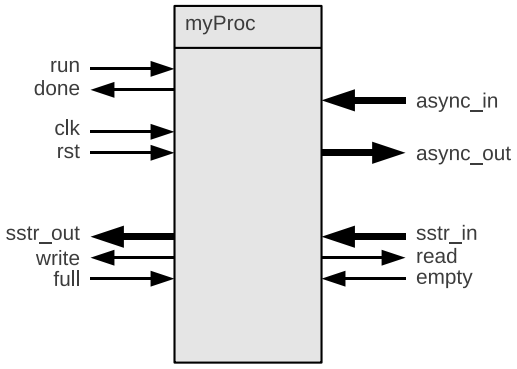


Abbildung 7.4.1: Prozessdeklaration und die erzeugte Hardwarekomponente mit synchronen und asynchronen Streams

gabestream eine FIFO implementiert, die mit den Daten- und Steuersignalen verbunden ist. Ansonsten werden die Streams ohne Zwischenspeicher direkt mit zusätzlicher Logik zur Synchronisierung der Handshake-Signale verschaltet, wie in Abb. 7.4.2 gezeigt. Aufgrund der kürzeren Zugriffszeiten bestehen FIFOs immer aus Distributed-RAM und nicht aus Block-RAM. Sollen sehr große Zwischenspeicher genutzt werden, so müssen extra Prozesse zwischen den Stream geschaltet werden, die für die Zwischenspeicherung der Daten z.B. in einem großen Block-RAM oder einem externen DRAM zuständig sind.

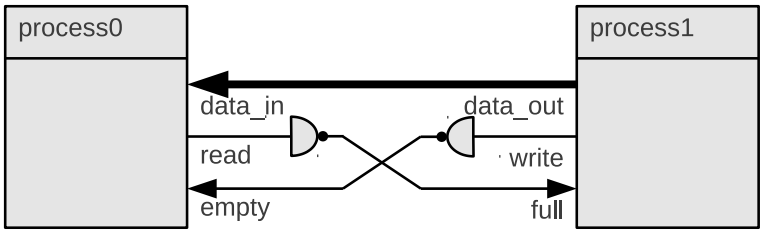


Abbildung 7.4.2: Verbindung von zwei synchronen Streams ohne FIFO

Wenn Daten von einer Operation aus in einen synchronen Verbindungs- oder Ausgabestream geschrieben werden, wird das `write`-Signal im asynchronen Prozess des Automaten auf 1 gesetzt und die Daten angelegt. Der Automat kann jedoch nur in den nächsten Zustand wechseln, wenn das entsprechende `full`-Signal 0 ist, so dass Daten geschrieben werden können. Volle FIFOs reagieren nicht auf Schreibanfragen, so dass `write`-Signale gesetzt werden können, auch wenn die FIFO keine Daten annehmen kann. Das Lesen aus synchronen Verbindungs- und Eingabestreams erfolgt analog dazu. Nur wenn das `empty`-Signal nicht gesetzt ist, wechselt der Automat in den nächsten Zustand.

Das sofortige Setzen der `write`- und `read`-Signale führt auch Probleme mit sich, wenn etwa im selben Kontrollschritt aus zwei synchronen Streams gelesen werden soll, wobei der eine Daten besitzt, der andere aber nicht. In diesem Fall bleibt der Automat so lange im selben Zustand, bis auch der zweite Stream Daten enthält. In dieser Zeit werden aus dem ersten Stream jeden Takt Daten entnommen, die jedoch nicht gespeichert werden. Zusätzliche Kontrollmechanismen würden zu einem erhöhten Hardwareaufwand führen.

Wenn mehrere synchrone Streams in einem Kontrollschritt abgefragt werden, so wird eine Warnung erzeugt. Somit kann der Programmierer diese Abfragen explizit trennen und hintereinander ausführen. Das hat den Vorteil, dass Streamabfragen, bei denen bekannt ist, dass immer Daten anliegen, trotzdem im selben Kontrollschritt ausgeführt werden können. Dies führt zu einer geringeren Laufzeit, als wenn mehrere Streamabfragen vom Scheduler immer in verschiedene Kontrollschritte gelegt werden. Obwohl Streamoperationen im Datenflussgraphen durch mehrere Knoten wie den primären und sekundären Referenzen dargestellt werden, wird nur eine einzige Operation gescheduled und im Automaten implementiert. Bevor die Streams in den entsprechenden VHDL-Code übersetzt werden können, muss die korrekte Benutzung, also sowohl ihre Verbindungen als auch die Randbedingungen für die Zahl von Quellen und Senken (Tabelle 3.1) überprüft werden.

7.5 Arrays

Arraydefinitionen in der Eingangssprache werden als Block-RAM bzw. Distributed-RAM in der Entity instantiiert. Im Compiler kann der Grenzwert für die Array-Größe eingestellt werden. Dieser bestimmt, ab wann auf Block-RAM statt Distributed-RAM zurückgegriffen wird. Block-RAM wird mit dedizierten SRAM-Blöcken realisiert, während Distributed-RAM aus Logikzellen aufgebaut ist. Mit Block-RAM können große Speicher realisiert werden, die keine Logikzellen verbrauchen, während Distributed-RAM in beliebiger Konfiguration erstellt werden kann (Größe, Datenbreiten) und besonders bei kleinen Größen erhebliche Geschwindigkeitsvorteile mit sich führt. Jedes Array erhält einen eigenen Speicher. Sollen mehrere Arrays an verschiedenen Positionen im selben Speicher abgelegt werden, so muss dies vorher durch Zwischencodetransformationen durchgeführt worden sein. Innerhalb des Automaten werden Arrayzugriffe im ungetakteten VHDL-Prozess implementiert. Dabei werden sowohl Adressen als auch Steuersignale und ggf. Schreibdaten im selben Kontrollschritt angelegt. Obwohl Arrayoperationen im Datenflussgraphen durch mehrere Knoten wie den primären und sekundären Referenzen dargestellt werden, werden sie beim Scheduling nur als eine einzige Operation betrachtet.

Wenn Arrays als Argumente dem Prozess übergeben werden, werden Speicherbusse und die dazugehörigen Kontrollsignale in die Entity geführt. Ein Speicherport einer Entity kann mit mehreren Arrays verbunden sein. Wenn dem Prozess bei einem Aufruf ein bestimmtes Array übergeben wird und bei einem späteren Aufruf ein anderes Array, so werden die Signale des generierten RAMs gemultiplext. Die *select*-Signale der Multiplexer werden bei jedem Aufruf neu gesetzt, so dass die Entity mit dem entsprechenden Speicher verbunden ist. Multiplexer werden auch benutzt, wenn mehrere Entities auf dasselbe Array zugreifen. Auf diese Weise kann derselbe Port einer Entity auf mehrere Speicher zugreifen und ein Speicher kann mit mehreren Entities verbunden werden. Die einzige Einschränkung hierbei ist die Art des RAMs, da aufgrund der unterschiedlichen Speicherlatenzen einem Port entweder nur Block-RAM oder nur Distributed-RAM übergeben werden darf. Ein RAM-Port besteht aus den Daten- und Adressbussen, die auf die Breite des Datentyps und die Größe

des Arrays angepasst sind, wobei es einmal einen Datenbus für Schreib- und einen für Lesedaten gibt. Ist etwa ein Array mit 1024 Einträgen in der Argumentenliste deklariert, so hat der Adressbus eine Breite von 10-Bit. Stimmen die Breiten von übergebenem Speicher und in der Argumentenliste des Prozesses deklarierten Speicher nicht überein, so werden die Breiten angepasst, auch wenn dabei Bits verloren gehen. Die Zugriffe werden mit den Signalen *en* und *we* gesteuert. Bei Lesezugriffen muss *en* aktiv sein, bei Schreibzugriffen beide Signale. Ist der Speicher als konstant definiert, so gibt es kein *we*-Signal und keinen Datenbus für Schreibdaten. Besitzt der Knoten für den Arrayzugriff zusätzlich eine Zugriffsbedingung (Abschnitt 5.4), so wird *en* nicht auf 1 gesetzt, sondern erhält dessen Ergebnis. Das gelesene Element ist undefiniert, wenn diese Bedingung *false* ist.

```
process void myProc(const int a[], int b[]);
```

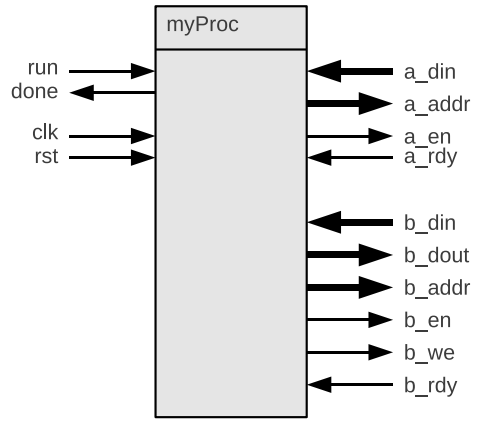


Abbildung 7.5.1: Prozessdeklaration und die erzeugte Hardwarekomponente mit Arrays. Bei dem konstanten Array fallen Daten und Steuersignale für Schreibzugriffe weg.

Gleichzeitiges Lesen und Schreiben in ein Array von mehreren Komponenten ist nicht möglich. Wenn gleichzeitige Zugriffe auftreten, bekommt nur die Komponente mit der höchsten Priorität die Erlaubnis, den Speicher zu benutzen. Die anderen verbleiben in ihrem aktuellen Zustand. Dies wird durch ein zusätzliches `rdy`-Signal erreicht, das Teil der Speicherschnittstelle ist. Bei einem Speicherzugriff kann der Automat eines Prozesses erst einen Zustandswechsel durchführen, wenn `rdy` 1 ist. Greifen Komponenten auf den Speicher zu, was mit `en` signalisiert wird, so wird `rdy` von allen Komponenten, die mit dem Array verbunden sind, auf 0 gesetzt. Nur bei der Komponente mit der höchsten Priorität bleibt `rdy` auf 1, wodurch diese den Zugriff erhält. Die Prioritäten sind statisch bestimmt und ändern sich zur Laufzeit nicht mehr. Abbildung 7.5.2 zeigt die Verschaltung von drei Prozessen, die sich einen Speicher teilen. Die Signale von der Komponente zum Speicher werden durch kaskadierte Multiplexer geleitet. Das *en*-Signal der höherpriorisierten Komponente bestimmt die Beschaltung der Multiplexer. Abb. 7.5.3 zeigt ein Beispiel für eine Komponente, die in verschiedenen Programmstufen verschiedene Arrays benutzt. Beim Prozessaufruf wird ein Multiplexer und ein Demultiplexer so umgeschaltet, dass die vom Prozess erzeugten Signale mit dem entsprechenden Array verbunden sind und auch vom entsprechendem Speicher gelesen wird. Der Multiplexer bleibt so lange in seinem Zustand, bis er durch einen neuen Aufruf umgeschaltet wird. Für diese Struktur haben die `rdy` und

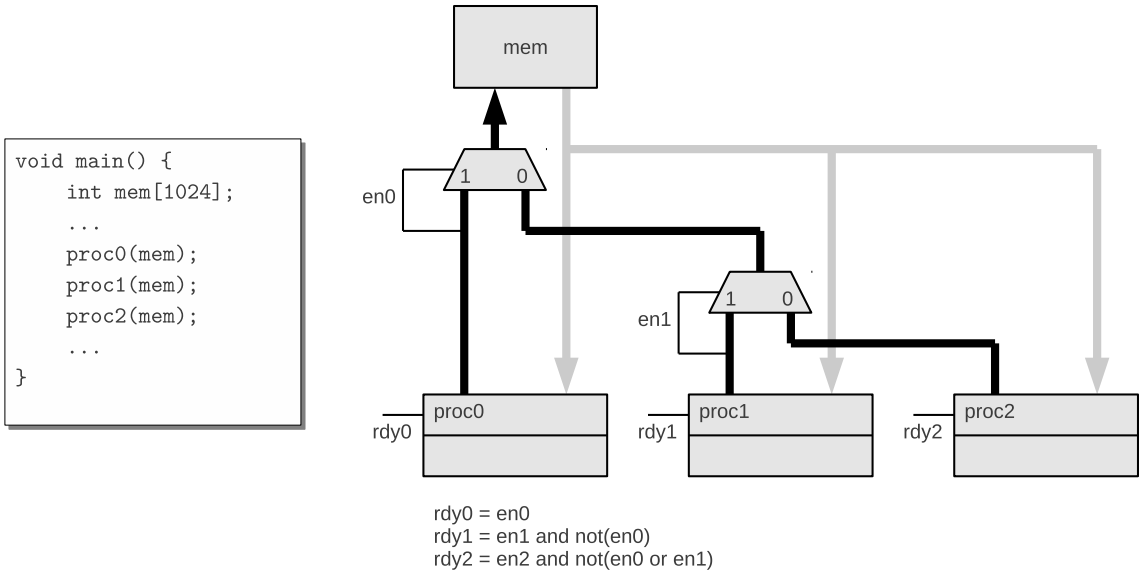


Abbildung 7.5.2: Verschaltung von drei Prozessen, die sich einen Speicher teilen. Die dicken schwarzen Linien stellen alle Signale dar, die vom Prozess zum Speicher laufen, wobei die *en*-Signale gleichzeitig die Multiplexer steuern. Die dicken grauen Linien sind Datensignale vom Speicher zum Prozess, diese müssen nicht gemultiplext werden.

`en`-Signale keine Bedeutung. Eine Mischform mit n Prozessen und m Speichern ist ebenfalls möglich, wobei jeder Prozess einen eigenen Multiplexer und Demultiplexer enthält und die kaskadierte Multiplexerstruktur vor den Speichern mit diesem anschließend verbunden wird.

`rdy` kann auch als Handshake-Signal benutzt werden, um Komponenten z.B. mit SDRAM-Kontrollern zu verbinden statt mit Block- oder Distributed-RAM. Wenn die Komponente auf SDRAM zugreift, ist `rdy` auf 0 gesetzt, bis die Daten tatsächlich anliegen.

Parallele Speicherzugriffe innerhalb eines Prozesses können mit dem Zwischenformat dargestellt werden. In diesem Fall besitzt der DFG mehrere Zugriffsknoten, die aufgrund der begrenzten Zahl von Speicherports durch den Scheduler serialisiert werden. Wird ein Array allerdings auf Dualport-Speicher abgebildet, was durch den Programmierer explizit angegeben werden kann, können zwei Zugriffe auf die Ports verteilt und parallel ausgeführt werden.

7.6 Datenstrukturen

Bevor Datenstrukturen in Hardware implementiert werden können, werden diese in eindimensionale Arrays äquivalenter Größe umgewandelt. Dabei werden Instanzen von Datenstrukturen in Arraydefinitionen konvertiert, aus den Strukturübergaben entstehen Arrayübergaben. Bei der Codegenerierung müssen Datenstrukturen somit nicht mehr berücksichtigt werden. Die Breite des Arrays ist dabei die Breite des größten Elements der Datenstruktur, die Anzahl der Elemente von Struktur und Array sind identisch, wie Abb.

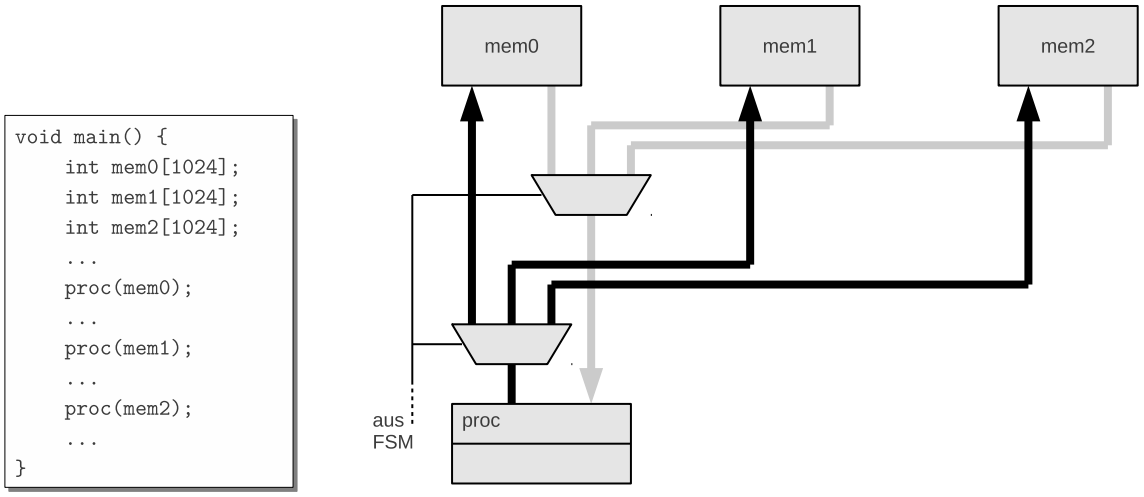


Abbildung 7.5.3: Verschaltung von einem Prozess, dem je nach Aufruf verschiedene Arrays übergeben werden. Die Beschaltung der Multiplexer wird von der FSM gesteuert, aus der `proc` aufgerufen wird.

7.6.1 zeigt. Diese Vorgehensweise ist eigentlich suboptimal, da bei kleineren Typen das Arrayelement nicht mehr vollständig ausgenutzt wird und mehr Speicher als notwendig allokiert wird. In weiteren Ausbaustufen des Synthesetools können Strukturelemente in einem Arrayelement zusammengefasst werden, so lange die Summe ihrer Breiten die Arraybreite nicht überschreitet. Allerdings gibt es dann Probleme bei Schreibzugriffen, da für die Änderung eines Elements zunächst ein Lesezugriff stattfinden muss, damit die anderen Elemente in der Speicherstelle ihren korrekten Wert wieder erhalten. Die Datenstruktur in dem Beispiel könnte auch in einem Array mit elf statt zwölf Feldern untergebracht werden, wenn man `ch` und `b` zusammenfasst.

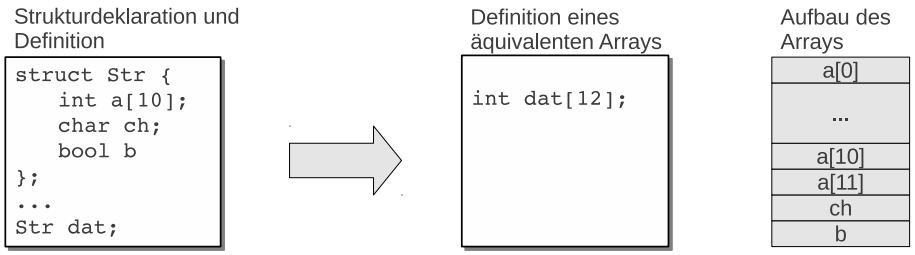


Abbildung 7.6.1: Datenstruktur und äquivalente Arraydefinition

Der C-to-Verilog-Compiler [90, 89] unterstützt ebenfalls Datenstrukturen, die allerdings nicht als Array, sondern als großes Register umgesetzt werden. Der Vorteil hier ist die höhere Parallelität, da man auf alle Elemente simultan zugreifen kann. Der Nachteil liegt im Verdrahtungsaufwand und in der Registerbreite. Für das Beispiel in Abb. 7.6.1 bräuchte man bei dieser Vorgehensweise ein 329-Bit breites Register ($10 \cdot 32 + 8 + 1$). Zudem können Strukturen in C-to-Verilog nur als Kopie, nicht aber als Referenz an eine Funktion übergeben werden. Der Ansatz mit den Arrays profitiert auch von Arrayoptimierungen wie der Eliminierung von Arrayzugriffen (Abschnitt 2.4.2 bzw. Abschnitt 6.2.2), der Umwand-

lung in Register (Abschnitt 2.4.2) oder den bedingten Zugriffen (Abschnitt 5.4) bzw. der Verwendung von Dualport-RAMs.

Das i-te Element der Datenstruktur wird auf das i-te Arrayelement abgebildet. Verschachtelte Datenstrukturen, d.h. Strukturen, die weitere Datenstrukturen und Arrays enthalten, werden abgeflacht. Dazu müssen alle Arraygrößen innerhalb der Struktur bekannt sein. Strukturzugriffe werden in Arrayzugriffe konvertiert, wobei der korrekte Index aus dem verschachtelten Strukturzugriff ermittelt werden muss. Abb. 7.6.2 zeigt eine Datenstruktur, die eine weitere Datenstruktur enthält. Der relativ komplexe Zugriff auf diese Struktur muss in eine Indexberechnung konvertiert werden, die allerdings mit Hilfe der Konstantenpropagation vereinfacht werden kann. Beim Strukturzugriff zeigt der erste Index auf das Element `x`. `x` ist das zweite Element in der Datenstruktur `y` und erhält somit den Index `1`. Da `x` ein Array ist, ist dort der Zugriffsindex `i`, wobei innerhalb des Arrays wieder auf das erste Element `a` zugegriffen wird, das mit dem Index `0` adressiert wird. Auf diese Weise kommt die Indexfolge `1-i-0` zustande. Beim Arrayzugriff muss der Index `i` mit `2` multipliziert werden, da die Unterstruktur `x` zwei Elemente enthält. Da `x` an fünfter Stelle in der Struktur `Y` anfängt, wird dieser Index um `5` erhöht. Der letzte Operand `1` des Arrayzugriffs ist die Zugriffsbedingung (Abschnitt 5.4), die im Normalfall immer 1 ist und sich erst durch Transformationen wie in Abschnitt 6.1 beschrieben ändern kann.

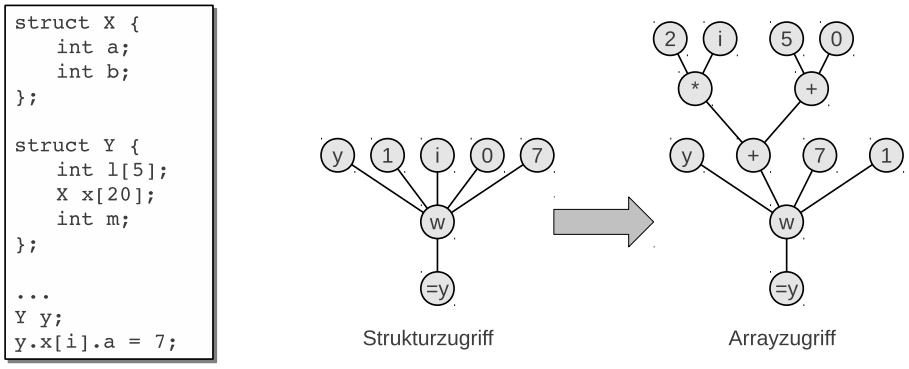


Abbildung 7.6.2: Konvertierung eines Struktur- und einen Arrayzugriff

In TransC können Datenstrukturen als Referenz an Prozesse und Funktionen übergeben werden. Diese Übergaben werden ebenfalls in Arrayübergaben umgewandelt. Die Breite des Adressbusses ist hier allerdings nicht an die Zahl der Elemente angepasst, da die übergebene Datenstruktur Teil einer größeren verschachtelten Struktur sein kann. Aus diesem Grund bekommt der Adressbus hier eine fest definierte Standardbreite. Zusätzlich zum eigentlichen Array wird ein Offset übergeben, der die Anfangsadresse der übergebenen Struktur in der gesamten Struktur angibt. Dieser Offset wird bei jeder Schreib- und Leseoperation auf den Zugriffsindex addiert. Ist die übergebene Struktur keine Unterkomponente einer größeren Struktur, so ist dieser Offset 0.

7.7 Ermittlung der Datenbreite und des Typs

Zur Implementierung des Datenflussgraphen muss für jedes Resultat die Bitbreite sowie der Datentyp feststehen. Das hier an einem Beispiel erläuterte Verfahren betrachtet vorzeichenlose und vorzeichenbehaftete Ganzzahltypen. Andere Datentypen wie Fließkomma werden wie vorzeichenlose Ganzzahlen behandelt, auf diesen dürfen allerdings auch keine Standardoperationen wie Addition oder Multiplikation durchgeführt werden. Die arithmetischen Grundoperationen müssen vorher durch entsprechende Prozessaufrufe ersetzt worden sein, die den Datentypen korrekt verarbeiten. Prozessaufrufe werden insofern berücksichtigt, als dass sowohl die Operanden- als auch die Resultatbreiten bereits definiert sind und dieser dann ähnlich wie eine Variablenzuweisung im Datenfluss wirkt.

Die Datenbreite und der Datentyp des Ergebnisses bei arithmetischen Grundoperationen richtet sich immer nach den Operanden und der Operation selber. Die Breite des Ergebnisses ist immer so gewählt, dass keine Bits verloren gehen. Bei einer Addition etwa, wo der eine Operand 5-Bit hat und der andere 7, hat das Ergebnis eine Breite von 8 Bit. Wenn die Operanden nicht alle denselben Typ haben (z.B. vorzeichenbehaftet und vorzeichenlos), so wird der vorzeichenlose Typ zu einem vorzeichenbehafteten Typ konvertiert, wobei dann auch die Datenbreite des Operanden um 1 erhöht werden muss. Auf diese Weise wird verhindert, dass Werte, die die maximale Breite ausnutzen, als negative Zahl interpretiert werden.

Die Breite des Datenbusses muss für jedes Resultat einer Operation ermittelt werden, so dass diese mit der Breite der Operanden zusammenpasst. Dafür wird der DFG entlang des Datenflusses durchlaufen und für jede Kante die Bitbreite ermittelt. Am Anfang eines DFGs stehen entweder Konstanten oder Eingabeknoten, deren Typ immer bekannt ist und somit auch dessen Breite. Bei Operationsknoten wird die Resultatbreite dann anhand der Breite der Operanden ermittelt, sodass keine Bits verloren gehen. Bei der Multiplikation beispielsweise ist die Resultatbreite die Summe der beiden Operandenbreiten, bei der Addition die maximale Operandenbreite um 1 erhöht. Die einzige Operation, bei der die Resultatbreite auf diese Weise nicht ermittelt werden kann ist die Linksschiebeoperation, da hier die Resultatbreite exponentiell mit der Breite des zweiten Operanden zunimmt. Daher bestimmt hier die Breite des ersten Operanden die Ergebnisbreite, wie es z.B. auch der Programmiersprache C durchgeführt wird. Da durch Optimierungen die Breiten der Operanden reduziert werden können, wird für jede Schiebeoperation die Resultatbreite vor den Optimierungen und Transformationen gespeichert, so dass sie sich nicht mehr ändern kann. Ist der DFG entlang des Datenflusses durchlaufen (vorwärts), wird er anschließend entgegengesetzt des Datenflusses traversiert (rückwärts), um die Breiten weiter zu reduzieren oder an die Operationen anzupassen, wie in Abb. 7.7.1 dargestellt. [16] zeigt ein ähnliches Verfahren, mit dem die Bitbreiten im Datenpfad reduziert werden, welches das Programm ebenfalls vorwärts und rückwärts durchläuft.

Das Ergebnis der Addition z.B. ist nach der Vorwärtstraversierung 49-Bit breit. Durch die Rückwärtstraversierung wird ermittelt, dass von den 49-Bit lediglich 32 benutzt werden,

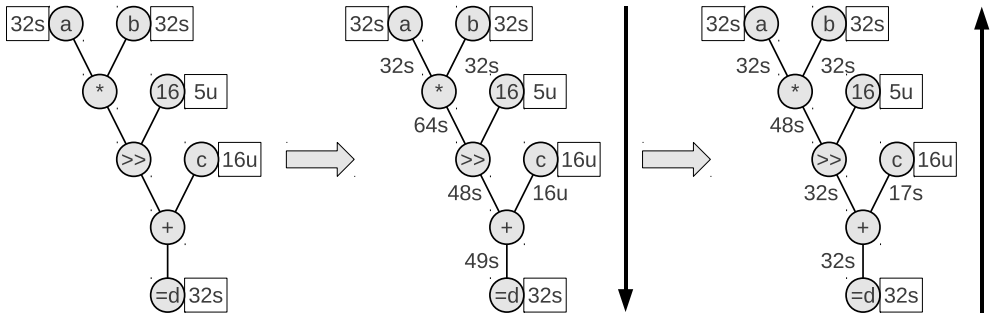


Abbildung 7.7.1: Ermittlung der Bitbreiten der Resultate in einem DFG. Die Typen und Breiten der Konstanten und Variablen sind anfangs bekannt. s steht für einen vorzeichenbehafteten, u für einen vorzeichenlosen Datentypen.

sodass die Breite weiter reduziert werden kann. Auf der anderen Seite benötigt der Knoten c aufgrund des Datentyps nach der Vorwärtstraversierung einen 16-Bit breiten Bus. Aufgrund der nachfolgenden Addition muss dieser allerdings wegen der Typkonvertierung um 1-Bit erhöht werden, was nur durch den Durchlauf entgegengesetzt des Datenflusses ermittelt werden kann.

7.8 Zeitmodell

Zeitabfragen sind durch die `time`-Funktion möglich. Sobald diese Funktion in Verwendung ist, wird im Prozess ein Register instantiiert, das jeden Takt inkrementiert wird. Der Aufruf `time(0)` setzt das Register auf 0, `time(1)` gibt den aktuellen Wert zurück. Innerhalb der Automaten im VHDL-Code unterscheidet sich `time` nicht von herkömmlichen Funktionen, so dass hier keine besonderen Maßnahmen zu treffen sind.

7.9 Guards

Die Umsetzung der Input- und Output-Guards in VHDL ist trivial. Innerhalb der Sprache wird über den *block*-Operator der Zustand eines synchronen Streams abgefragt, der bei Eingabestreams mit dem Signal `empty` und bei Ausgabestreams mit `full` angezeigt wird. Wie in Abb. 7.4.1 auf Seite 136 dargestellt, sind diese Status-Signale bereits in der Entity vorhanden, so dass sie bei entsprechenden Anweisungen lediglich mit den Signalen in den VHDL-Prozessen verbunden werden müssen. Ein zusätzlicher Hardwareaufwand ist nicht notwendig.

7.10 Externe VHDL-Module

Die Einbindung von bestehenden VHDL-Modulen wird bei der Hardwaregenerierung unterstützt. Programmierer können mit diesen Modulen über Prozessaufrufe, Streams oder einen gemeinsamen Speicher kommunizieren. Anders als etwa in der SR-Sprache [35] gibt

es keine extra Sprachkonstrukte zur Deklaration und Integration dieser Komponenten. Bei einfachen Komponenten, die eine ähnliche Aktivierung über Handshakesignale sowie ähnliche Steuerungs- und Kommunikationsmechanismen wie TransC-Prozesse verfügen, reicht es aus, diese in einem Wrapper zu kapseln. Die Schnittstelle des dadurch entstandenen Moduls wird mittels einer normalen Prozessdeklaration dem Programm zur Verfügung gestellt. Für komplexere Schaltungen kann der Compiler erweitert werden. Hierfür gibt es eine Softwareschnittstelle in Form einer Basisklasse, die abgeleitet und mit den notwendigen Informationen zum Betrieb versehen wird. Diese Informationen beinhalten beispielsweise den Prozessprototypen, mit dem das Modul aufgerufen wird sowie die Ausführungszeit, wenn diese konstant ist und Spezifikationen für das Pipelining. Desweiteren müssen diverse Memberfunktionen für die VHDL-Codeerzeugung implementiert werden. Diese Memberfunktionen beziehen sich nur auf den VHDL-Code für die Verwendung des Moduls, nicht auf die Implementierung selber. Dieser Ansatz über die Erweiterung des Compilers ist zwar aufwändiger als die Einbindung über die Sprache, man erhält jedoch eine höhere Flexibilität, um auch komplexere Schaltungen zu integrieren. Die Ableitung wurde auch benutzt, um TransC beispielsweise um die oben genannten Zeitbedingungen (`time`) zu erweitern. Hier wurde ein Funktionsaufruf integriert, der Zeitgeber setzen und abfragen kann. Auf diese Weise können taktgenaue Wartezyklen durch Zeitabfragen eingestellt werden, um die Echtzeitfähigkeit des Prozessmodells zu realisieren, wie in Abschnitt 3.8 beschrieben.

Kapitel 8

Beispiel-Implementierungen

8.1 Benchmarks

8.1.1 Vergleich mit anderen Synthesetools

In diesem Abschnitt werden Benchmarks durchgeführt, um sowohl die Effizienz als auch den Ressourcenverbrauch der TransC-Ausgabe zu untersuchen und mit anderen Synthesetools zu vergleichen. Zudem soll gezeigt werden, dass die parallelen Sprachkonstrukte ohne großen Schaltungsaufwand implementiert werden können. Der von den Synthesetools und vom TransC-Compiler erzeugte VHDL und Verilog-Code wurde von der Software ISE WebPACK 13.1[55] für einen Xilinx Spartan-6 SLX75T FPGA weiter synthetisiert. Dort wurde die Schaltung auf eine hohe Taktfrequenz optimiert mit einem ausgeglichenem Designziel (*balanced design goal*). Es wurden keine Randbedingungen für Fläche oder Zeit gesetzt, die Ergebnisse wurden nach der Platzierung und Verdrahtung ermittelt.

Zwölf Benchmarks wurden portiert und an die Gegebenheiten der Synthesetools angepasst. So sind Pointer durch Arrayübergaben und Indizes ersetzt oder nicht unterstützte Sprachkonstrukte wie lokale Variablen oder `enum`-Befehle abgeändert worden. Die Benchmarks decken Applikationen ab, die sowohl durch Datenfluss als auch durch Kontrollfluss dominiert werden. Sieben Testprogramme stammen von dem C-to-Verilog-Projekt [89], die restlichen wurden aus diversen Anwendungen entnommen.

- Benchmarks aus dem C-to-Verilog-Projekt:
  - bubble
  - crc
  - dwt
  - gcd
  - sha1
  - walsh
  - yuv2rgb

- Andere Benchmarks:
  - circle
  - line
  - sqrt
  - matmul
  - sieve

circle und *line* berechnen Kreise bzw. Linien aus Koordinaten. *sqrt* ist ein Algorithmus zur Berechnung von Wurzeln aus Ganzzahlen, *matmul* multipliziert zwei Matrizen und *sieve* dient zur Ermittlung von Primzahlen. Die vom TransC-Compiler generierten Resultate werden mit dem C-to-Verilog-Compiler[89] verglichen.

In C-to-Verilog wurden für Optimierungen die Standardeinstellungen verwendet, sofern nicht andere Konfigurationen empfohlen wurden. Auch die empfohlene Zahl an Speicherports zur Umsetzung der Arrays wurde übernommen. Im TransC-Synthesetool wurden alle vorhandenen Optimierungen aktiviert, einschließlich funktionalem Pipelining (6.3) und den Transformationen, die auf der partiellen Evaluierung basieren (6.2). Im C-to-Verilog-Compiler konnten bis auf die Zahl der Speicherports, das Abrollen von Schleifen und der Anwendung von Pipelining die Optimierungen nicht explizit gesetzt werden. Da es auf dem LLVM-Zwischencode [67] basiert, der eine Vielzahl von Optimierungen unterstützt, wird davon ausgegangen, dass dort für die Architektursynthese sinnvolle Optimierungen standardmäßig aktiviert sind. Die Abbildungen 8.1.1 bis 8.1.3 zeigen den relativen Verbrauch von LUTs, die Taktfrequenz und die Laufzeit der Benchmarks in Taktzyklen.

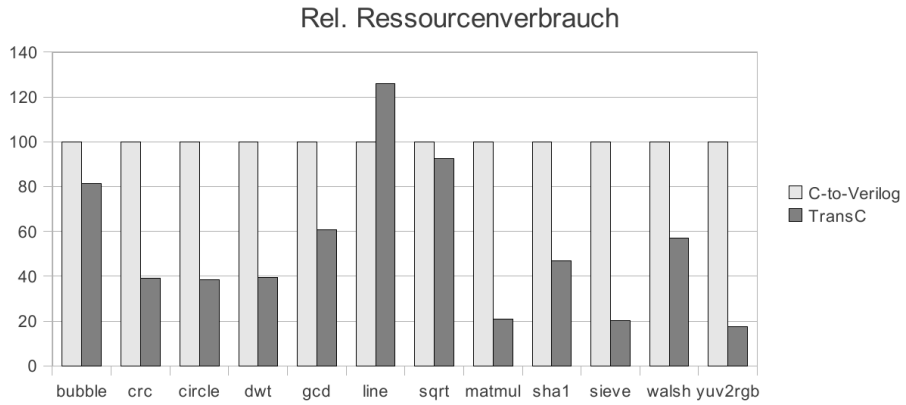


Abbildung 8.1.1: Relativer Ressourcenverbrauch nach Platzierung und Verdrahtung für TransC und C-to-Verilog

Im Vergleich zum erzeugten Code des C-to-Verilog Compilers war ISE WebPACK in der Lage, die Automaten im TransC-VHDL-Code zu erkennen und in allen Benchmarks zu optimieren. Dementsprechend waren die Ergebnisse in den kontrollfluss-dominierten Benchmarks wie Bubble-Sort, Prime-Sieve oder dem Linienalgorithmus von Bresenham wesentlich besser. Zudem besitzt die Ausgabe von C-to-Verilog eine ungewöhnlich große Anzahl

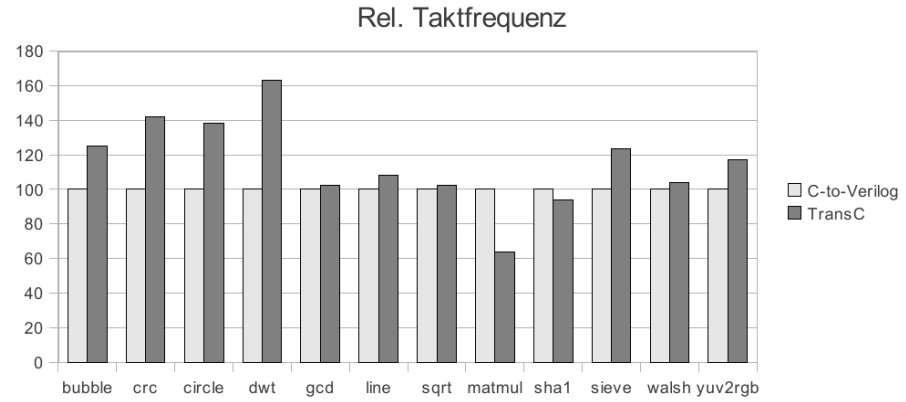


Abbildung 8.1.2: Relative Taktfrequenz nach Platzierung und Verdrahtung für TransC und C-to-Verilog

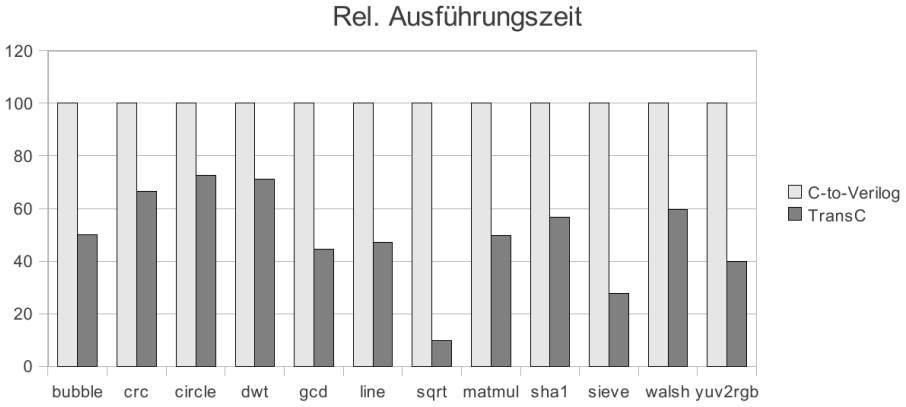


Abbildung 8.1.3: Relative Laufzeit in Takten für TransC und C-to-Verilog

an Ausführungseinheiten wie Addierer und viele Schaltungszustände, was sich ebenfalls negativ auf den Ressourcenverbrauch auswirkt. So meldet die Low-Level Synthese etwa 19 16-Bit Addierer beim Sieve-Algorithmus, wobei im C-Code selber nur sechs verschiedene Additionen vorkommen. Auch beim *dwt* allokierte C-to-Verilog 21 Addierer bzw. Subtrahierer, während im TransC-Compiler nur vier verwendet wurden. In C-to-Verilog haben im Vergleich zu TransC die Funktionseinheiten Multiplizierer und Shifter mehr Pipelinestufen, was sich in einigen Programmen durch einen höheren Takt, aber auch durch eine höhere Zyklenzahl bemerkbar macht. Im *bubble*-Algorithmus wurden vom TransC-Synthesetool die if-Zweige durch bedingte Speicherzugriffe ersetzt, wodurch Kontrollfluss eingespart wurde und sich die Laufzeit verkürzte. Den *dwt*, *crc*- und den *circle*-Benchmark haben beide Compiler durch die Implementierung von Pipelinestufen beschleunigt, durch die anderen Optimierungen und das Scheduling wurde in TransC eine kürzere Laufzeit erzielt. TransC konnte den *line*- und den *sqrt*-Benchmark im Vergleich zu C-to-Verilog wesentlich stärker parallelisieren, was in diesem Fall den Ressourcenverbrauch allerdings negativ beeinflusst hat. Bei *matmul* profitieren die C-to-Verilog-Schaltungen besonders von den vielen Pipeli-

nestufen der Multiplizierer, der in den TransC-Schaltungen den kritischen Pfad darstellt. Da die Schaltung aufgrund der vielen Speicherports zu viele Ein- und Ausgänge hat, wurden hier die Werte nach der Synthese und nicht nach der Platzierung und Verdrahtung genommen. In *sha1* wurden besonders die einfachen Schleifen zum Füllen oder Kopieren von Arrays vom TransC-Compiler effizient umgesetzt. Der *yuv2rgb*-Algorithmus konnte durch funktionales Pipelining optimiert werden, wobei sich die Zeit für eine Iteration der innersten Schleife von 7 auf 4 Takte reduzierte. Abbildung 8.1.4 zeigt die relativen Durchschnittswerte für alle Benchmarks. Der Ressourcenverbrauch und die Laufzeit sind nur halb so hoch, die Taktfrequenz konnte um 15 % gesteigert werden.

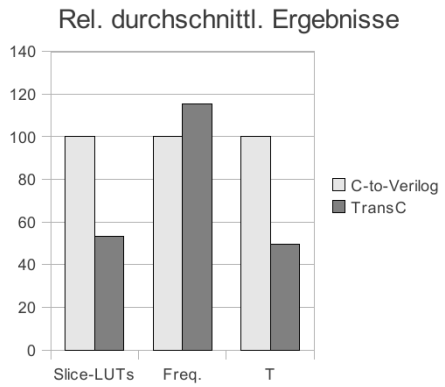


Abbildung 8.1.4: Relative Durchschnittswerte für TransC und C-to-Verilog

Andere Syntheseprogramme wie Spark[44] oder ROCCC[43] erzeugen bessere Schaltungen als der C-to-Verilog Compiler. Hier konnten aus verschiedenen Gründen die meisten Benchmarks jedoch nicht synthetisiert werden. Spark kann Arrays nicht effizient übersetzen, da es statt Adressleitungen für jedes Bit im Speicher ein `inout`-Signal erzeugt, wodurch sehr breite Busse in den Schnittstellen der Entities entstehen. Ein 16-Bit breites Array mit 256 Einträgen bräuchte somit ein 4096-Bit breites Signal. Spark kann z.B. den *gcd*-Benchmark synthetisieren, da hier keine Arrays definiert werden. Die Schaltung benötigt 483 Slice-LUTs, läuft mit einer Taktfrequenz von 173 MHz und berechnet den Divisor in 61 Taktzyklen, was ähnlich zur TransC-Ausgabe ist. ROCCC kann nur `for`-Schleifen mit konstanten Grenzen oder Endlosschleifen synthetisieren [81], variable Grenzen sind nicht möglich.

8.1.2 Parallele Prozesse

Um die Interprozesskommunikation für größere Applikationen mit anderen Compilern vergleichen zu können, wurde ein Mandelbrot-Programm, das auf der POSIX-Thread-Bibliothek [17] aufbaut und auf Festkommadaten arbeitet, nach TransC und Hybridthreads[95] portiert. Das Programm besitzt einen Hauptprozess, der verschiedene Ausschnitte der komplexen Ebene von Subprozessen berechnen lässt. Die Subprozesse ermitteln die Farben und schreiben das Resultat direkt in ein gemeinsames Array. In

Hybridthreads wird der gegenseitige Ausschluss über Mutexe realisiert, in TransC wurden vordefinierte Funktionen benutzt, die auf dem Peterson-Algorithmus [86] für mehrere Prozesse aufbauen. Die TransC-Ausgabe verbraucht 2132 Slice-LUTs und benötigt zur Ausführung 5,8 Mio. Taktzyklen, die mit einer Rate von 96 MHz abgearbeitet werden können. Die von Hybridthreads erzeugten Schaltungen verbrauchen hingegen 3557 Slice-LUTs und die Ausführung dauert 7,3 Mio. Taktzyklen bei einer Frequenz von 71 MHz. Da die Testapplikation bereits auf der POSIX-Threads-API aufbaute, war die Portierung nach Hybridthreads einfacher, aber die leichtgewichtige Prozessimplementierung in TransC und die einfachen Mechanismen für die IPC verbrauchen weniger Ressourcen. Wenn die Anwendung nicht auf POSIX-Threads basierte, wäre die Implementierung in TransC wesentlich einfacher gewesen aufgrund der aufwändigen Parameterübergabe bei den POSIX-Threads.

8.1.3 Pipelining

Dieser Abschnitt vergleicht die Ausgabe des TransC-Compilers mit und ohne Anwendung von funktionalem Pipelining. Im ersten Testlauf wurden bis auf das Pipelining alle Optimierungen, auch die partielle Evaluierung, aktiviert. Anschließend wurde das Pipelining mit aufgenommen. Dabei wurden die Parameter so gesetzt, dass die Transformationen bei einer Schleife nur angewendet werden, wenn dessen Laufzeit um mindestens 20 % reduziert werden kann. Abbildung 8.1.5 bis 8.1.7 vergleichen die Zahl der benötigten Slice-LUTs sowie die Taktfrequenz und die Laufzeiten der beiden Testläufe miteinander. Zusätzlich zu den oben beschriebenen Benchmarks aus dem C-to-Verilog-Projekt wurden mit *gsm* und *blowfish* zwei größere Benchmarks aus der CHStone-Suite [46] implementiert, die der C-to-Verilog-Compiler aufgrund der Größe nicht verarbeiten konnte.

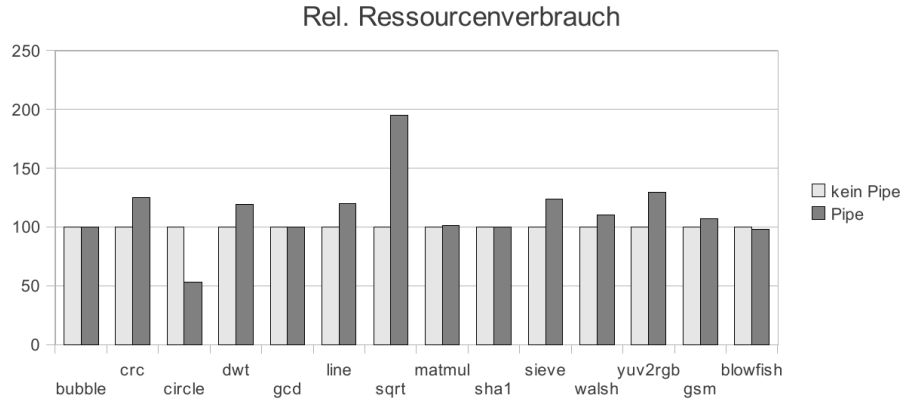


Abbildung 8.1.5: Relativer Ressourcenverbrauch mit und ohne funktionalem Pipelining

Wie man in den Grafiken erkennen kann, profitierten nicht alle Benchmarks vom Pipelining. Besonders die einfacheren Testprogramme ohne Schleifen oder mit speziellen Datenabhängigkeiten zeigen konstante Ergebnisse. Beim *line* und *sqrt*-Benchmark verschlechterten sich die Laufzeiten sogar, da zwar Pipeline-Strukturen aufgebaut wurden, diese jedoch aufgrund der geringen Zahl von Schleifeniterationen das Programm insgesamt nicht be-

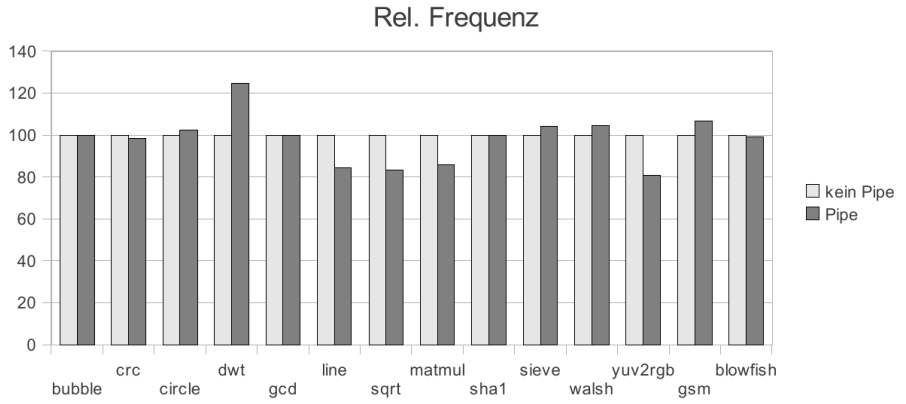


Abbildung 8.1.6: Relative Taktfrequenz mit und ohne funktionalem Pipelining

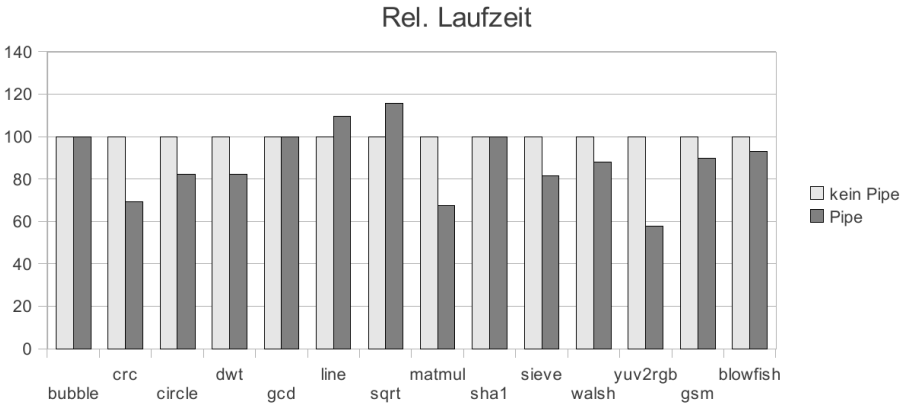


Abbildung 8.1.7: Relative Laufzeit in Takten mit und ohne funktionalem Pipelining

schleunigen konnten. Die erforderlichen Vorberechnungen mussten dennoch durchgeführt und zusätzliche Funktionseinheiten allokiert werden, die die Schaltung vergrößerten. Wie zu erwarten war, verringerte sich im Durchschnitt die Laufzeit der Algorithmen, während der Ressourcenverbrauch aufgrund der höheren Parallelität anstieg (Abb. 8.1.8). Trotz der komplexeren Datenpfade sank die durchschnittliche Taktfrequenz nur gering. Lediglich im *yuv2rgb* sank sie stärker ab, wobei hier eine Laufzeitersparnis von über 40 % erzielt werden konnte.

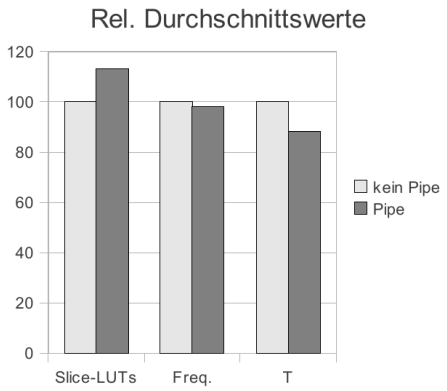


Abbildung 8.1.8: Relative Durchschnittswerte mit und ohne Pipelining

8.1.4 Partielle Evaluierung

Für den Test der partiellen Evaluierung wurden dieselben Testprogramme wie beim Pipelining verwendet. Durch die mit der partiellen Evaluierung verbundenen Optimierungen sollen weitere Konstanten detektiert sowie Bitbreiten reduziert und Speicherzugriffe eliminiert bzw. parallelisiert werden. Es wurden zwei Testreihen durchgeführt, in der ersten wurden die Benchmarks mit allen Optimierungen ohne PE synthetisiert, in der zweiten kam zusätzlich die PE hinzu.

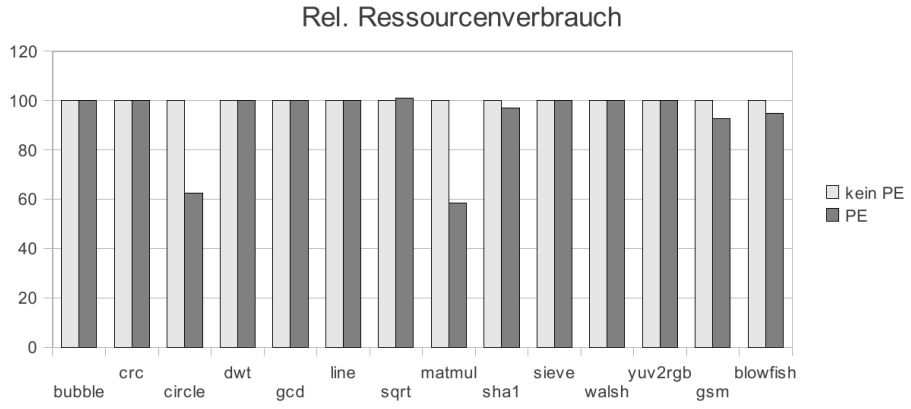


Abbildung 8.1.9: Relativer Ressourcenverbrauch mit und ohne partieller Evaluierung

Abbildungen 8.1.9 bis 8.1.11 zeigen die Anzahl der benötigten Slice-LUTs, die Laufzeit in Taktzyklen sowie die maximale Taktfrequenz der Schaltungen relativ zu den Ergebnissen ohne partielle Evaluierung. Alle Optimierungen, auch Pipelining wurden aktiviert. Wenn die partielle Evaluierung erfolgreich angewendet werden konnte, wurde der Ressourcenverbrauch aufgrund der Reduzierung der Datenpfadbreiten signifikant verringert. Allerdings profitierten davon nur zwei der zwölf kleineren Benchmarks. Der höhere Ressourcenverbrauch des *sqrt*-Programms liegt am Low-Level-Synthesetool, das den erzeugten VHDL-Code weiterverarbeitet. Auch hier hätte sich der Verbrauch aufgrund von geringeren Datenbreiten eigentlich verringern sollen. Bei Benchmarks mit konstanten Schleifengren-

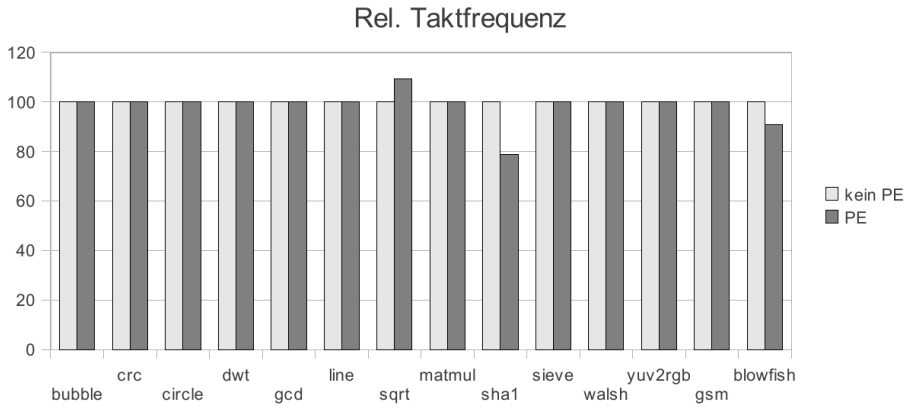


Abbildung 8.1.10: Relative Taktfrequenz mit und ohne partieller Evaluierung

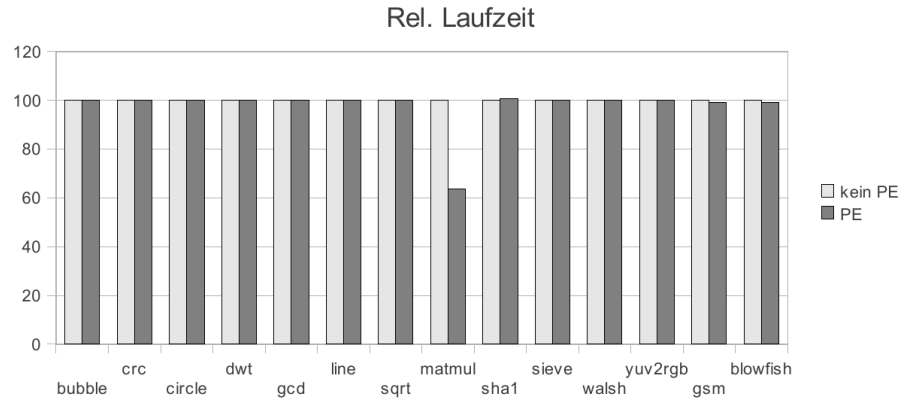


Abbildung 8.1.11: Relative Laufzeit in Takten mit und ohne partieller Evaluierung

zen und Arrayzugriffen wurden die Arrayindizes korrekt berechnet und aufeinanderfolgende Speicherzugriffe parallelisiert. Dadurch war in *matmul* die Laufzeit niedriger als in Testreihen, in der PE nicht aktiviert war. Die höhere Parallelität beeinflusste die maximale Taktfrequenz nicht negativ.

Die größeren Testprogramme *gsm* und *blowfish*, die aus mehreren hundert Zeilen Quellcode bestehen, zeigen mit partieller Evaluierung eine um etwa 5 bis 10 % geringere Laufzeit. Die Auswertung von Funktionen, die aufgrund eines komplexen Kontrollflusses viele Verzweigungen und unbekannte Sprungbedingungen besaßen, wurde vorzeitig abgebrochen, wenn die Funktion *evalBlocks* des PE-Algorithmus eine Rekursionstiefe von 20 erreichte. Auf diese Weise hielt sich auch die Kompilierzeit in einem angemessenen Rahmen und erhöhte sich durchschnittlich nur um 200 ms bzw. 18 % im Vergleich zur deaktivierten PE. Neue Konstanten wurden durch die partielle Evaluierung nicht gefunden, hier wurden schon alle durch die Konstantenpropagation bzw. -faltung eliminiert.

8.1.5 Umwandlung von Kontrollstrukturen in Multiplexer

Im folgenden Test wird die Optimierung deaktiviert, die Verzweigungen im Kontrollpfad in Multiplexer im Datenpfad umwandelt, wie in Abschnitt 6.1 beschrieben. Abb. 8.1.12 bis 8.1.14 zeigen die Ergebnisse nach der Platzierung und Verdrahtung.

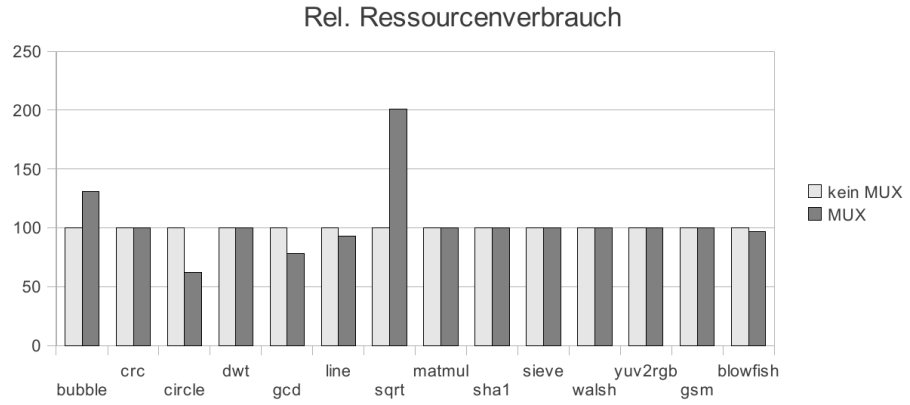


Abbildung 8.1.12: Relativer Ressourcenverbrauch mit und ohne Umwandlung von Kontrollfluss in Multiplexer

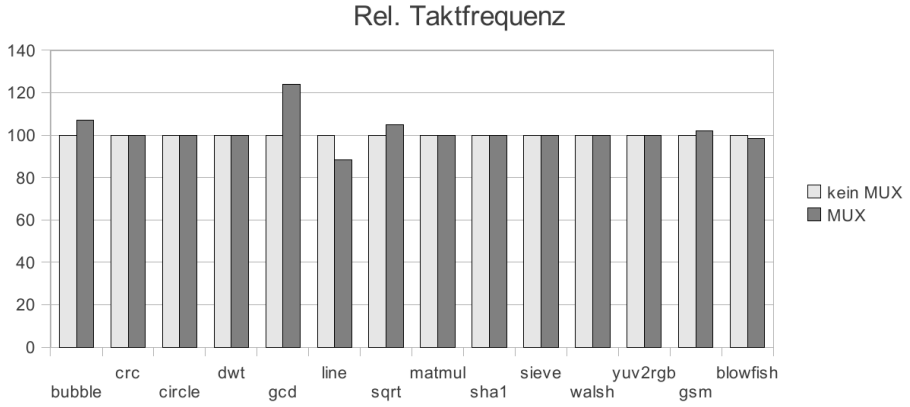


Abbildung 8.1.13: Relative Taktfrequenz mit und ohne Umwandlung von Kontrollfluss in Multiplexer

Durch die Umwandlung fallen Kontrollstrukturen weg und Grundblöcke können zusammengefasst werden, was weitere Optimierungen ermöglicht und die Parallelität erhöht. Daher wirkt sich diese Transformation in den meisten Tests positiv auf die Laufzeit aus. Allerdings erhöht sich durch die gestiegene Anzahl von simultanen Berechnungen auch die Zahl der Funktionseinheiten, was einen höheren Ressourcenverbrauch und eine niedrigere Taktfrequenz zur Folge hat. Ohne die Eliminierung des Kontrollflusses in den innersten Schleifen kann kein funktionales Pipelining angewendet werden. Da im *sqrt*-Benchmark das Pipelining die Ergebnisse verschlechterte, hat hier auch die Erzeugung der Multiplexer sehr negative Auswirkungen auf den Ressourcenverbrauch. Ohne den Multiplexern ist auch kein Pipelining möglich, wodurch der Ressourcenverbrauch nicht so stark angestiegen

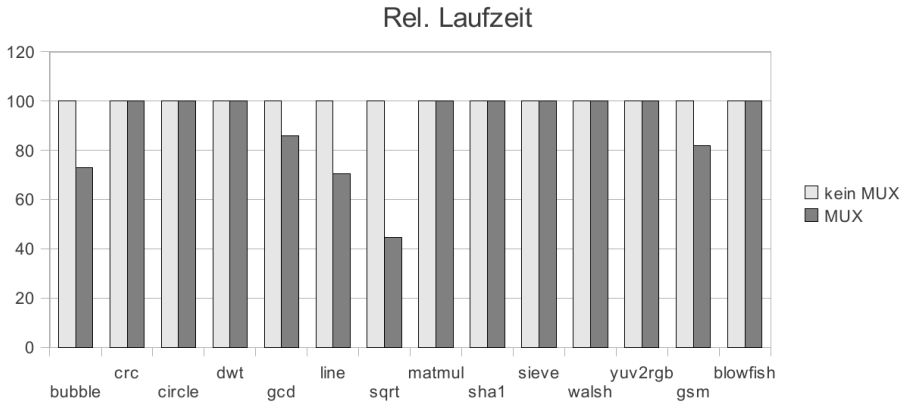


Abbildung 8.1.14: Relative Laufzeit in Takten mit und ohne Umwandlung von Kontrollfluss in Multiplexer

wäre.

8.1.6 Vergleich mit Softprozessor

Der TransC-Compiler setzt Programme in der Regel mit Hilfe eines anwendungsspezifischen Automaten um, der die Ausführung der Operationen steuert. Eine alternative Möglichkeit stellt die Implementierung eines programmierbaren Controllers dar mit einem Rechenwerk, das auf die Datentypen der Applikation angepasst ist. Das Rechenwerk kann ähnlich wie bei digitalen Signalprozessoren mit komplexeren anwendungsspezifischen Operationen erweitert werden. Die Vorteile des Automaten liegen in der höheren Taktfrequenz und der geringeren Anzahl von Taktzyklen zur Abarbeitung des Programms, da die erzeugte Schaltung - anders als ein Prozessor - genau an die Applikation angepasst ist. Je länger das Programm ist, desto mehr Zustände erhält der Automat und umso komplexer wird das Rechenwerk aufgrund der multiplexer-basierten Architektur, wodurch sich der Ressourcenverbrauch erhöht und die Taktfrequenz sinkt. Beim Controller hingegen sind beide Parameter konstant, längere Programme benötigen lediglich einen größeren Instruktionsspeicher. Ab einer bestimmten Größe verbraucht die spezifische Schaltung somit mehr Ressourcen und durch die niedrigere Taktfrequenz ist ab einem bestimmten Punkt auch die Rechenzeit insgesamt länger als auf einem Controller.

In den folgenden Testreihen werden automaten-basierte Implementierungen welchen auf einem programmierbaren Steuerwerk gegenübergestellt. Dabei wird ermittelt, wie groß der Vorteil hinsichtlich Laufzeit und Ressourcenverbrauch ist und abgeschätzt, ab wann sich der Einsatz eines programmierbaren Steuerwerks lohnt. Abbildung 8.1.15 zeigt die Taktfrequenz, Größe und Laufzeit der erzeugten TransC-Schaltungen von synthetisch erzeugten Benchmarks in Abhängigkeit von deren Programmgröße. Die Benchmarks führen eine zufällig generierte Folge von 32-Bit Daten- und Kontrolloperationen durch. Die Programmgröße wird in der Anzahl der Zustände gemessen, die die von TransC-Compiler umgesetzten FSM aufweisen. Die Testprogramme sind künstlich, da so die Breite des Datenpfades immer kon-

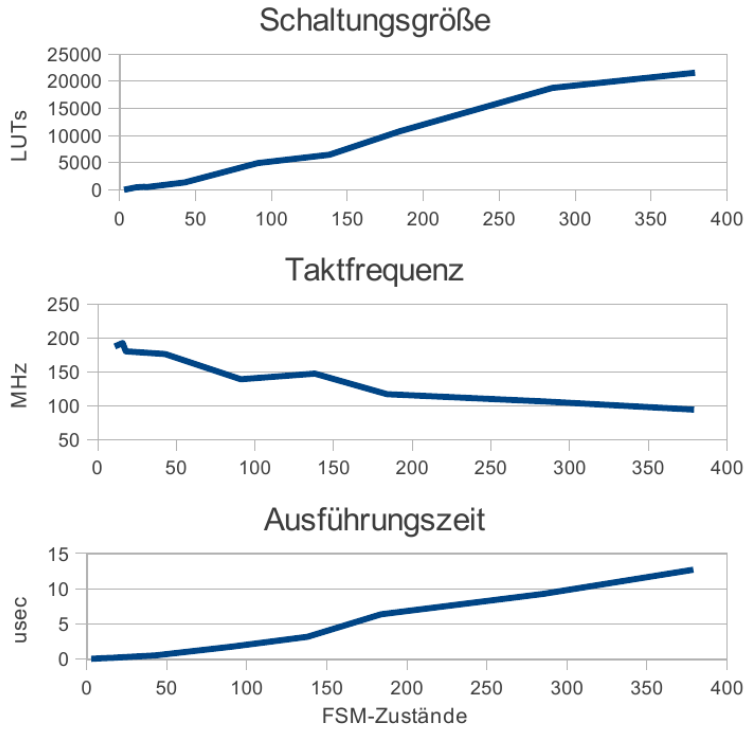


Abbildung 8.1.15: Schaltungseigenschaften abhängig von der Zahl der FSM-Zustände

stant ist und somit die Ausführungseinheiten unabhängig von der Programmlänge immer die gleiche Latenz aufweisen. Man kann erkennen, dass die Schaltungsgröße proportional zur Größe der FSM zunimmt. Das liegt daran, dass bei einem komplexeren Steuerpfad mehrere Funktionseinheiten des gleichen Typs allokiert werden, da sonst die multiplexerbasierte Datenpfadarchitektur zu groß wird. Die Taktfrequenz nimmt dabei ab. Die Laufzeit nimmt nicht nur aufgrund des größeren Programms zu, sondern auch wegen der sinkenden Taktfrequenz, da die Operationen mit einer niedrigeren Rate abgearbeitet werden.

Selbst bei Programmen, die mit einer Schaltungsgröße von 25000 Slice-LUTs größere FPGAs füllen (Xilinx XC6SLX75T besitzt 46648 Slice-LUTs [55]), kann die FSM noch mit einer Rate von etwa 100 MHz getaktet werden, was immer noch der Frequenz von schnellen Soft-CPUs entspricht. Aufgrund des spezialisierten Datenpfades und der damit verbundenen parallelen Verarbeitung von mehreren Operationen ist die Laufzeit dennoch um einige Faktoren geringer. Allerdings belegt die FSM-Schaltung dann wesentlich mehr Ressourcen als eine Soft-CPU.

Um genau zu ermitteln, ab wann ein Soft-Prozessor hinsichtlich Fläche und Laufzeit die besseren Ergebnisse liefert, werden die generierten TransC-Schaltungen mit der SHARF-CPU [13] verglichen. Die SHARF-CPU ist ein Softprozessor, der im Institut für Rechnertechnologie an der Technischen Universität Hamburg-Harburg (TUHH) entwickelt wurde. Eine besondere Eigenschaft dieser CPU ist die strikte Trennung von Kontroll- und Anwendungsdaten, wodurch sich der Datenpfad leicht an die jeweilige Anwendung anpassen lässt. Mit einer 32-Bit Integer-ALU benötigt die CPU auf dem Spartan-6 etwa 900 Slice-LUTs

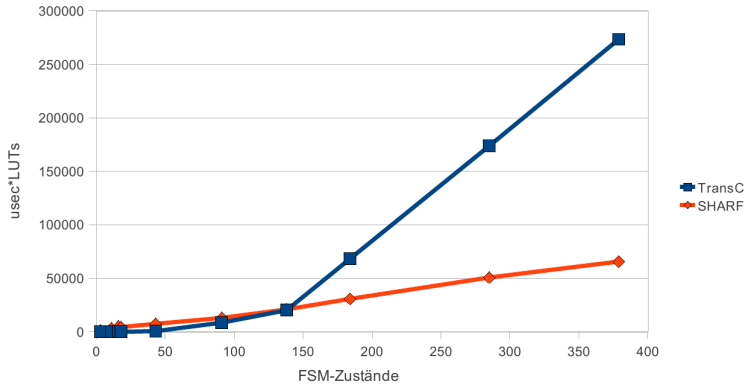


Abbildung 8.1.16: Vergleich von FSM und Soft-CPU abhängig von der Zahl der FSM-Zustände

und läuft mit 80 MHz. Ist eine Anwendung nicht zeitkritisch oder erfüllt die Ausführung auf der CPU die geforderten Zeitschranken, so lohnt sich der Einsatz einer FSM ab einer Größe von 900 Slice-LUTs nicht mehr.

Abbildung 8.1.16 zeigt das Produkt aus Fläche und Laufzeit der generierten Benchmarks auf der Soft-CPU und auf den FSMs. Die FSM kann im Vergleich zur CPU mehrere Operationen parallel ausführen, weshalb die Laufzeit bei den kleineren Programmen niedriger ausfällt. Hinzu kommt, dass die CPU hier mehr Schaltungsressourcen beansprucht. Mit zunehmender Schaltungsgröße sinkt jedoch die Taktfrequenz und der Ressourcenverbrauch steigt so stark an, dass ab etwa 140 FSM-Zuständen der Einsatz einer CPU günstiger hinsichtlich Laufzeit und Flächenverbrauch ist. Bei anderen Testprogrammen und bei anderen CPUs kann dieser Wert allerdings stark abweichen, da u.a. auch die Breite des Datenpfades oder die allokierten Funktionseinheiten eine Rolle spielen. Für größere Applikationen wird die optimale Lösung aus einer Mischform von CPUs und FSMs bestehen, wobei die zeitkritischen oder häufig ausgeführten Routinen in Form einer FSM implementiert werden und der Rest auf einer CPU ausgeführt wird.

8.1.7 Fließkommaoperationen

Die folgenden Testprogramme zeigen die Effizienz von Algorithmen, die auf Fließkommaoperationen zurückgreifen. Da Fließkommadatentypen standardmäßig nicht in TransC vorhanden sind, gibt es für die Implementierung zwei Möglichkeiten. Die erste Möglichkeit besteht darin, Fließkommaoperationen mit normalen TransC-Funktionen nachzubilden, die die entsprechenden Berechnungen mit Hilfe von ganzzahligen Datentypen wie *int* oder *uint* durchführen (Softfloat). Daneben können spezialisierte Einheiten in VHDL implementiert werden, deren Schnittstelle kompatibel zu TransC-Funktionen ist und die anhand eines Prototypen und der Angabe der Latenz bei Pipelinefähigkeit dem Programm bekannt gegeben werden, ähnlich wie in Abschnitt 4.1.1 gezeigt.

Als Basis für den Fließkommabenchmark dient ein Algorithmus *dfsin* zur Berechnung von Sinuswerten auf der Basis von 64-Bit Fließkommazahlen, der aus der CH-Stone-

Sammlung [46] entnommen wurde. Das Programm führt die Berechnung des Sinus mit Hilfe von grundlegenden Fließkommaoperationen durch, die mit Hilfe von Ganzzahloperationen implementiert sind. *dfsin* wurde auf TransC portiert und anschließend mit allen zur Verfügung stehenden Optimierungen kompiliert.

Für den Referenzbenchmark wurden grundlegende Fließkommaeinheiten wie die Multiplikation, Addition/Subtraktion, Division und die Integerkonvertierung mit Hilfe des Xilinx Floating-Point Operator v5.0 [55] erstellt. Damit diese Komponenten von TransC aus aufgerufen werden können, wurden sie in einen Wrapper integriert, der die selbe Schnittstelle besitzt wie die entsprechende TransC-Funktion. Außer die Division waren alle Module pipelinefähig. Anschließend wurden die ursprünglichen Funktionsaufrufe durch die neuen ersetzt, so dass in der generierten Schaltung die Fließkommaeinheiten statt der Softfloat-Funktionen benutzt wurden. Tabelle 8.1 vergleicht den Ressourcenverbrauch, sowie die maximale Taktfrequenz und die Ausführungszeit beider Varianten miteinander.

Tabelle 8.1: Vergleich von Fließkomma-Berechnungen mit in TransC beschriebenen und generierten Einheiten

	LUTs	DSP48	f (MHz)	t (Takte)
mit Xilinx Einheiten	2466	19	48	6411
mit TransC Einheiten	14584	51	63	55803

Durch die Verwendung der hochoptimierten Fließkommaeinheiten des Floating-Point Operators konnte die Laufzeit um den Faktor 8,7 gesenkt werden, dafür verschlechterte sich die Taktfrequenz geringfügig um 15 MHz (Faktor 1,3). Der Ressourcenverbrauch der TransC-Komponenten war ebenfalls erheblich höher, etwa Faktor 5,9 bei den Slice-LUTs und 2,5 bei den DSP48-Einheiten.

Die in TransC implementierte Fließkommamultiplikation und -addition benötigte etwa 20 Taktzyklen, während die mit dem Floating-Point Operator erstellten Einheiten 6 bzw. 8 Taktzyklen benötigten. Das liegt an den vielen Verzweigungen im Kontrollfluss, den die Funktionen in *dfsin* besitzen. Da ein Grundblock im TransC-Zwischencode maximal nur zwei Nachfolger besitzen darf, müssen viele Taktzyklen für die Wahl des korrekten Zweiges aufgebracht werden. Wenn ein Block mehrere Nachfolger besitzen könnte und die entsprechenden Sprungbedingungen parallel ausgewertet werden könnten, ließe sich hier Laufzeit einsparen.

Die Sinuswerte werden in dem Programm iterativ mit einer Schleife berechnet, die abbricht, wenn das Ergebnis eine bestimmte Genauigkeit erreicht hat. Insgesamt ermittelt das Programm 36 Sinuswerte, womit die Variante mit den Xilinx-Einheiten 178 Takte für einen Wert benötigt und eine Schleifeniteration 17 Takte dauert. In späteren Optimierungen könnte das Synthesetool die Schleife so modifizieren, dass sich die einzelnen Sinuswerte überlappend in einer Pipeline berechnen ließen, um die Ausführungszeit weiter zu senken und die Auslastung der Ausführungseinheiten zu erhöhen.

8.2 Systeme

In den nächsten Abschnitten werden nicht nur einzelne Benchmarks und Anwendungen getestet, sondern komplette, in sich abgeschlossene Systeme erstellt, die auf einem FPGA implementiert werden und über Schnittstellen mit Peripheriegeräten kommunizieren.

8.2.1 Sensor

Bei dem hier vorgestellten System ist ein Beschleunigungssensor vom Typ MXC6202 auszulesen und der Wert auf einem zweizeiligen Textdisplay darzustellen. Da der Sensor neben der Beschleunigung auch die Umgebungstemperatur messen kann, soll mit einem Taster der Betriebsmodus umgestellt werden können, so dass statt der Beschleunigung die Temperatur angezeigt wird. Abbildung 8.2.1 zeigt ein Blockschaltbild mit den Bezeichnungen für die Schnittstellen.

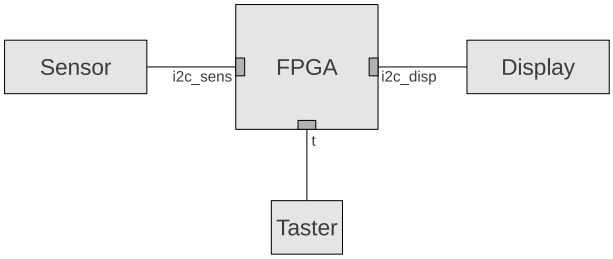


Abbildung 8.2.1: Blockschaltbild für die Sensoranwendung

Für diese Anwendung ist kein großer Rechenaufwand erforderlich, daher wäre auch eine Realisierung mit einem einfachen Mikrocontroller möglich, der durch entsprechende Schnittstellen mit der Peripherie kommuniziert. Hier soll gezeigt werden, dass mit TransC die Implementierung auf einem FPGA in einer relativ kurzen Entwicklungszeit im Vergleich zu Hardwarebeschreibungssprachen möglich ist.

TransC-Implementierung

Die beiden Peripheriegeräte sind über zwei separate I2C-Busschnittstellen [94] als Slave mit dem FPGA verbunden. Somit ist das FPGA der Busmaster und erzeugt die Steuersignale unter Berücksichtigung zeitlicher Randbedingungen. Für den I2C-Bus wird das Taktsignal `sck` sowie das Datensignal `sda` benötigt. Da Sensor und Display nicht an einem gemeinsamen Bus hängen, benötigt jede Komponente eine eigene Schnittstelle, also `sens_sck` und `sens_sda` sowie `disp_sck` und `disp_sda`. Zusätzlich gibt es einen Tastereingang `t`. Weitere Steuer- oder Datensignale sind nicht vorhanden, womit der Prototyp für den obersten TransC-Prozess feststeht:

```
1 process void main(
2             istream<uint1> t,
3             ostream<uint1> sens_sck,
4             ostream<uint1> sens_sda_in,
```

```
5             istream<uint1> sens_sda_out,
6             ostream<uint1> disp_sck_disp,
7             ostream<uint1> disp_sda_in,
8             istream<uint1> disp_sda_out
9 );
```

Sämtliche Kommunikationsleitungen sind mit asynchronen Streams realisiert. Die Datensignale für den I2C-Bus sind in Ein- und Ausgang `sda_in` und `sda_out` aufgetrennt worden, da über einen Stream nicht bidirektional kommuniziert werden kann. Beim I2C-Bus sind die Signale über Pull-up-Widerstände mit der Versorgungsspannung V_{DD} verbunden. Die daran angeschlossenen Geräte haben Open-Collector-Ausgänge, wodurch sich zusammen mit den Pull-up-Widerständen eine UND-Schaltung ergibt. In TransC wird die Verbindung eines I2C-Signals mit Masse durch eine 0 und der hochohmige Zustand durch eine 1 dargestellt. Der Tastereingang `t` wird durch Betätigung des Tasters auf 0 gezogen, standardmäßig liegt an ihm eine 1 an.

Innerhalb des Prozesses gibt es zwei Betriebsarten, `ACCEL` für die Beschleunigung und `TEMP` für die Temperatur. Per Tastendruck werden diese umgestellt. Somit wird ein Prozess `switchMode` für die Überwachung von `t` und die Einstellung des Betriebsmodus erstellt, der von `main` aus aufgerufen wird. Anfangs besitzt er die Betriebsart `ACCEL`, durch eine negative Flanke an `t` wird diese umgeschaltet. Die folgende Auflistung zeigt den `switchMode`-Prozess, der Wartebefehl und die zusätzliche Abfrage dient zum Entprellen des Schalters.

```
1  const bool ACCEL = 0;
2  const bool TEMP = 1;
3  process void switchMode(istream<uint1> t, ostream<bool> mode=ACCEL) {
4     uint1 pt = 1;
5     while(1) {
6        if(pt==1 && t==0) {
7           delay(30);
8           if(t==0) mode = !mode;
9        }
10       pt = t;
11    }
12 }
```

Die eigentliche Kommunikation mit einem I2C-Bus wird von der Funktion `i2c` durchgeführt, in der der Controller implementiert ist.

```
1 uint9 i2c( ostream<uint1> scl,
2            ostream<uint1> sda,
3            istream<uint1> sda_in,
4            uint8 val, uint3 param);
```

Die asynchronen Streams werden direkt mit der Busschnittstelle der Aufrufer verbunden. `val` übergibt den Wert, der auf die Leitung gelegt werden soll, und `param` die Parameter für den Bus, z.B. ob ein Stop-Bit oder ein Start-Bit gesetzt werden soll, usw. Der gelesene 8-Bit Wert wird zusätzlich mit dem Acknowledge-Bit zurückgegeben, weshalb das Ergebnis eine Breite von 9-Bit aufweist.

Für die Sensorkommunikation ist die Funktion `readSens` zuständig, die nach Aufruf mit Hilfe einer Instanz der `i2c`-Funktion den Messwert ausliest und zurückgibt. Der Messwert ist entweder die Beschleunigung oder die Temperatur, daher muss der Betriebsmodus ebenfalls als Argument übergeben werden. Die Beschleunigung wird sowohl in x- als auch in y-Richtung gemessen und kann negativ sein, weshalb es zwei Rückgabewerte vom Typ Integer gibt.

```
(int,int) readSens(
            ostream<uint1> scl,
            ostream<uint1> sda,
            istream<uint1> sda_in,
            bool mode);
```

Aufrufe der `i2c`-Funktion innerhalb `readSens` zur Kommunikation mit dem Sensor sehen wie folgt aus:

```
...
x = i2c(scl, sda, sda_in, I2C_ACK | 0xff) >>1;
x <<= 8;
x |= i2c(scl, sda, sda_in, I2C_ACK | 0xff) >>1;
y = i2c(scl, sda, sda_in, I2C_ACK | 0xff) >>1;
y <<= 8;
y |= i2c(scl, sda, sda_in, I2C_STOP | 0xff) >>1;
...
```

Bevor die gelesenen Messwerte am Display angezeigt werden können, müssen sie in einen String umgewandelt werden, was mit der Funktion `int2str` realisiert wird. `int2str` konnte von einer bestehenden C-Funktion übernommen werden und ist im Folgenden gelistet. Da die NUMERIC_STD-Bibliothek keine Divisionsoperatoren unterstützt, musste die Division als Funktionsaufruf realisiert werden. Im TransC-Programm ist nur der Prototyp bekannt, die Funktionalität selber ist in einem VHDL-Modul hinterlegt. Die umgewandelte Zahl wird als ASCII-Zeichenkette im Array `str` gespeichert.

```
process (short,short) tc_div(short dividend, short divisor);

void int2str(int num, int size, char str[32]) {
   short q,r;
   int i;
   bool neg=0;

   if(num<0) {
      neg = 1; num = -num;
   }

   i = size;
   do {
      (q,r) = tc_div(num, 10);
      str[--i] = r + '0';
      num = q;
```

```
17      } while(num>0 && i<size );
18
19      if (neg) str[--i] = '-';
20      while(i>0) str[--i] = ' ';
21  }
```

Die erzeugten Strings müssen je nach Betriebsart als Text an das Display übergeben werden, wofür die Funktion `display` zuständig ist. Ihr wird der I2C-Bus des Displays als Stream sowie die anzuzeigenden Strings und Informationen zum Betriebsmodus überreicht, z.B. welche Displayzeile verwendet werden soll. Intern wird hier ebenfalls die Funktion `i2c` aufgerufen.

```
1  void display(
2          ostream<uint1> scl,
3          ostream<uint1> sda,
4          istream<uint1> sda_in,
5          int val,
6          int line,
7          bool mode);
```

Die Funktionen und Prozesse werden im TransC-Hauptprozess integriert. Sowohl das Auslesen als auch die Konvertierung und die Eingabe werden sequentiell durchgeführt. Der Taster wird permanent überwacht, weshalb `switchMode` als Prozess implementiert ist. Der Betriebsmodus darf sich innerhalb einer Schleifeniteration nicht ändern, so dass er am Anfang der Schleife abgespeichert wird. Mit Hilfe von `delay` wird das Aktualisierungsintervall eingestellt.

```
1  process void main( ... ) {
2     nstream<bool> smode;
3     bool mode;
4     int x,y;
5     char buf[32];
6
7     switchMode(t, smode);
8     while(1) {
9        mode = smode;
10       (x,y)=readSens(sens_scl, sens_sda, sens_sda_in, mode);
11       int2str(x, 7, buf);
12       display(disp_scl, disp_sda, disp_sda_in, x, 1, mode);
13       if(mode==ACCEL) {
14          int2str(y, 7, buf);
15          display(disp_scl, disp_sda, disp_sda_in, y, 2, mode);
16       }
17       delay(...);
18    }
19 }
```

VHDL

Ein Großteil der Anwendung konnte in TransC implementiert werden, jedoch wird das TransC-Modul noch in ein oberstes Systemmodul, das in VHDL beschrieben ist, eingebettet. Das Systemmodul enthält die Ein- und Ausgänge für die I2C-Anschlüsse sowie einen Eingang für den Taster. `sda` ist hier als `inout`-Signal deklariert, das direkt an das `sda_in`-Signal des TransC-Moduls durchgereicht wird. Das Problem mit den Open-Collector-Ausgängen wird durch hochohmige Verbindungen gelöst. Wenn am jeweiligen Steuersignal `sda` oder `scl` vom TransC-Modul eine 0 anliegt, so wird auch eine 0 am Systemausgang angelegt, bei einer 1 allerdings ein Z, wodurch der Ausgang vom Bus getrennt wird.

Neben dem TransC-Modul und der Konvertierungslogik für die Signale beinhaltet das System-Modul eine DCM (engl. *Digital Clock Manager*) für die Einstellung der Taktfrequenz sowie einen Resetgenerator, der nach der FPGA-Konfiguration ein Reset-Signal erzeugt. Für das `run`-Signal zum Starten des generierten TransC-Moduls wird lediglich das Resetsignal um einige Takte verzögert. Das `done`-Signal wird nicht benötigt, da es sich bei `main` um einen zyklischen Prozess handelt.

Aufwand

Einzelne Module wie `int2str` usw. konnten in C geschrieben und auf einem normalen Prozessor getestet und debugged werden. Hardwarenahe Module wie beispielsweise der `i2c`-Prozess wurden im VHDL-Simulator verifiziert. Neben den I2C-Controllern wurden I2C-Slaves implementiert, um in der Testbench entsprechende Antwortsignale generieren zu können. Die Gesamtfunktion des TransC-Hauptprozesses konnte somit in einer Testbench überprüft werden, anschließend folgte die Integration im VHDL-Systemmodul und die Implementierung im FPGA. Aufgrund des sehr geringen VHDL-Anteils am Gesamtprojekt war die Implementierung ähnlich aufwändig wie die auf einem Mikrocontroller. Am meisten Zeit nahm die Entwicklung der I2C-Module in Anspruch, mit einer entsprechenden Bibliothek an Schnittstellen und Controllern würde dieser Teil allerdings wegfallen. Ein Nachteil im Vergleich zum Mikrocontroller lag in den langen Rechenzeiten von ISE WebPACK während der Low-Level-Synthese. Der Ressourcenverbrauch des Gesamtsystems auf einem Spartan-3 1400A betrug 878 Slices, die das FPGA zu ca. 7 % füllten. Die Taktfrequenz lag bei 129 MHz.

8.2.2 Audio-Effektgerät

Das Audio-Effektgerät ist ein System zur Modifikation von Audiosignalen wie z.B. die Anhebung oder Absenkung von bestimmten Frequenzbereichen, die Verzerrung des Signals oder die Einschränkung des Dynamikumfangs.

Dafür wird ein FPGA mit einem sogenannten Audio-Codec verbunden. An dem Codec befindet sich eine analoge Audioquelle, die digital gewandelt und als Datenstrom an das FPGA geschickt wird. Die Signalverarbeitung findet im FPGA statt, wo Effekte zugefügt

oder Filter angewendet werden. Anschließend werden die digitalen Daten an den Codec zurückgesendet, der sie wieder in ein analoges Signal konvertiert. Für den Codec wird ein Stereo-Baustein vom Typ SSM2603 verwendet. Zusätzlich sind über einen weiteren Analog-Digital-Konverter (ADC) vom Typ MCP3208 Potentiometer am FPGA angeschlossen, mit denen ein Benutzer Einfluss auf Effektparameter und somit auf den Klang nehmen kann.

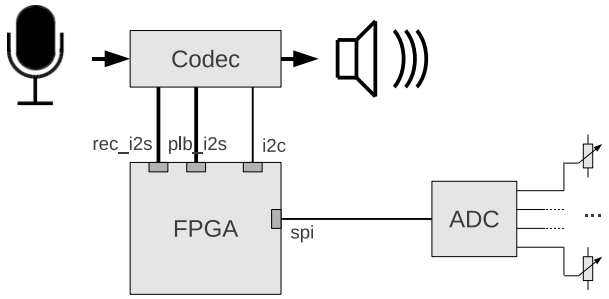


Abbildung 8.2.2: FPGA als Audio-Effektgerät

Die Übertragung des digitalen Audiosignals zwischen Codec und FPGA wird mit dem I2S-Protokoll[93] durchgeführt, wofür drei Leitungen benötigt werden, `SCK`, `SD` und `WS`. `SD` (Serial Data) überträgt synchron zum Taktsignal `SCK` (Serial Clock) seriell die einzelnen Audio-Samples. Über `WS` (Word Select) wird angegeben, ob es sich um ein Sample für den linken oder rechten Kanal handelt. Hier arbeitet das FPGA als Slave, d.h. der Codec erzeugt die Steuersignale `SCK` und `WS`, das FPGA muss die Daten entsprechend einlesen bzw. ausgeben. Da es sowohl einen Audioeingang als auch einen Ausgang gibt, sind zwei I2S-Schnittstellen vorhanden. Die Konfiguration des Codec findet über eine I2C-Schnittstelle statt, die aus dem im vorigen Abschnitt beschriebenen Sensorsystem übernommen werden kann.

Der ADC für die Potentiometer wird ebenfalls vom FPGA aus gesteuert. Der ADC stellt die digitalen Daten mittels SPI (Serial Peripheral Interface) zur Verfügung und besitzt acht analoge Eingänge, wovon vier durch die Regler in Benutzung sind. Zur Kommunikation dienen die Signale `ADC_CS`, `ADC_DIN`, `ADC_DOUT` und `ADC_CLK`. `ADC_CLK` ist das Taktsignal zur synchronen Datenübertragung. Mit `ADC_CS` wird der ADC aktiviert und anschließend über `ADC_DOUT` die Adresse des auszulesenden Analogeingangs eingetaktet. Anschließend kann die Spannung vom jeweiligen Analogeingang seriell über `ADC_DIN` ausgelesen werden.

Die Abtastrate des Audiosignals soll auf 48 kHz und die Größe eines Abtastwertes auf 24-Bit gesetzt werden. Es muss gewährleistet sein, dass die Verarbeitungsrate der Samples im FPGA höher als die Abtastfrequenz ist, da die ankommenden Daten sonst nicht schnell genug angenommen werden können. Dementsprechend ist hier der Rechenaufwand je nach Effekt im Vergleich zu der Sensoranwendung sehr hoch, so dass sich dieses System mit dem Einsatzgebiet der digitalen Signalprozessoren überschneidet.

TransC-Implementierung

Anhand der am FPGA angeschlossenen ICs lässt sich der Prototyp des obersten TransC-Prozesses bestimmen, der ebenfalls wie in Abschnitt 8.2.1 beschrieben in eine VHDL-Umgebung zu integrieren ist.

```
1  void main(
2        //audio in
3        istream<uint1> rec_sck,
4        istream<uint1> rec_ws,
5        istream<uint1> rec_sd,
6
7        //audio out
8        istream<uint1> plb_sck,
9        istream<uint1> plb_ws,
10       ostream<uint1> plb_sd,
11
12       //codec config
13       ostream<uint1> scl,
14       ostream<uint1> sda,
15       istream<uint1> sda_in,
16
17       //adc
18       ostream<uint1> adc_cs,
19       ostream<uint1> adc_clk,
20       istream<uint1> adc_din,
21       ostream<uint1> adc_dout,
22 );
```

Die Konfigurierung des Codec wird mit einer Funktion `config` durchgeführt, die ein `i2c`-Modul enthält und nach dem Reset einmalig gestartet wird. Anschließend beginnt der eigentliche Betrieb des Effektgerätes. Der erste TransC-Prozess hierfür ist `rec`, der für das Einlesen des seriellen Audiosignals zuständig ist. Hier werden `rec_sck` sowie `rec_ws` abgetastet und dementsprechend das serielle Signal auf `rec_sd` zu einem 24-Bit Wort zusammengefasst. Dieses Wort wird in einen synchronen Ausgabestream `sample` geschrieben, auf dem sowohl die Samples des linken als auch des rechten Kanals ausgegeben werden. Daher wird festgelegt, dass der erste Wert nach einem Reset zum rechten Kanal gehört. Aufgrund der Größe der Abtastwerte hat `sample` eine Breite von 24-Bit.

```
1  void rec(istream<uint1> rec_sck,
2           istream<uint1> rec_ws,
3           istream<uint1> rec_sd,
4           sostream<int<24> > sample
5         );
```

Zur Pufferung der Daten soll `sample` im Hauptmodul mit einer ausreichend großen FIFO verbunden werden. Neben `rec` gibt es einen Prozess `play`, der ein 24-Bit Sample wieder serialisiert und an den Codec ausgibt. Da das FPGA bei der Übertragung im Slave-Modus arbeitet, müssen in `play` die Signale `plb_sck` und `plb_ws` abgetastet und dementsprechend

das Sample auf den seriellen Ausgang gelegt werden. Nach dem Reset darf das erste Sample erst ausgegeben werden, wenn `rec_ws` den rechten Kanal anfordert, da der erste in `rec` abgetastete Wert ebenfalls vom rechten Kanal stammt. Die Samples, die an `play` ausgegeben werden, sollen ebenfalls über eine FIFO gepuffert werden. Wenn keine Daten an `sample` anliegen, so wird eine 0 ausgegeben.

```
1  void play(istream<uint1> plb_sck,
2            istream<uint1> plb_ws,
3            ostream<uint1> plb_sd,
4            sistream<int<24> > sample
5           );
```

Mit Hilfe der Potentiometer kann zur Laufzeit Einfluss auf die Effekte genommen werden. Die Steuerung des ADCs übernimmt der Prozess `control`, der etwa 10 mal pro Sekunde den Status der vier Potentiometer aktualisieren soll. Die Kommunikation nach außen mit dem ADC findet über die vier asynchrone Streams statt, die Positionen der Regler wird der Anwendung ebenfalls über vier asynchrone Streams zur Verfügung gestellt. Der ADC hat zwar eine Auflösung von 12-Bit, für die Darstellung der Reglerposition reichen allerdings 8-Bit aus.

```
1  void control(
2        ostream<uint1> adc_cs,
3        ostream<uint1> adc_clk,
4        istream<uint1> adc_din,
5        ostream<uint1> adc_dout,
6
7        ostream<uint8> ctrl0,
8        ostream<uint8> ctrl1,
9        ostream<uint8> ctrl2,
10       ostream<uint8> ctrl3 );
```

Den Prozessen für die eigentlichen Berechnung der Effekte steht somit der Sample-Ausgang des `rec`-Prozesses sowie der Sample-Eingang von `play` und die vier asynchronen Streams für die Reglerstellungen zur Verfügung. Verbindet man die synchronen Streams von `rec` und `play` über eine FIFO miteinander, so werden die Audiodaten vom Eingang direkt an den Ausgang weitergeleitet. Über die Multiplikation mit einem Festkommawert, der von einem der Potentiometer abhängt, kann beispielsweise die Lautstärke verändert werden. Eine komplexere Anwendung ist ein FIR-Filter (engl. *Finite Impulse Response*), mit dem der Frequenzgang manipuliert werden kann.

Das folgende Beispiel zeigt einen Ausschnitt aus einem FIR-Filter, dessen Filterverhalten von festen Koeffizienten abhängig ist. Der Filter wird direkt über FIFOs mit den Sample-Ein- und -ausgängen verbunden. Die Pragmas geben an, dass die Arrays auf Register abgebildet (`arrayToReg`), bzw. dass die Schleife vollständig abgerollt werden soll (`unroll`).

```
1  const int N = 512;
2  process void fir(sistream<int24> din, sostream<int24> dout) {
```

```
3     #pragma arrayToReg sample_buffer
4     #pragma arrayToReg coefficient
5     short sample_buffer[N];
6     int accu;
7     int i;
8     const short coefficient[N] = {...};
9     while(1) {
10       for (i = N-1; i > 0; --i) {
11          #pragma unroll full
12          sample_buffer[i] = sample_buffer[i-1];
13       }
14       sample_buffer[0] = din;
15       accu = 0;
16       for (i = 0; i < N; ++i) {
17          #pragma unroll full
18          accu += sample_buffer[i] * coefficient[i];
19       }
20       dout = accu;
21    }
22 }
```

Neben Filtern ist es möglich, beliebige in C geschriebene Effektfunktionen zu portieren und in die Anwendung zu integrieren. Der Test dieser Funktionen kann vollständig auf dem Entwicklungsrechner z.B. mit Audiodateien durchgeführt werden. Vorlagen für Audioeffekte sind auf [30] zu finden. Aufgrund der parallelen Ressourcen eines FPGAs können hier mehrere Effektmodule seriell oder parallel miteinander verschaltet werden. Je mehr Module allerdings hintereinander gesetzt werden, desto höher ist die Latenz zwischen Audio-Ein- und -Ausgang.

VHDL

Wie in dem Sensorsystem ist auch hier ein oberstes VHDL-Modul zu implementieren. Die Daten- und Steuerleitungen können bis auf die I2C-Schnittstelle einfach weitergeleitet werden. Die DCM für das Taktsignal sowie der Resetgenerator sind hier ebenfalls zu integrieren.

Aufwand

Die Implementierung der I2S-Schnittstellen hat am meisten Zeit in Anspruch genommen. Die Beschreibung der Algorithmen war relativ einfach, da hier der Funktionstest vollständig auf dem PC durchgeführt werden konnte.

Kapitel 9

Zusammenfassung und Ausblick

Durch die ständig wachsende Anzahl von Logikzellen und festverdrahtenten Komponenten in FPGAs erschließt diese Technologie ständig neue Einsatzgebiete. Die ursprünglichen Anwendungsfelder erweiterten sich vom Test digitaler Schaltungen über die Realisierung von Verbindungslogik bis hin zu komplexen DSP-Algorithmen. Allerdings steigt mit wachsender Komplexität auch die Schwierigkeit, diese Anwendungen zu implementieren, da FPGAs ähnlich wie ASICs mit Hardwarebeschreibungssprachen konfiguriert werden. Daher wird sowohl im kommerziellen als auch im akademischen Bereich nach Lösungen gesucht, um FPGAs, aber auch ASICs hochsprachlich zu programmieren. Die dabei entstandenen Tools zur Architektursynthese haben dennoch nicht die Akzeptanz von RTL-Synthesetools erreicht.

Zusammenfassung

In dieser Arbeit wurde die TransC-Sprache sowie das zugehörige Prozessmodell und die Synthesetools vorgestellt, die es ermöglichen, effiziente und parallel arbeitende Hardwareschaltungen aus einer verhaltensbasierten Beschreibung zu generieren. Auf diese Weise konnten FPGAs einfach und ohne tiefe Kenntnisse von Hardwarebeschreibungssprachen konfiguriert werden.

Die Eingangssprache orientiert sich an C und besitzt weniger zusätzliche Schlüsselwörter oder spezielle Pragmas im Vergleich zu anderen Dialekten wie Impulse C oder Handel-C. Die generierten Schaltungen arbeiten effizient hinsichtlich Ressourcenverbrauch und Geschwindigkeit, wobei auch die zusätzlichen Funktionen zur Modellierung mehrerer Prozesse sowie die Kommunikation zwischen diesen sehr wenig Overhead in den resultierenden Schaltungen erzeugen.

Das entwickelte Prozessmodell ist an die Gegebenheiten des FPGAs angepasst, auf eine dynamische Prozesserzeugung und eine dynamische Kommunikationsstruktur wurde verzichtet. Dafür besitzt es ausreichend viele Mechanismen zur Interprozesskommunikation und -synchronisation und kann aufgrund des geringen Protokollaufwands mit wenig Aufwand in Hardware implementiert werden. Da die gesamte Kommunikation über fest

definierte Schnittstellen geschieht und die Festlegung der Struktur nicht bereits bei der Prozessdefinition, sondern erst bei der Instantiierung erfolgt, eignet es sich auch für die Implementierung von Bibliotheksfunktionen.

Der TransC-Quellcode wird in ein Zwischenformat übersetzt, das sich an Kontroll-Datenflussgraphen orientiert, jedoch noch eine zusätzliche Kantenart besitzt, um separat den Datenfluss über Kontrollknoten hinweg darzustellen. Auf diese Weise lassen sich globale Datenabhängigkeiten repräsentieren, was Transformationen und Optimierungsalgorithmen vereinfacht, da Datenabhängigkeiten nicht mehr über Variablen dargestellt werden. Eine Vielzahl von bestehenden Optimierungen wurde auf das Zwischenformat angepasst, zusätzlich wurden neue Optimierungen implementiert, die auf einer Form der partiellen Auswertung basieren. Mit Hilfe dieser Optimierungen war es möglich, die Bitbreiten von Datenpfaden zu reduzieren oder die Parallelität von Speicherzugriffen zu erhöhen.

Aus dem Zwischenformat wurde über die Allokation, Ablaufplanung und Bindung VHDL-Code auf Registertransferebene mit Automaten erzeugt. Für grundlegende Berechnungen wird nicht auf auf Komponenten wie Addierer oder Multiplizierer zurückgegriffen, sondern soweit möglich, Operatoren aus der numerischen Standard-Bibliothek[11] verwendet. Somit hat das spätere Low-Level-Synthesetool mehr Freiraum für eigene Optimierungen, was sich neben dem Prozessmodell und den Optimierungen zusätzlich positiv auf die Effizienz der Schaltungen auswirkt.

Im Vergleich zu akademischen Compilern, die VHDL oder Verilog aus einem C-Dialekt generieren, schneidet der TransC-Compiler deutlich besser ab, bzw. ist überhaupt in der Lage, bestimmte Sprachkonstrukte zu übersetzen. Außerdem wurde gezeigt, dass sich TransC als Hochsprache nicht nur für die Implementierung von Algorithmen, sondern auch von Schnittstellen für die Kommunikation mit Peripheriegeräten unter Echtzeitbedingungen eignet.

TransC wurde zusätzlich zu dem hier beschriebenen Demonstrationsanwendungen an der Technischen Universität Hamburg-Harburg (TUHH) im Sommersemester 2010 erfolgreich im *Praktikum III Mobile Roboter* zum Antrieb und zur Regelung von Motoren eingesetzt, die in den Robotern verbaut waren. Derzeit (2011) wird es an der TUHH im Institut für Rechnertechnologie verwendet für die Implementierung von FPGA-Modulen für den Google Lunar-X-Wettbewerb. Der TransC-Compiler kann im Internet online auf der Webseite `http://cgi.tu-harburg.de/~ti6hm` verwendet werden. Dort befinden sich auch Sprachbeispiele und weitere Dokumentation.

Ausblick

Der derzeitige Stand der TransC-Sprache sowie die Implementierung des Synthesetools lässt sich noch erweitern bzw. verbessern. Dieser Abschnitt zählt die wichtigsten Punkte auf, die in zukünftigen Versionen integriert werden können.

```
void main(ostream<uint1> tx0,
          istream<uint1> rx0,
          ostream<uint1> tx1,
          istream<uint1> rx1) {

   ...
   send(tx0, 0x42)@if0;
   a = receive(rx0)@if0;
   ...

   send(tx1, 0x38)@if1;
   b = receive(rx1)@if1;
   ...

}
```

a)

```
void main(ostream<uint1> tx0,
          istream<uint1> rx0,
          ostream<uint1> tx1,
          istream<uint1> rx1) {

   SerIf if0(tx0, rx0);
   SerIf if1(tx1, rx1);
   ...
   if0.send(0x42);
   a = if0.receive();
   ...

   if1.send(0x38);
   b = if1.receive();
   ...

}
```

b)

Abbildung 9.0.1: Bisherige (a) und mögliche (b) Realisierung einer seriellen Schnittstelle

Objektorientierung Konstrukte aus der objektorientierten Programmierung wie Klassen eignen sich sehr gut für die Beschreibung von Hardware. Einfache objektorientierte Ansätze lassen sich in TransC ähnlich wie in C bereits über Datenstrukturen und Funktionen erreichen, die Algorithmen auf diesen Datenstrukturen ausführen. Allerdings würden nativ unterstützte Sprachkonstrukte weit über diese einfache Form hinausgehen. Bestimmte Schaltungen können als Klasse bereitgestellt werden, wobei Objekte dieser Klasse direkt auf Instanzen der Schaltung abgebildet werden. Diese Klassen können dann ähnlich wie Funktionen bisher entweder direkt in VHDL oder in TransC implementiert worden sein. Bei der VHDL-Variante müsste dann lediglich die Klassendeklaration in TransC bekannt sein.

In Abb. 9.0.1a wird als Beispiel die bisherige Realisierung für die Benutzung von zwei seriellen Schnittstellen gezeigt. Das Senden und Empfangen geschieht über zwei verschiedene Funktionen, wobei die Instanz und die RX- und TX-Kommunikationsleitungen bei jedem Aufruf mit angegeben werden müssen. Abbildung 9.0.1b stellt eine Möglichkeit für die Integration von objektorientierten Sprachkonstrukten dar. Hier werden zwei verschiedene UARTs (*Universal Asynchronous Receiver Transmitter*) erzeugt, wobei die Verbindungen nur einmal bei der Instantiierung angegeben werden. Da nach der Instantiierung mit den Objekten gearbeitet wird, fällt auch die Angabe der Instanz hinter dem Methodenaufruf weg.

Parallelisierung von Grundblöcken Bisher unterstützt der TransC-Compiler zwei verschiedene Arten der Parallelität. Zum einen werden Operationen innerhalb von Grundblöcken parallel ausgeführt, so weit dies die Datenabhängigkeiten und die zur Verfügung stehenden Ressourcen erlauben. Zum anderen werden Prozesse simultan ausgeführt. Die erste Form der Parallelität ist implizit, da sie unabhängig vom Programmierer geschieht,

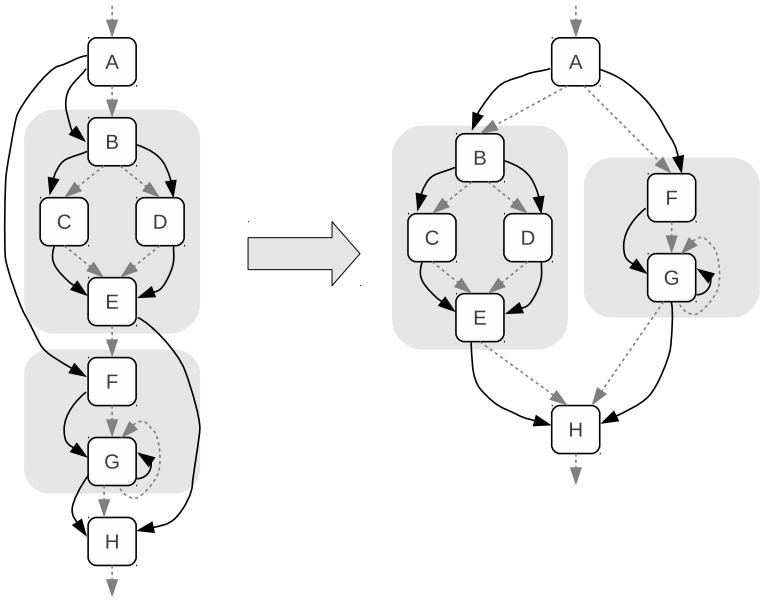


Abbildung 9.0.2: Parallelisierung von Grundblockfolgen

die zweite explizit, da sie direkt vom Programmierer angegeben wird.

Im Compiler kann eine weitere Form der Parallelität integriert werden, die ähnlich wie in Handel-C in Form von parallelen Kontrollflüssen dargestellt wird, jedoch automatisch in Programmen erkannt wird und somit implizit ist. Dies geschieht mit Hilfe der E_{DD}-Kanten. Wenn zwei Folgen von Grundblöcken in einem Kontrollfluss keine blockübergreifenden Datenabhängigkeiten untereinander aufweisen, so können sie simultan ausgeführt werden. Dies wird in Abb. 9.0.2 verdeutlicht. Die weißen Kästen stehen für Grundblöcke, die grauen gestrichelten Pfeile stellen den Kontrollfluss dar. Wenn zwischen den DFGs von zwei Blöcken E_{DD}-Kanten verlaufen, so sind diese mit einem schwarzen Pfeil verbunden. Da die Blockmengen $\{B, C, D, E\}$ und $\{F, G\}$ untereinander keine Datenabhängigkeiten aufweisen, können sie parallel abgearbeitet werden.

Wie häufig dieser Fall auftritt und welche Vorbereitungen man beispielsweise bei Arrayzugriffen treffen muss, um das Optimierungspotential zu erhöhen, kann in anschließenden Arbeiten untersucht werden. Bei abgerollten Schleifen, dessen Körper Kontrollkonstrukte beinhaltet, kann diese Optimierung erhebliche Geschwindigkeitsvorteile bringen.

Schedulingalgorithmen In dieser Arbeit kommt für die Ablaufplanung eine einfache Form des List-Scheduling ohne bestimmte Prioritätskriterien zum Einsatz. In bestimmten Fällen ist die Ausführungsreihenfolge von Operationen besonders bei knappen Ressourcen wie Speicherports nicht optimal [97]. Der Einsatz von Prioritätskriterien oder die Implementierung des Force-Directed-Scheduling bzw. von ILP-Techniken würde besonders bei Anwendungen mit einer hohen Anzahl von Operationen innerhalb eines Grundblocks die Laufzeit verbessern.

Blockübergreifende Operationen Bisher muss ein Prozess, dessen Rückgabewerte benötigt werden, immer innerhalb des aufgerufenen Grundblocks auch terminieren, da seine Rückgabewerte abgespeichert werden. Aus diesem Grund muss im Grundblock, in dem der Aufruf stattfand, gewartet werden, auch wenn die Rückgabewerte erst wesentlich später im Programm verwendet werden. Durch die Trennung von Aufruf und dem Abholen der Ergebnisse könnten Zeitschritte eingespart werden. Die Ergebnisse werden erst zu einem späteren Zeitpunkt abgeholt, wenn sie für weitere Berechnungen benötigt werden. Auf diese Weise fallen Wartezyklen weg und es können weitere Blöcke abgearbeitet werden. Dies kann auch bei Operationen angewendet werden, dessen Berechnung viele Takte dauert. Allerdings muss hier im Vergleich zu Prozessen auf Grund des fehlenden Handshakings das Ergebnis exakt zu dem Zeitpunkt abgeholt werden, an dem es am Ausgang anliegt. Die folgende Auflistung zeigt ein Beispiel, bei dem die Laufzeit durch blockübergreifende Operationen verringert werden kann. Hier wird vor der Verzweigung auf die Zuweisung von `a` durch den Prozess `p0` gewartet, obwohl `a` erst hinter der Verzweigung benötigt wird.

```
1  ...
2  a = p0(...);
3  if(...) {
4      ...
5  }
6  else {
7      ...
8  }
9  c = a * b;
10 ...
```

Umgekehrt können auch Prozessaufrufe vorgezogen gestartet werden, sobald alle Operanden bekannt sind. Auf diese Weise kann das Resultat des Prozesses bereits vorliegen, wenn es benötigt wird. Allerdings darf der Prozess keine Seiteneffekte wie Speicherzugriffe oder eine Streamkommunikation aufweisen.

Semaphore und Mutexe Wie bereits erwähnt sind Semaphore bzw. Mutexe zur Prozesssynchronisation nicht vorhanden und müssen etwa in Form des Peterson Algorithmus [86] selber implementiert bzw. als Bibliotheksfunktion hinterlegt werden. Da diese Konstrukte jedoch Primitiven für die Interprozesskommunikation darstellen, sollten diese nativ in die Sprache integriert werden, da man auf diese Weise einheitliche Aufrufe erhält und das Synthesetool diese direkt in eine entsprechende Hardwareschaltung umsetzen kann. In vielen Threadbibliotheken [22, 17] werden Mutexe als globale Variable implementiert, was sich mit den TransC-Grundsätzen widerspricht, dass die gesamte Kommunikation in der Schnittstelle sichtbar sein muss. Somit wäre denkbar, dass Mutexe etwa bei Bibliotheksfunktionen in der Argumentenliste übergeben bzw. wie bei globalen Variablen und Feldern nachträglich automatisch in die Argumentenliste mit aufgenommen werden.

Softprozessoren Ab einer gewissen Komplexität und Länge wird es aufgrund des Ressourcenverbrauchs ineffizient, Programme in Form einer FSM zu implementieren (Abschnitt 8.1.6). Die Zahl der Zustände steigt etwa linear zur Programmgröße, wodurch der Aufwand zur Auswertung der Zustände zu groß und auch die Verschaltung der Multiplexer im Datenpfad zu komplex wird. Hier lohnt es sich, lediglich kleine zeitkritische Funktionen oder häufig durchlaufene Programmausschnitte als FSM zu synthetisieren und den Rest als Programm für einen Softprozessor innerhalb des FPGAs zu kompilieren. Geeignet für solche Aufgaben sind Softprozessoren mit anpassbarem Datenpfad, die sich ggf. auch zu SIMD- und VLIW-Systemen erweitern lassen und somit auch die Rekonfigurierbarkeit des FPGAs ausnutzen. Entsprechende Softprozessoren wie die SHARF-CPU[13] oder die auf der CPU3[72] basierende CPU1C wurden am Institut für Rechnertechnologie an der Technischen Universität Hamburg-Harburg entwickelt. Hier kann in weiteren Arbeiten untersucht werden, ab wann genau sich der Einsatz eines Softprozessors lohnt. Die Entscheidungen, welche Programmausschnitte auf Hardware und welche in Software abgebildet werden sowie die Generierung der Schnittstellen usw. sind Bestandteil der Systemsynthese.

Multi-FPGA-Systeme Für bestimmte Klassen von Rechenproblemen sind Computer geeignet, die aus einer Vielzahl von FPGAs aufgebaut sind. Ein Beispiel zum Brechen kryptografischer Verfahren ist der Parallelrechner COPACOBANA[63]. Er besteht aus 120 FPGAs und wurde für eine vollständige Schlüsselsuche des Data Encryption Standards[92] verwendet.

In zukünftigen Compilerversionen sollte eine Verteilung von Algorithmen auf Multi-FPGA-Systeme unterstützt werden. Voraussetzung dazu ist allerdings auch eine Anwendung, die entweder eine ausreichend große Anzahl an Prozessen hat oder sich z.B. durch das Abrollen von Schleifen stark genug parallelisieren lässt. Hardwareseitig muss die FPGA-Verschaltung auf Leiterplattenebene bekannt sein und entsprechende Schnittstellenmodule vorhanden sein, die mit den generierten Komponenten zusammengeschaltet werden.

Prozesse, die stark voneinander abhängig bzw. durch geteilte Ressourcen eng miteinander verzahnt sind, werden im selben FPGA implementiert. Sobald alle Ressourcen im FPGA verbraucht sind, werden die noch nicht implementierten Prozesse auf freie FPGAs abgebildet. Um bestimmen zu können, ob alle Ressourcen eines FPGAs verbraucht sind, muss der Compiler mit herstellerspezifischen Tools zusammenarbeiten. Dies kann nicht nur in Multi-FPGA-Systemen angewendet werden. Auch bei einem einzelnen FPGAs kann die Vorgabe sein, alle Ressourcen möglichst vollständig mit einem Programm auszulasten. Dazu kann das Programm so lange parallelisiert werden, bis alle Logikzellen verwendet sind.

Aufrufe von Instanzen von mehreren Funktionen heraus Bei der Implementierung der Sensoranwendung in Abschnitt 8.2.1 wird ein System beschrieben, das mit externen Bauteilen kommuniziert. Hier gibt es eine Komponente für die Sensor- und eine für die

Displaykommunikation, die beide auf verschiedene I2C-Busse zugreifen und somit separate Buscontroller als Submodul besitzen. Die Implementierung hätte einen erheblichen Mehraufwand mit sich geführt, wenn sich Sensor und Display denselben Bus geteilt hätten. In diesem Fall hätte man nur einen einzigen Buscontroller, der in den Komponenten nicht mehr als internes Submodul integrierbar wäre. Der Controller hätte sich außerhalb befinden müssen und wäre nicht mehr über Aufrufe, sondern nur über Streams zu steuern. Dies hätte neben der eigentlichen Streamkommunikation auch Synchronisationsmechanismen für die Transaktionen erfordert, wodurch auch Mutexe notwendig geworden wären.

Die Lösung dieses Problems in der Beschreibungssprache liegt in der Erzeugung von Objekten, die mehreren Prozessen als Parameter übergeben werden. Innerhalb dieser Prozesse werden diese Objekte wie ein lokal erzeugtes behandelt. Die notwendigen Kommunikationsbusse usw. werden automatisch generiert. In 9.1 ist ein Codebeispiel aufgelistet, wie die Definition und die Übergabe der Objekte aussehen könnte. Ein I2C-Busmodul mit den zugehörigen Streams wird instantiiert und den Prozessen übergeben. Neben den eigentlichen Kommunikationsroutinen werden auch Anweisungen zur Synchronisation verwendet.

Listing 9.1: Objekte, die weiteren Prozessen übergeben werden

```
1  void proc1(I2CBus &i2c)
2  {
3      i2c.lock()
4      i2c.send(...);
5      ...
6      i2c.unlock()
7  }
8  void main(ostream<uint1> sda,...) {
9      ...
10     I2CBus i2c(sda,...);
11     proc1(i2c);
12     proc2(i2c);
13     ...
14 }
```

Kapitel 10

Anhang

10.1 TransC-Grammatik

10.1.1 TransC-Grammatik, Flex-Spezifikation

```
1  DecNum    ([0-9])|([1-9][0-9]*)
2  HexNum    ("0x"|"0X")([0-9a-fA-F])+
3  BinNum    ("0b"|"0B")([01])+
4  Space     [ \t\n\r]
5
6  Pf   (u|U|l|L)*
7
8
9  %%
10
11 {Space} ;
12
13 "int"[1-9][0-9]*  return _INT;
14 "int"             return _INT;
15 "uint"[1-9][0-9]* return _UINT;
16 "uint"            return _UINT;
17 "bool"            return _BOOL;
18 "void"            return _VOID;
19 "struct"          return _STRUCT;
20
21 "unsigned"      return _UNSIGNED;
22 "signed"        return _SIGNED;
23 "char"          return _CHAR;
24 "short"         return _SHORT;
25 "long"          return _LONG;
26
27 "typedef"       return _TYPEDEF;
28 "const"         return _CONST;
29 "istream"       return _ISTREAM;
30 "ostream"       return _OSTREAM;
31 "sistream"      return _SISTREAM;
```

```
32 "sostream"      return _SOSTREAM;
33 "stream"        return _NSTREAM;
34 "nstream"       return _NSTREAM;
35 "sstream"       return _SNSTREAM;
36 "snstream"      return _SNSTREAM;
37 "block"         return _BLOCK;
38
39 "while"         return _WHILE;
40 "for"           return _FOR;
41 "do"            return _DO;
42 "sync"          return _SYNC;
43 "process"       return _PROCESS;
44 "if"            return _IF;
45 "else"          return _ELSE;
46 "switch"        return _SWITCH;
47 "case"          return _CASE;
48 "default"       return _DEFAULT;
49 "return"        return _RETURN;
50 "break"         return _BREAK;
51
52 "inline"        return _INLINE;
53 "static"        return _STATIC;
54
55
56 {DecNum}{Pf}?  return _NUMBER;
57 {HexNum}{Pf}?  return _NUMBER;
58 {BinNum}       return _NUMBER;
59 "'"[ -~]"'"    return _NUMBER;
60
61 ";"             return (*yytext);
62 "("             return (*yytext);
63 ")"             return (*yytext);
64 "["             return (*yytext);
65 "]"             return (*yytext);
66 "{"             return (*yytext);
67 "}"             return (*yytext);
68 ":"             return (*yytext);
69 "?"             return (*yytext);
70 ","             return (*yytext);
71 "."             return (*yytext);
72 "@"             return (*yytext);
73 "§"             return (*yytext);
74
75
76 ">"             return (*yytext);
77 "<"             return (*yytext);
78 "=="            return _EQEQ;
79 "!="            return _NEQ;
80 ">="            return _GEQ;
81 "<="            return _LEQ;
```

```
82
83  "+="            return _PLUSEQ;
84  "-="            return _MINUSEQ;
85  ">>="           return _SHREQ;
86  "<<="           return _SHLEQ;
87  "&="            return _ANDEQ;
88  "|="            return _OREQ;
89  "^="            return _XOREQ;
90  "*="            return _MULEQ;
91  "/="            return _DIVEQ;
92  "%="            return _MODEQ;
93
94
95  "++"            return _ADDADD;
96  "--"            return _SUBSUB;
97
98
99  "||"            return _OROR;
100 "&&"            return _ANDAND;
101 "|"             return _OR;
102 "^"             return _XOR;
103 "&"             return _AND;
104
105 ">>"            return _SHR;
106 "<<"            return _SHL;
107 "+"             yreturn _ADD;
108 "-"             return _SUB;
109 "*"             return _MUL;
110 "/"             return _DIV;
111 "%"             return _MOD;
112 "~"             return _INV;
113 "!"             return _BOOLNOT;
114
115 "="             return (*yytext);
116
117 "$"             return (*yytext);
118
119
120 [A-Za-z_][A-Za-z0-9_]*  {
121           /*
122            *  Pseudo-Code:
123            *  ------------
124            * if (yytext == var_type)
125            *       return(_CUSTOMTYPE);
126            *  else
127            *      return(_IDENTIFIER);
128            */
129             return _IDENTIFIER;
130       }
131
```

```
132
133 "//"[^\n]*      ;
134
135
136 "/*"           /* Kommentare filtern */
137 #pragma"␣"     /* Pragmaanweisungen verarbeiten */
138 #"␣"           /* bestimmte Präprozessoranweisungen verarbeiten... */
139
140
141 .                 return _ERROR;
142
143 %%
```

10.1.2 TransC Bison-Grammatik

```
1  %glr-parser
2
3  %start translation_unit
4
5  %token <unInt> _VOID _INT _UINT _BOOL _STRUCT
6  %token <unPVarType> _CUSTOMTYPE
7  %token <untInt> _NUMBER
8  %token <unPString> _IDENTIFIER _STRING
9  %token _EQEQ _NEQ _GEQ _LEQ _ADDEQUAL _SUBEQUAL
10 %token _ERROR
11 %token _WHILE _FOR _DO _BREAK _CONST _UNSIGNED _SIGNED _CHAR _SHORT _LONG
12 %token _SYNC _PROCESS
13 %token _IF _ELSE _SWITCH _CASE _DEFAULT _RETURN _ISTREAM _OSTREAM _NSTREAM
14                _SISTREAM _SOSTREAM _SNSTREAM _BLOCK
15 %token _TYPEDEF _INLINE _STATIC
16
17 %token _OROR _ANDAND _OR _XOR _AND
18 %token _SHR _SHL _ADD _SUB _MUL _DIV _MOD _INV _BOOLNOT
19 %token _PLUSEQ _MINUSEQ _SHREQ _SHLEQ _ANDEQ _XOREQ _OREQ _MULEQ
20                _DIVEQ REMAINDEREQUAL
21 %token _UNARYMINUS _UNARYPLUS
22
23
24
25 %right  '=' _PLUSEQ _MINUSEQ _SHREQ _SHLEQ _ANDEQ _OREQ _XOREQ _MULEQ
26                _DIVEQ _MODEQ
27
28 %left   ':' '?'
29 %left _OROR
30 %left _ANDAND
31 %left _OR
32 %left _XOR
33 %left _AND
34 %left _EQEQ _NEQ
```

```
35 %left _GEQ '>' '<' _LEQ
36 %left _SHR _SHL
37 %left _ADD _SUB
38 %left _MUL _DIV _MOD
39 %left   ')'
40 %left _ADDADD _SUBSUB _UNARYMINUS _UNARYPLUS _BOOLNOT _INV
41
42 %%
43
44
45 /*********************************************/
46
47 translation_unit:
48    global_decls
49    ;
50
51 global_decls
52    : func_decl1
53    | struct_decl
54    | typedef
55    | global_var_decl
56    | func_decl1      global_decls
57    | struct_decl     global_decls
58    | typedef         global_decls
59    | global_var_decl global_decls
60    ;
61
62 struct_decl
63    : _STRUCT identifier '{' declarations '}' ';'
64    ;
65
66 typedef:
67    _TYPEDEF var_type identifier typedef_array ';'
68    |
69    _TYPEDEF var_type _CUSTOMTYPE typedef_array ';'
70    ;
71
72 typedef_array
73    :
74    | array_dims
75    ;
76
77 global_var_decl
78    : declaration ';'
79    ;
80
81 func_decl1
82    : func_decl
83    | _STATIC func_decl1
84    | _INLINE func_decl1
```

```
85      | _PROCESS func_decl
86      ;
87
88  func_decl
89      : func_head '{' func_body '}'
90      | func_head ';'
91      ;
92
93  func_head
94      : func_result func_name '(' param_list ')' func_properties
95      ;
96
97  func_result
98      : result_type
99      | '(' result_types ')'
100     ;
101
102 result_type
103     : var_type
104     ;
105
106 result_types
107     : result_type
108     | result_types ',' result_type
109     ;
110
111 func_name
112     : identifier
113     ;
114
115 param_list :
116             | param_dekl
117             | param_dekl ',' param_list
118             ;
119
120 param_dekl
121     : var_type identifier array_def
122     | var_type _AND identifier
123     ;
124
125 stream_type
126     : _ISTREAM
127     | _OSTREAM
128     | _NSTREAM
129     | _SISTREAM
130     | _SOSTREAM
131     | _SNSTREAM
132     ;
133
134
```

```
135 func_properties
136    :
137    ;
138
139 /*********************************************/
140
141 func_body
142    : func_declarations statements
143    ;
144
145 declarations
146    : }
147    | declaration ';' declarations
148    ;
149
150 func_declarations
151    :
152    | func_declarations declaration ';'
153    ;
154
155 declaration :
156    var_type var_list
157    ;
158
159 var_name : identifier ;
160
161 array_def
162    :
163    | '=' const_expr
164    | array_dims array_init
165    ;
166
167 array_init
168    :
169    | '=' '{' array_numbers1 '}'
170    ;
171
172 array_numbers1
173    : array_numbers
174    | array_numbers ','
175    ;
176
177 array_numbers
178    : const_expr
179    | array_numbers ',' const_expr
180    ;
181
182 array_dims
183    : '[' ']'
184    | array_dims '[' ']'
```

```
185     | '[' const_expr ']'
186     | array_dims '[' const_expr ']'
187     ;
188
189 const_expr
190     : expression
191     ;
192
193 var_list
194     : var_list1 ','
195     | var_list1
196     ;
197
198 var_list1
199     : var_decl
200     | var_list1 ',' var_decl
201     ;
202
203 expr_list
204     : expr_list1
205     | expr_list1 ','
206     ;
207
208 expr_list1
209     : expression
210     | expr_list1 ',' expression
211     ;
212
213 var_decl
214     : var_name array_def
215     ;
216
217 statements :
218     | stmt statements
219     ;
220
221 stmt
222     : ';'
223     | expression ';'
224     | '{' statements '}'
225     | _WHILE '(' expression ')' stmt
226     | _BREAK
227     | for_loop
228     | do_while_loop
229     | functioncall_stmt ';'
230     | if_stmt
231     | switch_stmt
232     | return_stmt ';'
233     | _SYNC ';'
234     | _SYNC functioncall_expr ';'
```

```
235     | _SYNC identifiers ';'
236     ;
237
238 if_stmt
239     : if_head stmt
240     | if_head stmt _ELSE stmt
241     ;
242
243 if_head:
244     _IF '(' expression ')'
245     ;
246
247 switch_stmt
248     : _SWITCH  '(' expression ')' '{' case_list case_default '}'
249     ;
250
251 case_list
252     : case_block case_list
253     | case_block
254     ;
255
256 case_head
257     : _CASE const_expr ':'
258     ;
259
260 case_block
261     : case_head statements
262     ;
263
264 case_default_head
265     : _DEFAULT ':'
266     ;
267
268 case_default
269     :
270     | case_default_head statements
271     ;
272
273 return_stmt:
274     _RETURN
275     | _RETURN expression
276     | _RETURN '('ret_list')'
277     | _RETURN ret_list
278     ;
279
280 ret_list
281     : expression ',' expression
282     | ret_list ',' expression
283     ;
284
```

```
285 for_loop
286    : _FOR '(' expression ';' expression ';' expression ')' stmt
287    ;
288
289
290 do_while_loop
291    : _DO stmt _WHILE '(' expression ')'
292    ;
293
294
295 functioncall_stmt:
296    '(' fctcall_ret_list ')' '=' functioncall_expr
297    ;
298
299 fctcall_ret_list
300    : fctcall_ret_list1 ',' fctcall_ret_list1
301    | /* empty */
302    ;
303
304 fctcall_ret_list1:
305    fctcall_ret_list1 ',' fctcall_ret_list1
306    | variable
307    | /* empty */
308    ;
309
310 fctcall_param_list:
311    fctcall_param_list1
312    | /* empty */
313    ;
314 fctcall_param_list1
315    : expression
316    | expression ',' fctcall_param_list1
317    ;
318
319
320 /*******************************************/
321
322
323 var_type
324       : _INT  '<' const_expr '>'
325       | _UINT '<' const_expr '>'
326       | _INT
327       | _UINT
328
329       | _BOOL
330       | _CUSTOMTYPE
331
332
333       | _SHORT
334       | _SHORT _INT
```

```
335       | _CHAR
336       | _LONG
337       | _LONG _INT
338       | _LONG _LONG
339       | _LONG _LONG _INT
340
341       | _SIGNED _SHORT
342       | _SIGNED _SHORT _INT
343       | _SIGNED
344       | _SIGNED _CHAR
345       | _SIGNED _INT
346       | _SIGNED _LONG
347       | _SIGNED _LONG _INT
348       | _SIGNED _LONG _LONG
349       | _SIGNED _LONG _LONG _INT
350
351       | _UNSIGNED _SHORT
352       | _UNSIGNED _SHORT _INT
353       | _UNSIGNED
354       | _UNSIGNED _CHAR
355       | _UNSIGNED _INT
356       | _UNSIGNED _LONG
357       | _UNSIGNED _LONG _INT
358       | _UNSIGNED _LONG _LONG
359       | _UNSIGNED _LONG _LONG _INT
360
361       | _VOID
362       | _STRUCT _CUSTOMTYPE
363       | _CONST var_type
364       | stream_type '<' var_type '>'
365       ;
366
367
368
369
370 expression
371    : term
372    | _INV term
373    | _BOOLNOT term
374    | expression _AND expression
375    | expression _OR expression
376    | expression _ANDAND expression
377    | expression _OROR expression
378    | expression _XOR expression
379    | expression _ADD expression
380    | expression _SUB expression
381
382    | expression _EQEQ expression
383    | expression _NEQ expression
384    | expression _GEQ expression
```

```
385     | expression '>'  expression
386     | expression _LEQ expression
387     | expression '<'  expression
388 
389     | expression _SHR expression
390     | expression _SHL expression
391     | expression _MUL expression
392     | expression _DIV expression
393     | expression _MOD expression
394     | expression '$' expression
395 
396     | expression '=' expression
397 
398     | expression _PLUSEQ expression
399     | expression _MINUSEQ expression
400     | expression _SHREQ expression
401     | expression _SHLEQ expression
402     | expression _ANDEQ expression
403     | expression _OREQ expression
404     | expression _XOREQ expression
405     | expression _MULEQ expression
406     | expression _DIVEQ expression
407     | expression _MODEQ expression
408 
409     | '(' var_type ')' expression
410     | _SUB expression %prec _UNARYMINUS
411     | _ADD expression %prec _UNARYPLUS
412     | functioncall_expr
413     | variable access
414     | expression '?' expression ':' expression
415     |  expression _ADDADD
416     |  expression _SUBSUB
417     | _ADDADD expression
418     | _SUBSUB expression
419     |'<' expr_list '>'
420     ;
421 
422 
423 access
424     : '.' identifier
425     | '[' expression ']'
426     | access '.' identifier
427     | access '[' expression ']'
428     ;
429 
430 functioncall_expr:
431     identifier '(' fctcall_param_list ')' opt_identifier
432     ;
433 
434 
```

```
435 term
436    : variable
437    | constant
438    | '(' expression ')'
439    | variable '.' _BLOCK '(' ')'
440    ;
441
442 variable:
443    identifier
444    ;
445
446 constant : _NUMBER
447
448 opt_identifier
449    : '@' identifier
450    |
451    ;
452
453 identifier : _IDENTIFIER;
454
455
456 identifiers
457    : identifier
458    | identifiers ',' identifier
459    ;
```

10.2 Beispiel-VHDL-Code

Die folgenden Abschnitte zeigen das Zwischenformat in grafischer Form sowie die vom TransC-Compiler erzeugte VHDL-Ausgabe für den euklidischen ggT-Algorithmus (größter gemeinsamer Teiler). Anschließend werden alle verfügbaren Optimierungen durchgeführt und das Zwischenformat sowie die VHDL-Ausgabe nochmals aufgezeigt.

ggT-Algorithmus in TransC

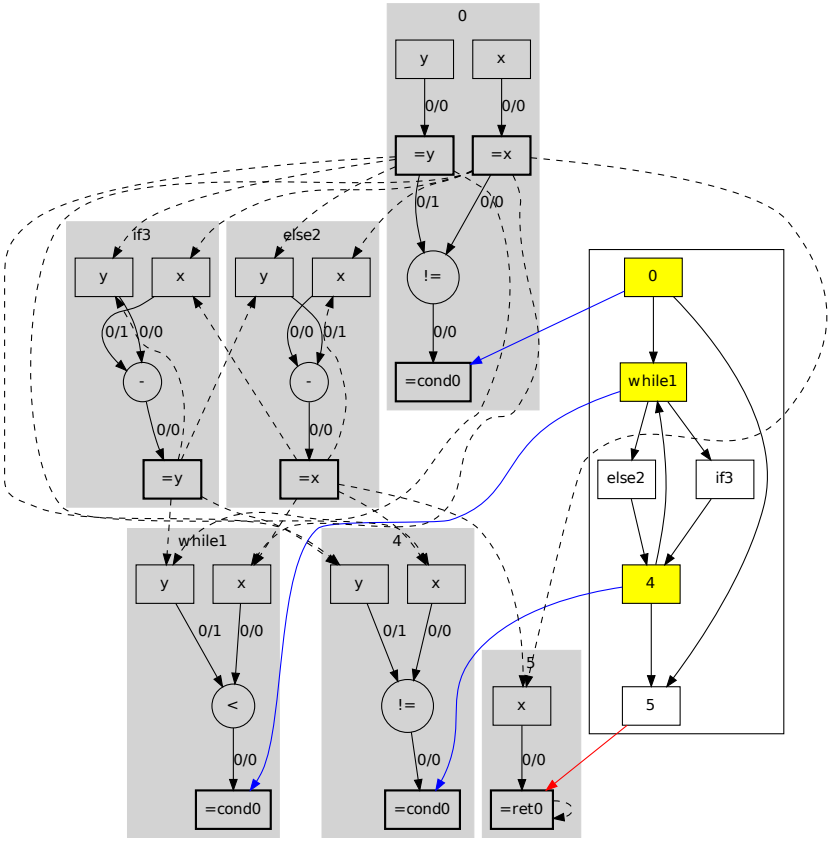
```
1  unsigned int ggt (unsigned int x, unsigned int y)
2  {
3     while (x != y) {
4        if (x<y)
5           y -= x;
6        else
7           x -= y;
8     }
9     return x;
10 }
```

Resultate ohne Optimierungen

Aufruf:

```
1 transc ggt.c
```

CDFG ohne Optimierungen



VHDL-Ausgabe ohne Optimierungen

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use work.transc.all;
5
6  entity ggt is
7  port (
8     arg_x : in  std_logic_vector(31 downto 0);
9     arg_y : in  std_logic_vector(31 downto 0);
10    ret0 : out std_logic_vector(31 downto 0);
11    run  : in  std_logic;
12    done : out std_logic;
13    clk  : in  std_logic;
```

```
14    rst  : in  std_logic
15 );
16 end ggt;
17 
18 architecture RTL of ggt is
19 
20    --workaround: additional signals for ISE webpack to detect the FSM--
21    signal w6a : std_logic;
22    signal w0a : std_logic;
23    signal w3a : std_logic;
24    ----------------------------------------------------------------------
25 
26    signal w0 : std_logic;
27    signal w1 : std_logic_vector(31 downto 0);
28    signal w2 : std_logic_vector(31 downto 0);
29    signal w3 : std_logic;
30    signal w4 : std_logic_vector(31 downto 0);
31    signal w5 : std_logic_vector(31 downto 0);
32    signal w6 : std_logic;
33    signal x : std_logic_vector(31 downto 0);
34    signal y : std_logic_vector(31 downto 0);
35 
36    type STATE_TYPE is (
37       SI,
38       S0_0,
39       S1_0,
40       S2_0,
41       S3_0,
42       S4_0,
43       S5_0  ); -- 6 states
44    signal current_state, next_state: STATE_TYPE;
45 
46 
47 
48 begin
49    active_proc: process
50    begin
51       wait until clk'event␣and␣clk␣=␣'1';
52 ␣␣␣␣␣␣if␣rst='1'␣then
53 ␣␣␣␣␣␣␣␣␣␣current_state␣<=␣SI;
54 ␣␣␣␣␣␣else
55 ␣␣␣␣␣␣␣␣␣␣current_state␣<=␣next_state;
56 ␣␣␣␣␣␣end␣if;
57 ␣␣␣end␣process;
58 ␣␣␣done␣<=␣'1'␣when␣current_state=SI␣else␣'0';
59 
60 
61 
62 
63 ␣␣␣states:␣process(current_state,␣run,␣w6a,␣w0a,␣w3a)
```

```
64  ␣␣␣begin
65  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣current_state;
66  ␣␣␣␣␣␣␣case␣current_state␣is
67  ␣␣␣␣␣␣␣when␣SI␣=>
68  ␣␣␣␣␣␣␣␣␣␣␣if␣run='1'␣then
69  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S0_0;
70  ␣␣␣␣␣␣␣␣␣␣␣else
71  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣SI;
72  ␣␣␣␣␣␣␣␣␣␣␣end␣if;
73  ␣␣␣␣␣␣␣when␣S0_0␣=>
74  ␣␣␣␣␣␣␣␣␣␣␣if␣SL2BOOL(w6a)␣then
75  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S1_0;
76  ␣␣␣␣␣␣␣␣␣␣␣else
77  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S5_0;
78  ␣␣␣␣␣␣␣␣␣␣␣end␣if;
79  ␣␣␣␣␣␣␣when␣S1_0␣=>
80  ␣␣␣␣␣␣␣␣␣␣␣if␣SL2BOOL(w0a)␣then
81  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S3_0;
82  ␣␣␣␣␣␣␣␣␣␣␣else
83  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S2_0;
84  ␣␣␣␣␣␣␣␣␣␣␣end␣if;
85  ␣␣␣␣␣␣␣when␣S2_0␣=>
86  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S4_0;
87  ␣␣␣␣␣␣␣when␣S3_0␣=>
88  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S4_0;
89  ␣␣␣␣␣␣␣when␣S4_0␣=>
90  ␣␣␣␣␣␣␣␣␣␣␣if␣SL2BOOL(w3a)␣then
91  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S1_0;
92  ␣␣␣␣␣␣␣␣␣␣␣else
93  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S5_0;
94  ␣␣␣␣␣␣␣␣␣␣␣end␣if;
95  ␣␣␣␣␣␣␣when␣S5_0␣=>
96  ␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣SI;
97  ␣␣␣␣␣␣␣when␣others␣=>
98  ␣␣␣␣␣␣␣end␣case;
99  ␣␣␣end␣process;
100
101
102 ␣␣␣reg_proc:␣process
103 ␣␣␣begin
104 ␣␣␣␣␣␣␣wait␣until␣clk'event and clk = '1';
105       case current_state is
106       when SI =>
107          x <= arg_x;
108          y <= arg_y;
109       when S0_0 =>
110          x <= x;     -- =x
111          y <= y;     -- =y
112       when S1_0 =>
113       when S2_0 =>
```

```
114            x <= w2;     -- =x
115         when S3_0 =>
116            y <= w1;     -- =y
117         when S4_0 =>
118         when S5_0 =>
119            ret0 <= x;     -- =ret0
120         when others =>
121         end case;
122      end process;
123
124
125      bus_proc: process(current_state, run, w0, w1, w2, w3, w4, w5, w6, x, y)
126      begin
127         w0 <= '-';
128         w1 <= (others=>'-');
129         w2 <= (others=>'-');
130         w3 <= '-';
131         w4 <= (others=>'-');
132         w5 <= (others=>'-');
133         w6 <= '-';
134         case current_state is
135         when SI =>
136         when S0_0 =>
137            w4 <= x;     -- =x
138            w5 <= y;     -- =y
139            w6 <= BOOL2SL(signed(w4)/=signed(w5));
140         when S1_0 =>
141            w0 <= BOOL2SL(signed(x)<signed(y));
142         when S2_0 =>
143            w2 <= std_logic_vector( signed(x) - signed(y) );
144         when S3_0 =>
145            w1 <= std_logic_vector( signed(y) - signed(x) );
146         when S4_0 =>
147            w3 <= BOOL2SL(signed(x)/=signed(y));
148         when S5_0 =>
149         when others =>
150         end case;
151      end process;
152
153      --workaround: additional signals for ISE webpack to detect the FSM--
154      w6a <= w6;
155      w0a <= w0;
156      w3a <= w3;
157      ---------------------------------------------------------------------
158
159
160   end RTL;
```

Ergebnisse Platzierung und Verdrahtung

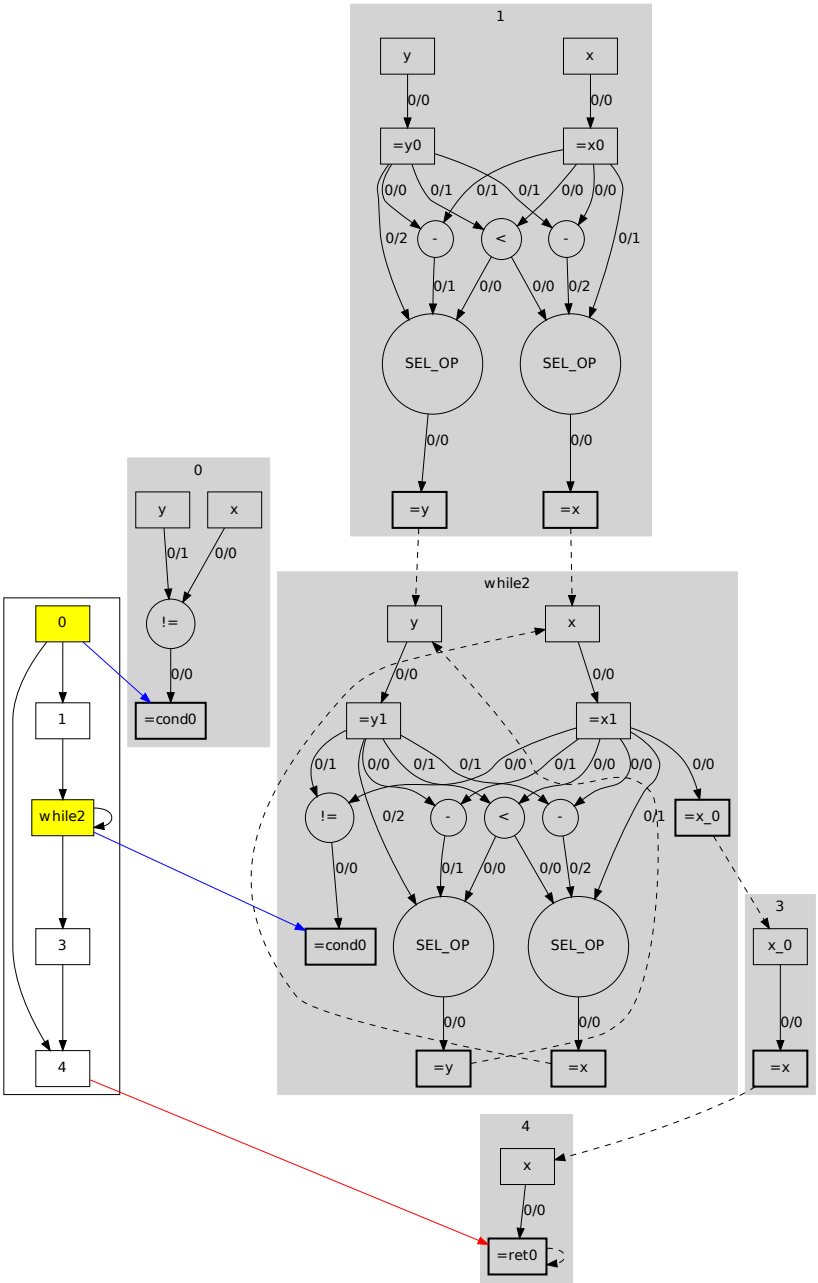
- FPGA: Xilinx XC6SLX75T
- Low-Level-Synthesetool: Xilinx ISE WebPACK, Version 13.1
- Ressourcenverbrauch: 179 Slice-LUTs
- Max. Taktfrequenz: 205 MHz
- Laufzeit: 59 Takte (*Berechnung ggt(600,36)*)

Resultate mit Optimierungen

Aufruf:

```
1 transc ggt.c -O3 --eval -c -p20
```

CDFG mit Optimierungen



VHDL-Ausgabe mit Optimierungen

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use work.transc.all;
5
6 entity ggt is
7 port (
```

```
8     arg_x : in  std_logic_vector(31 downto 0);
9     arg_y : in  std_logic_vector(31 downto 0);
10    ret0 : out std_logic_vector(31 downto 0);
11    run  : in  std_logic;
12    done : out std_logic;
13    clk  : in  std_logic;
14    rst  : in  std_logic
15 );
16 end ggt;
17
18 architecture RTL of ggt is
19
20    --workaround: additional signals for ISE webpack to detect the FSM--
21    signal w0a : std_logic;
22    signal w15a : std_logic;
23    ---------------------------------------------------------------------
24
25    signal w0 : std_logic;
26    signal w1 : std_logic_vector(31 downto 0);
27    signal w2 : std_logic_vector(31 downto 0);
28    signal w3 : std_logic_vector(32 downto 0);
29    signal w4 : std_logic;
30    signal w5 : std_logic_vector(31 downto 0);
31    signal w6 : std_logic_vector(32 downto 0);
32    signal w7 : std_logic_vector(31 downto 0);
33    signal w8 : std_logic_vector(31 downto 0);
34    signal w9 : std_logic_vector(31 downto 0);
35    signal w10 : std_logic_vector(32 downto 0);
36    signal w11 : std_logic;
37    signal w12 : std_logic_vector(31 downto 0);
38    signal w13 : std_logic_vector(32 downto 0);
39    signal w14 : std_logic_vector(31 downto 0);
40    signal w15 : std_logic;
41    signal x_0 : std_logic_vector(31 downto 0);
42    signal y0 : std_logic_vector(31 downto 0);
43    signal x0 : std_logic_vector(31 downto 0);
44    signal y1 : std_logic_vector(31 downto 0);
45    signal x1 : std_logic_vector(31 downto 0);
46    signal x : std_logic_vector(31 downto 0);
47    signal y : std_logic_vector(31 downto 0);
48
49    type STATE_TYPE is (
50       SI,
51       S0_0,
52       S1_0,
53       S2_0,
54       S3_0,
55       S4_0  ); -- 5 states
56    signal current_state, next_state: STATE_TYPE;
57
```

```
58
59
60  begin
61     active_proc: process
62     begin
63        wait until clk'event␣and␣clk␣=␣'1';
64  ␣␣␣␣␣␣if␣rst='1'␣then
65  ␣␣␣␣␣␣␣␣␣␣current_state␣<=␣SI;
66  ␣␣␣␣␣␣else
67  ␣␣␣␣␣␣␣␣␣␣current_state␣<=␣next_state;
68  ␣␣␣␣␣␣end␣if;
69  ␣␣␣end␣process;
70  ␣␣␣done␣<=␣'1'␣when␣current_state=SI␣else␣'0';
71
72
73
74
75  ␣␣␣states:␣process(current_state,␣run,␣w0a,␣w15a)
76  ␣␣␣begin
77  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣current_state;
78  ␣␣␣␣␣␣case␣current_state␣is
79  ␣␣␣␣␣␣when␣SI␣=>
80  ␣␣␣␣␣␣␣␣␣␣if␣run='1'␣then
81  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S0_0;
82  ␣␣␣␣␣␣␣␣␣␣else
83  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣SI;
84  ␣␣␣␣␣␣␣␣␣␣end␣if;
85  ␣␣␣␣␣␣when␣S0_0␣=>
86  ␣␣␣␣␣␣␣␣␣␣if␣SL2BOOL(w0a)␣then
87  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S1_0;
88  ␣␣␣␣␣␣␣␣␣␣else
89  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S4_0;
90  ␣␣␣␣␣␣␣␣␣␣end␣if;
91  ␣␣␣␣␣␣when␣S1_0␣=>
92  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S2_0;
93  ␣␣␣␣␣␣when␣S2_0␣=>
94  ␣␣␣␣␣␣␣␣␣␣if␣SL2BOOL(w15a)␣then
95  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S2_0;
96  ␣␣␣␣␣␣␣␣␣␣else
97  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S3_0;
98  ␣␣␣␣␣␣␣␣␣␣end␣if;
99  ␣␣␣␣␣␣when␣S3_0␣=>
100 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣S4_0;
101 ␣␣␣␣␣␣when␣S4_0␣=>
102 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣next_state␣<=␣SI;
103 ␣␣␣␣␣␣when␣others␣=>
104 ␣␣␣␣␣␣end␣case;
105 ␣␣␣end␣process;
106
107
```

```
␣␣␣reg_proc:␣process
␣␣␣begin
␣␣␣␣␣␣␣wait␣until␣clk'event and clk = '1';
      case current_state is
      when SI =>
         x <= arg_x;
         y <= arg_y;
      when S0_0 =>
      when S1_0 =>
         y <= w5;     -- =y
         x <= w7;     -- =x
      when S2_0 =>
         y <= w12;     -- =y
         x <= w14;     -- =x
         x_0 <= w9;     -- =x_0
      when S3_0 =>
         x <= x_0;     -- =x
      when S4_0 =>
         ret0 <= x;     -- =ret0
      when others =>
      end case;
   end process;


   bus_proc: process(current_state, run, w0, w1, w2, w3, w4, w5, w6,
               w7, w8, w9, w10, w11, w12, w13, w14, w15, x_0,
               y0, x0, y1, x1, x, y)
   begin
      w0 <= '-';
      w1 <= (others=>'-');
      w2 <= (others=>'-');
      w3 <= (others=>'-');
      w4 <= '-';
      w5 <= (others=>'-');
      w6 <= (others=>'-');
      w7 <= (others=>'-');
      w8 <= (others=>'-');
      w9 <= (others=>'-');
      w10 <= (others=>'-');
      w11 <= '-';
      w12 <= (others=>'-');
      w13 <= (others=>'-');
      w14 <= (others=>'-');
      w15 <= '-';
      case current_state is
      when SI =>
      when S0_0 =>
         w0 <= BOOL2SL(signed(x)/=signed(y));
      when S1_0 =>
         w1 <= y;     -- =y0
```

```
158          w2 <= x;     -- =x0
159          w3 <= std_logic_vector( resize(signed(w1), 33) -
160                resize(signed(w2), 33) );
161          w4 <= BOOL2SL(signed(w2)<signed(w1));
162          if SL2BOOL(w4) then
163             w5 <= w3(31 downto 0);
164          else
165             w5 <= w1;
166          end if;
167          w6 <= std_logic_vector( resize(signed(w2), 33) -
168                resize(signed(w1), 33) );
169          if SL2BOOL(w4) then
170             w7 <= w2;
171          else
172             w7 <= w6(31 downto 0);
173          end if;
174       when S2_0 =>
175          w8 <= y;     -- =y1
176          w9 <= x;     -- =x1
177          w10 <= std_logic_vector( resize(signed(w8), 33) -
178                resize(signed(w9), 33) );
179          w11 <= BOOL2SL(signed(w9)<signed(w8));
180          if SL2BOOL(w11) then
181             w12 <= w10(31 downto 0);
182          else
183             w12 <= w8;
184          end if;
185          w13 <= std_logic_vector( resize(signed(w9), 33) -
186                resize(signed(w8), 33) );
187          if SL2BOOL(w11) then
188             w14 <= w9;
189          else
190             w14 <= w13(31 downto 0);
191          end if;
192          w15 <= BOOL2SL(signed(w9)/=signed(w8));
193       when S3_0 =>
194       when S4_0 =>
195       when others =>
196       end case;
197    end process;
198
199    --workaround: additional signals for ISE webpack to detect the FSM--
200    w0a <= w0;
201    w15a <= w15;
202    ----------------------------------------------------------------------
203
204
205 end RTL;
```

Ergebnisse nach Platzierung und Verdrahtung

- FPGA: Xilinx XC6SLX75T
- Low-Level-Synthesetool: Xilinx ISE WebPACK, Version 13.1
- Ressourcenverbrauch: 525 Slice-LUTs
- Max. Taktfrequenz: 170 MHz
- Laufzeit: 25 Takte (*Berechnung ggt(600,36)*)

10.3 Compilerbenutzung

Aufruf auf der Kommandozeile:

```
1 transc <Dateiname> [Optionen]
```

Für möglichst effiziente Schaltungen wird folgender Aufruf empfohlen:

```
1 transc <Dateiname> -p30 --eval -O3 -c
```

10.3.1 Kommandozeilenparameter:

-h –help

Hilfe ausgeben

-c –combine-if

IF-Blöcke in Select-Anweisungen transformieren, wenn möglich

-g –graph

CDFG im GraphViz-Format exportieren

-On

Optimierungsgrad (0...3). Ohne Optimierungen ist der erzeugte VHDL-Code leichter zu lesen, da Konstrukte, Variablen usw. aus dem Originalprogramm erhalten bleiben. Für eine möglichst hohe Schaltungseffizienz wird *-O3* empfohlen.

-v –version

Compilerversion anzeigen

-i –ir

Zwischenformat ausgeben (XML-Format)

-p[<n>] –pipe

Schleifen parallelisieren. Wenn n spezifiziert ist, muss die Schleife mindestens um n % beschleunigt werden, sonst wird die Optimierung nicht angewendet (0 ist Standardwert)

–bram <n>

BRAM Grenze (Standardwert 16). Arrays oberhalb der hier angegebenen Größe werden als Block-RAM synthetisiert.

-u –unroll

Schleifen vollständig abrollen, wenn möglich

-P[<n>] –eval

Partielle Evaluierung und alle resultierenden Optimierungen anwenden. Abbrechen, wenn mehr als n Blöcke evaluiert wurden (Standardwert 10000).

-t –top <name>

Top-Modul spezifizieren. Wenn ein Top-Modul angegeben ist, werden alle nicht benötigten Module gelöscht. Bei der Verwendung von globalen Variablen muss ein Top-Modul angegeben werden.

–lib

Dateien mit den benötigten Bibliotheken erzeugen. Die Bibliotheken sind immer identisch und unabhängig vom eingelesenen Quellcode. Zuvor generierte Dateien können wiederverwendet werden, insofern sie mit der selben Version des TransC-Compilers generiert wurden.

10.4 Sprachdokumentation

Die TransC-Sprachdokumentation ist als technischer Report auf der Seite `http://cgi.tu-harburg.de/~ti6hm` verfügbar.

Literaturverzeichnis

[1] The Go Programming Language, `http://golang.org/`, October 2011.

[2] T. Adam, K. Chandy, and J. Dickson. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM*, 17:685–690, 1974.

[3] R. Adams and G. Steven. A parallel pipelined processor with conditional instruction execution. *SIGARCH Comput. Archit. News*, 19:135–142, March 1991.

[4] Al.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[6] A.V. Aho and J.D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[7] A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. In *ESOP*, pages 221–235, 1988.

[8] A. Aiken, A. Nicolau, and S. Novack. Resource-Constrained Software Pipelining. *IEEE Trans. Parallel Distrib. Syst.*, 6(12):1248–1270, 1995.

[9] P. Arato, V. Tamas, and I. Jankovits. *High Level Synthesis of Pipelined Datapaths.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[10] J. Armstrong. *Programming Erlang - Software for a Concurrent World.* The Pragmatic Bookshelf, Raleigh, North Carolina; Dallas, Texas, 2007.

[11] Design Automation and Standards Committee. IEEE Standard VHDL Synthesis Packages. *World Wide Web Internet And Web Information Systems*, 1997.

[12] M.F. Banahan, D. Brady, and M. Doran. *The C book - featuring the ANSI C standard (2. ed.).* Addison-Wesley, 1991.

[13] C.S. Bassoy, H. Manteuffel, and F. Mayer-Lindenberg. Sharf: An FPGA-based customizable processor architecture. In *19th International Conference on Field Programmable Logic and Applications*, pages 516–520. IEEE, 2009.

[14] B. Baumgarten. *Petri-Netze: Grundlagen und Anwendungen.* BI Wissenschaftsverlag, 1990.

[15] Y. Ben-Asher and N. Rotem. Binary Synthesis with multiple memory banks targeting array references. In *19th International Conference on Field Programmable Logic and Applications*, pages 600–603. IEEE, 2009.

[16] M. Budiu and S.C. Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *EuroPar 2000 European Conference on Parallel Computing*, pages 969–979. Springer Verlag, 2000.

[17] D. Butenhof. *Programming with Posix Threads.* Addison-Wesley, 1997.

[18] T.J. Callahan. *Automatic compilation of C for hybrid reconfigurable architectures.* PhD thesis, 2002. AAI3082127.

[19] A.M.K. Cheng. *Real-time systems: scheduling, analysis, and verification.* Wiley-Interscience. Wiley-Interscience, 2002.

[20] J. Cocke. Global common subexpression elimination. In *Proc. of a symposium on Compiler optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[21] Altera Corporation. Altera Website, `http://www.altera.com`, October 2011.

[22] Nokia Corporation. Qt Website, `http://doc.qt.nokia.com/`, October 2011.

[23] O.-J. Dahl, E.W. Dijkstra, and T. Hoare. *Structured Programming.* Academic Press, London, 1972.

[24] G. De Micheli. Hardware Synthesis from C/C++ Models. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, pages 382–383, 1999.

[25] S. Debois. Imperative program optimization by partial evaluation. In *Proc. of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '04, pages 113–122, New York, NY, USA, 2004. ACM.

[26] A. Dehon. The Density Advantage of Configurable Computing. *IEEE Computer*, 33:41–49, 2000.

[27] E.W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[28] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). Technical Report 3174, 9 2001.

[29] M. Ehrgott. *Multicriteria Optimization (2. ed.).* Springer, 2005.

[30] Smart Electronix. Music-DSP Source Code Archive, `http://www.musicdsp.org/`, October 2011.

[31] The International Technology Roadmap for Semiconductors. 2005 edition. Technical report, ITRS, 2005.

[32] The International Technology Roadmap for Semiconductors. 2009 Edition. Technical report, ITRS, 2009.

[33] Y. Futamura. Partial Computation of Programs. In *RIMS Symposium on Software Science and Engineering*, pages 1–35, 1982.

[34] D.D. Gajski. *High-level synthesis: introduction to chip and system design.* Kluwer Academic, 1992.

[35] N. Gasson and N. Audsley. Synthesis of the SR Programming Language for Complex FPGAs. In *19th International Conference on Field Programmable Logic and Applications*, pages 617–621. IEEE, 2009.

[36] R. Genevriere and A. Hoffmann. PMOSS - A Modular Synthesis and HW/SW-Codesign System, 1994.

[37] J. Gerlach. *Transformationale Entwurfsraum-Exploration für den Entwurf eingebetteter Systeme.* Logos Verlag Berlin, 2000.

[38] A.J. Glenstrup and N.D. Jones. Termination Analysis and Specialization-Point Insertion in Off-line Partial Evaluation. *ACM Trans. Program. Lang. Syst*, 27:2005, 2004.

[39] Mentor Graphics. Mentor Graphics, Catapult C Website, `http://www.mentor.com/esl/catapult/`, October 2011.

[40] Mentor Graphics. Mentor Graphics, Handle C Website, `http://www.mentor.com/products/fpga/handel-c/`, October 2011.

[41] T. Grotker. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[42] M. Gschwind and V. Salapura. A VHDL Design Methodology for FPGAs. In *Proc. of the 5th International Workshop on Field-Programmable Logic and Applications*, FPL '95, pages 208–217, London, UK, 1995. Springer-Verlag.

[43] Z. Guo, B. Buyukkurt, and W. Najjar. Optimized Generation of Data-path from C Codes for FPGAs. In *Proc. ACM/IEEE Design Automation and Test Europe*, DATE 05, 2005.

[44] S. Gupta, R.K. Gupta, and N.D. Dutt. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.* 2004.

[45] H. Gädke and A. Koch. Comrade - A Compiler for Adaptive Computing Systems Using a Novel Fast Speculation Technique. In Koen Bertels, Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors, *17th International Conference on Field Programmable Logic and Applications*, pages 503–504. IEEE, 2007.

[46] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *The IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195. IEEE, 2008.

[47] D. Herrmann and R. Ernst. Improved interconnect sharing by identity operation insertion. In *Proc. of Computer-Aided Design, 1999*, Computer-Aided Design, 1999, pages 489–492, 1999.

[48] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[49] J. Horowitz. *Critical path scheduling: management control through CPM and PERT.* Ronald Press Co., 1967.

[50] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 10(4):464–475, 1991.

[51] Free Software Foundation Inc. Bison - GNU parser generator, `http://www.gnu.org/s/bison/`, May 2011.

[52] Free Software Foundation Inc. GCC, The GNU Compiler Collection, `http://gcc.gnu.org/`, May 2011.

[53] Impulse Accelerated Technologies Inc. Impulse C, `http://www.impulseaccelerated.com/`, October 2011.

[54] Xilinx Inc. AutoESL Website C, `http://www.autoesl.com/`, October 2011.

[55] Xilinx Inc. Xilinx Website, `http://www.xilinx.com`, October 2011.

[56] K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *STTT*, 9(3-4):213–254, 2007.

[57] G. Jones. *Programming in Occam.* Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[58] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[59] T. Kailath. *Modern signal processing.* Proc. of the Arab School on Science and Technology. Hemisphere, 1985.

[60] R.B. Kieburtz and A. Silberschatz. Comments on "Communicating Sequential Processes". *ACM Trans. Program. Lang. Syst.*, 1:218–225, October 1979.

[61] S. Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization.* Wiley-IEEE Press, 2007.

[62] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[63] S. Kumar, C. Paar, and J. Pelzl. How to Break DES for BC 8,980. In *2nd International Workshop on Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS'06)*, pages 3–4.

[64] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007. Modern effort to quantify the relative area, power, and delay of FPGAs compared to ASICs.

[65] M.S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI*, pages 318–328, 1988.

[66] B. Landwehr and Verein Deutscher Ingenieure. *ILP-basierte Mikroarchitektur-Synthese mit komplexen Bausteinbibliotheken: Wege zu kleineren und schnelleren Schaltungen unter Einsatz algebraischer Optimierungen.* Berichte der GI/GMM/ITG-Kooperationsgemeinschaft Rechnergestützter Schaltungs- und Systementwurf. VDI-Verlag, 1998.

[67] C. Lattner. The LLVM Compiler System. *Language*, 2007.

[68] G. Lindenmaier. libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Technical Report 2002-5, September 2002.

[69] D.J. Mallon and P.B. Denyer. A new approach to pipeline optimisation. In *Proc. of the conference on European design automation*, EURO-DAC '90, pages 83–88, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[70] Z. Manna and R.J. Waldinger. Synthesis: Dreams - Programs. *IEEE Trans. Software Eng.*, 5(4):294–328, 1979.

[71] A.J. Martin. A new generalization of Dekker's algorithm for mutual exclusion. *Inf. Process. Lett.*, 23:295–297, December 1986.

[72] F. Mayer-Lindenberg. *Dedicated Digital Processors: Methods in Hardware/Software System Design.* John Wiley & Sons, 2004.

[73] F. Mayer-Lindenberg. High-level FPGA Programming through Mapping Process Networks to FPGA Resources. In *2009 International Conference on ReConFigurable Computing and FPGAs*, ReConFig '09, pages 302–307, 2009.

[74] M.C. McFarland. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *Proc. of the 23rd ACM/IEEE Design Automation Conference*, DAC '86, pages 474–480, Piscataway, NJ, USA, 1986. IEEE Press.

[75] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[76] T. Mogensen and P. Sestoft. Partial evaluation An article for Encyclopedia of Computer Science and Technology Version 1.01 of 1996-04-09, 1996.

[77] S.S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[78] M. Mukherjee and R. Vemuri. A Novel Synthesis Strategy Driven by Partial Evaluation Based Circuit Reduction for Application Specific DSP Circuits. In *Proc. of the 21st International Conference on Computer Design*, ICCD '03, pages 436–, Washington, DC, USA, 2003. IEEE Computer Society.

[79] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 1999.

[80] H.R. Nielson and F. Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[81] University of California at Riverside. ROCCC 2.0 User's Manual - Revision 0.5, `http://roccc.cs.ucr.edu/documentation/`, October 2011.

[82] S. Palnitkar. *Verilog HDL: a guide to digital design and synthesis.* Number Bd. 1. SunSoft Press, 2003.

[83] P.R. Panda, N.D. Dutt, and Nicolau A. Exploiting off-chip memory access modes in high-level synthesis. In *Proc. of the 1997 IEEE/ACM international conference on Computer-aided design*, ICCAD '97, pages 333–340, Washington, DC, USA, 1997. IEEE Computer Society.

[84] B.M. Pangrle and D.D. Gajski. Design Tools for Intelligent Silicon Compilation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(6):1098–1112, 1987.

[85] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[86] G.L. Peterson. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[87] The Flex Project. flex: The Fast Lexical Analyzer, `http://flex.sourceforge.net/`.

[88] P. Raghavan, H. Shachnai, and M. Yaniv. Dynamic Schemes for Speculative Execution of Code. In *Proc. of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[89] N. Rotem. C-to-Verilog Website, `http://www.c-to-verilog.com`, October 2011.

[90] N. Rotem and Y.B. Asher. Synthesis for Variable Pipelined Function units. In *6th International System-on-Chip (SoC) Conference*, SOC 08, 2008.

[91] A. Rushton. *VHDL for Logic Synthesis*. John Wiley & Sons, 2011.

[92] B. Schneier. *Angewandte Kryptographie - Der Klassiker. Protokolle, Algorithmen und Sourcecode in C*. Pearson Studium, München, 2005.

[93] Philips Semiconductors. *I2S bus specification*, February 1986.

[94] Philips Semiconductors. *The I2C-bus specification*, January 2000.

[95] J. Stevens. Hybridthreads Compiler: Generation of Application Specific Hardware Thread Cores from C. In *15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA 07, 2007.

[96] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[97] J. Teich, J. Teich, and C. Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer, 2007.

[98] S. Thompson and A. Mycroft. Bit-level partial evaluation of synchronous circuits. In *Proc. of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '06, pages 29–37, New York, NY, USA, 2006. ACM.

[99] H. Trickey. Flamel: A High-Level Hardware Compiler. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(2):259–269, 1987.

[100] C.-J. Tseng and D.P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 5(3):379–395, 1986.

[101] Stanford University. The SUIF 2 Compiler System, `http://suif.stanford.edu/suif/suif2/`, May 2011.

[102] T.C Wilson, G.W. Grewal, and D.K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *ICCD*, pages 581–586. IEEE Computer Society, 1994.

[103] B. Yang, N. Joshi, and R. Karri. A constant array multiplier core generator with dynamic partial evaluation architecture selection. In *Proc. of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 280–280, New York, NY, USA, 2005. ACM.

[104] D.C. Zaretsky, G. Mittal, R.P. Dick, and P. Banerjee. Balanced Scheduling and Operation Chaining in High-Level Synthesis for FPGA Designs. In *Proc. of the 8th International Symposium on Quality Electronic Design*, ISQED '07, pages 595–601, Washington, DC, USA, 2007. IEEE Computer Society.

Lebenslauf

Name, Vorname:	Manteuffel, Henning
Geburtsdatum:	19.01.81
Geburtsort:	Hamburg

1987-1991	Grundschule Scheeßeler Kehre in Hamburg
1991-2000	Alexander-von-Humboldt-Gymnasium in Hamburg Abitur
09.2000-06.2001	Bundeswehr Gem. Lazarettregiment in Leer
10.2001-10.2003	Grundstudium an der Technischen Universität Hamburg-Harburg Studiengang Diplom Informatik-Ingenieurwesen Vordiplom
10.2003-01.2007	Hauptstudium an der Technischen Universität Hamburg-Harburg Studiengang Diplom Informatik-Ingenieurwesen Diplom
01.2007-12.2009	Berufstätigkeit bei der TuTech GmbH
01.2010-09.2011	Wissenschaftlicher Mitarbeiter an der Technischen Universität Hamburg-Harburg am Institut für Rechnertechnologie