TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Steuerwerkssynthese für Datenflussprogramme

BachelorThesis

von

Aso Saleem

22. September 2021

Technische Universität Kaiserslautern,
Department of Computer Science,
67663 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
Msc. Julius Roob

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Steuerwerkssynthese für Datenflussprogramme" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 22.9.2021

Aso Saleem

Abstract

Due to the increasing demand for high-level synthesis tools used in the compilation of programming languages to hardware languages, this thesis deals with the conception of a code that takes the Mini C language as a starting point for the translation into Verilog. The goal is to design circuits in order to be able to subject the functionality and validity of these to a verification process before they go into material realization. Related approaches dealing with this topic are already available. This work uses the MiniC language as a starting point, since it is a slimmed down form of the C programming language and therefore has less function and a reduced complexity. To ensure an equivalent representation of our source language, MiniC is translated into a control data flow graph, using the MiniC Compiler Tool. Here, the basic blocks of the control data flow graph and their transitions are considered as a finite state machine.This process is represented in verilog with a switch case. Each case is a basic block, all information in a BB is translated sequentially, which ensures that each entry in the BB is also found in the corresponding case in Verilog. Therefore, it is particularly well suited to present a clear visualization process of the final circuit output to Verilog. In order to reduce the error-proneness of the above translation process, we first trace the genesis of compiling using the MiniC compiler tool, and then choose for us the tailor-made and appropriate solution, which we elaborate in the form of the use of a control data flow graph. The goal of this paper is to present an efficient, application-oriented solution that can be validated and checked for maximum traceability at each step.

Zusammenfassung

Aufgrund von steigendem Bedarf an High-Level-Synthesis Tools, die im Compiling von Programmiersprachen zu Hardwaresprachen eingesetzt werden, befasst sich die vorliegende Arbeit mit der Konzeption eines Codes, der die Sprache Mini C als Ausgangspunkt für die Übersetzung in Verilog nimmt.
Ziel ist es, finale Schaltungen zu entwerfen, um die Funktionalität und Validität zu prüfen, bevor die Schaltungen materiell realisiert werden. Verwandte Ansätze, die sich mit eben dieser Thematik befassen, liegen bereits vor. Diese Arbeit verwendet die Sprache MiniC als Ausgangspunkt, da diese eine verschlankte Form der Programmiersprache C ist und dementsprechend weniger Funktion und eine damit einhergehende reduzierte Komplexität aufweist. Um eine äquivalente Repräsentation von unserer Ausgangssprache in Verilog zu gewährleisten, gehen wir hier folgendermaßen vor: MiniC wird mittels des MiniC Compiler Tools in einen Kontrolldatenflussgraph übersetzt, hierbei werden die Basicblocks des Kontrolldatenflussgraphen und ihre Übergänge als endlicher Automat betrachtet. Dieser Prozess wird in Verilog mit einem Switch-Case repräsentiert. Jeder Case ist ein Basicblock, alle Informationen in einem BB werden hierbei sequenziell übersetzt, was gewährleistet, dass jeder Eintrag im BB auch im entsprechenden Case in Verilog zu finden ist. MiniC eignet sich besonders gut, um die Funktionsmechanismen der Übersetzung Schritt für Schritt nachvollziehen zu können und das Endprodukt unserer Übersetzung in Verilog (die finale Schaltung) übersichtlich zu visualisieren. Um die Fehleranfälligkeit des Übersetzungsprozesses zu reduzieren, vollziehen wir zunächst die Genese des Compiling mittels des MiniC Compiler Tools nach, um dann für uns die maßgeschneiderte und passsende Lösung zu wählen, die wir inform der Verwendung eines Kontrolldatenflussgraphen erarbeiten. Ziel der vorliegenden Arbeit ist es, eine effiziente, anwendungsorientierte Lösung zu präsentieren, die sich validieren lässt und in jedem Schritt auf größtmögliche Nachvollziehbarkeit hin überprüfen lässt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

KDFG Kontrolldatenflussgraph

BB Basicblock

1. Einführung

Für diverse Programmiersprachen existiert bereits die Möglichkeit, Software in Hardware durch automatisierte Prozesse (Tools) übersetzen zu lassen. Hardware-Ingenieure haben einen steigenden Bedarf an Übersetzungstools für Software-Programmiersprachen, da sie diese zur Optimierung der Kosten bei der Erstellung digitaler Schaltungen benötigen. Der zunehmende Bedarf resultiert aus der vermehrten Nutzung von neu aufkommenden Programmiersprachen, die sukzessive Compiler notwendig machen, da diese für die Übersetzung von Softwarebeschreibungssprache in Hardwarebeschreibungssprache unerlässlich sind.

Der zunehmende Einsatz von FPGAs (Field Programmable Gate Arrays), welche im Gegensatz zu MPUs (Memory Protection Units) nicht mehr nur über ihre Software definiert sind und damit verbundene Einschränkungen nicht aufweisen, steigert simultan den Bedarf an Bauplänen für Schaltungen. Da bei FPGAs die Hardware der Aufgabe entspricht, die nicht mehr im klassischen Sinne von der Software definiert werden muss, können FPGAs die Ausführung einer Aufgabe mit spezifisch für diese Aufgabe entwickelte Hardware lösen. Dadurch können sie nicht nur zeitkritische Netzwerkfunktionen beschleunigen und die Effizienz und Geschwindigkeit steigern, sondern auch mit paralleler Architektur aufwarten, die gewährleistet wird, indem jede Codezeile in der FPGA-Software gleichzeitig ausgeführt wird. Die Echtzeitkonfiguration vieler FPGAs sorgt dafür, dass die Konfiguration genau mit der jeweiligen Arbeitslast des Benutzers übereinstimmt, daher gibt es bei FPGAs keine vordefinierten Konfigurationen. FPGAs sind besonders gefragt in Hochleistungsanwendungen, wie z.B. Rechenzentren. Durch die Modellierung der Schaltung können Fehler frühzeitig erkannt werden und somit ein funktionierender Ablauf der vielen parallelen Prozesse, die in den FPGAs ausgeführt werden, garantiert werden.

Die Verwendung einer High-Level-Language (F#) um das gegebene Programm (in diesem Fall Mini C) in eine Hardware-Beschreibungssprache (in diesem Fall Verilog) zu übersetzen ist auch Grundlage der hier bearbeiteten Aufgabenstellung. Ein HLS Tool (Compiler tool) wie es in dieser Arbeit vorgestellt wird gab es bisher für die Programmiersprache Mini C nicht, sodass die Übersetzung manuell erfolgen musste. Das hier vorgestellte HLS Tool soll dies überflüssig machen und demzufolge Prozesse optimieren, vereinfachen und effizienter gestalten. Die vorherige Konzeption der zu erstellenden Hardware in Form eines Bauplans (Schaltung) ist essenziell, da kostspielige und teilweise irreversible Fehler den Erstellungsprozess massiv verlangsamen und beeinträchtigen

können.

In dem hier vorliegenden Ansatz habe ich ein bereits bestehendes Mini-C- Programm verwendet, dieses durch ein Tool (Mini C Compiler) in einen Datenkontrollflussgraph konvertiert, sukzessive dann den Datenkontrollflussgraph in die Hardware- Beschreibungssprache Verilog übersetzt, um die finale Konzeption der Schaltungen sichtbar zu machen. Die hohe Fehleranfälligkeit des manuellen Übersetzungsprozesses von Softwarebeschreibungssprache in Hardwarebeschreibungssprache resultiert in der Erarbeitung automatisierter Prozesse. Der manuelle Übersetzungsprozess ist darüber hinaus sehr zeitintensiv und bedarf einer speziellen Ausbildung, was den Kreis der Anwender demzufolge klein hält. Im Folgenden erläutere ich in aller gebotenen Kürze zunächst die Sprache MiniC, danach gehe ich auf den in meinem Algorithmus verwendeten Kontrolldatenflussgraph ein, erläutere dessen Eigenschaften und erkläre, weshalb dessen Verwendung einen besonders hohen Stellenwert in der Konzeption meines Algorithmus einnimmt. In Kapitel 3.1 gehe ich hier zunächst auf das zum Verständnis geforderte Vorwissen ein, dieses wird dann anhand eines Beispiels in Kapitel 3.2 ausführlich durchexerziert und angewandt.

2. Stand der Technik

Es gibt einige Compiler, die die Sprache C, die Similaritäten zu MiniC aufweist, in Verilog übersetzen. Wie oben bereits erwähnt ist der Bedarf an graphischer Darstellung für Schaltungen im Bereich Hardware-Entwicklung und Konzeption sehr hoch und daher gibt es entsprechend viel Forschungsliteratur zu dieser Thematik. Auf die Genese dieses Bereichs kann aus zeitlichen Gründen nicht eingegangen werden, daher haben wir ein Paper ausgewählt, dass sich exemplarisch mit der Übersetzung von C in VHDL beschäftigt.

Zu Anfang gehen wir kurz auf die Abhandlung von De Micheli [De 99] ein. De Micheli konstatiert in seiner wissenschaftlichen Arbeit, dass Hardwareschaltungen sich durch Simulation schnell bewerten lassen und diese direkt kompiliert werden können, wenn eine Softwarelösung gesucht wird. Zur damaligen Zeit hat man versucht Schaltungen mit Programmiersprachen wie C zu simulieren, was einige Nachteile und Probleme mit sich brachte. De Micheli führt aus, dass Hardwareschaltungen einen hohen Grad an Parallelität ausführen können und Programmiersprachen wie C demgegenüber auf Sequentialität ausgelegt sind. Darüber hinaus beinhalten besagte Schaltungen strukturelle Informationen, die in C nicht gegeben sind. Ebenso kann C keine Clock darstellen, die jedoch aufgrund der Leistungs- und Flächenanforderung gegeben sein muss. Dies illustriert, dass bereits 1999 der Bedarf an Übersetzungslösungen, die eine Hardwareschaltung modellieren, vorhanden war. De Micheli postulierte in seiner damaligen Abhandlung, dass die nächste Generation von Hardwareschaltungen und die entsprechenden Entwicklungswerkzeuge (um Hardwaresynthesis zu betreiben, e. A.) wahrscheinlich von großen, lokalen eingebetteten DRAM-Arrays profitieren werden. Diese Vermutung hat sich bestätigt, da wir mittlerweile mit sehr viel größeren DRAM-Arrays arbeiten, die meistens lokal eingebettet sind.

Zusammenfassend können wir feststellen, dass wir uns mittlerweile in einem Stadium befinden, in dem C/C++ Modelle in Hardware synthetisiert werden können. Dies erleichtert die Migration von Softwaremodellen in Hardware und vermeidet Hardware-Spezifikation auf Softwareebene, was die bereits oben beschriebenen Vorteile zur Folge hat.

Die wissenschaftliche Arbeit von Jiang Long und Robert Brayton [LB15] modelliert eine konkrete Übersetzung von C in Verilog und verifiziert diese anhand einer Verifikation mittels des Verilator Tools, das Verilog in C++ konvertiert. Die Übersetzung von C in Verilog geht hier in zwei Schritten vonstatten. Im ersten Schritt wird der C Code in LLVM Bytecode dargestellt.

Man benutzt den LLVM Compiler, um LLVM Bytecode zu generieren. Da LLVM-Bytecode in SSA Form ist, gestaltet sich die sukzessive Übersetzung in Verilog unkompliziert. Bei der Betrachtug des LLVM Bytecodes als SSA Programm mit Basicblocks als Knoten und Verzweigungen bzw. Rücksprünge als Kanten, kann besagtes SSA-Programm als Kontrollflussgraph betrachtet werden . Die Basic Blocks werden als finite state machine betrachtet und die Knoten in den BBs werden sequenziell übersetzt. In Verilog sind zwei Blocks vorhanden, die den Übersetzungsprozess möglich machen: der always_comb block und der always_ff Block. Der always_comb Block stellt die BBs dar und der always_ff Block stellt die Knoten im BB dar. Basic Blocks können grundsätzlich keine Zyklen enthalten, daher können die Knoten hier sequenziell abgearbeitet werden. Die Autoren verwenden anschließend das Tool Verilator zur Validierung des Übersetzungsergebnisses, indem Verilog in C++ konvertiert wird. Als nächster Schritt lässt sich der C-Code aus dem gewonnen C++ Code regenerieren, um das Ergebnis final zu kontrollieren [Sch12].

Bei unserer Übersetzung haben wir eine sehr ähnliche Herangehensweise gewählt, jedoch wird statt der LLVM Bytecode Darstellung von uns ein Kontrolldatenflussgraph gewählt, der sich mittels unseres MiniC Compiler Tools ausgeben lässt. Die Basic Blocks und ihrer Verzweigung/Rückkopplung werden bei uns ebenso wie bei Long und Brayton als finite state machine betrachtet, allerdings haben wir die Darstellung nicht durch die zwei Blocks always comb und always ff gelöst. Wir verwenden stattdessen einen Switch case block, der pro positivem Takt die Schleife einmal abgeht. Darüber hinaus bestehen selbstverständlich Unterschiede, die aus den differierenden Softwaresbeschreibungssprachen resultieren, die wir verwenden. Die Autoren verwenden C, in meiner Arbeit verwende ich MiniC. MiniC ist eine einfachere Form der Sprache C und hat einige Funktionen nicht die C besitzt. In Kapitel 3.1.1 wird die Sprache MiniC spezifiziert.

Abschließend lässt sich feststellen, dass der im Zeitverlauf gewachsene Bedarf an Übersetzungslösungen vermutlich nicht abreißen wird und Übersetzungslösungen daher noch stärker differenziert und optimiert werden. Bei Long und Brayton haben wir illustrativ gesehen, dass die Übersetzung schematisch in zwei Schritten abläuft, die unserem Übersetzungsmodell ähneln, sich aber in der konkreten Ausführung unterscheiden. Es bleibt also festzuhalten, dass die konkrete inhaltliche und technische Ausgestaltung der gewählten Übersetzungslösung maßgeblich von den Eigenschaften der Ausgangssprache, aus der übersetzt werden soll, abhängt.

3. Algorithmus

Mit diesem Algorithmus lässt sich ein beliebiges MiniC Programm in Verilog übersetzen. Verilog ist eine Hardwarebeschreibungssprache, mittels derer eine Synthese erfolgt, die die Voraussetzung für die Erstellung eines Bauplans einer Schaltung ist. Ursprünglich wollten wir den MiniC Code in ein Datenflussprogramm (DPN) übersetzten, sukzessive das Datenflussprogramm in eine Schaltung. Allerdings fehlte dem DPN der Kontrollfluss.

Um eine Übersetzung vornehmen zu können ist ein Kontrollfluss jedoch essentiell, da dieser für die strukturierte Ausgabe und Gruppierung der Daten sorgt. Alternativ könnte man den Schritt über DPNs gehen, allerdings würde eine derartige Implementierung den Rahmen der Arbeit sprengen, da sämtliche Aufteilungen und Gruppierungen manuell erfolgen müssten. Die daraus resultierende Komplexität wäre in dieser Arbeit unsachgemäß. Eine manuelle Gruppierung der Knoten wäre hier erforderlich gewesen; Dies hätte die Komplexität der Aufgabe deutlich erhöht. Dabei hat das Framework "MiniC Compiler"die Funktion, sich einen Kontrolldatenflussgraphen ausgeben zu lassen, der die benötigten Eigenschaften bereits besitzt. Aus o.g. Gründen lassen wir uns daher einen Kontrolldatenflussgraphen ausgeben, und nehmen diesen als Ausgangspunkt für die Übersetzung in die Hardwarebeschreibungssprache Verilog. Genauer erläutert wird der Kontrolldatenflussgraphen im Abschnitt 3.1.2.

3.1. Allgemeine Erklärung

3.1.1. MiniC

Der Algorithmus ist für die Sprache MiniC entworfen worden. MiniC ist eine vereinfachte Form der Programmiersprache C und wurde von der Arbeitsgruppe „Embedded Systems“ des Fachbereichs Informatik der Technischen Universität in Kaiserslautern entworfen. Diese Sprache soll Studierenden den Umgang mit Prozessoren und Compilern näherbringen. Dadurch, dass die Sprache MiniC nicht alle Funktionen bzw. Eigenschaften von C besitzt und damit in ihrer Komplexität reduziert wurde, gestaltet sich die Arbeit mit MiniC simpler.

Beispielsweise ist MiniC im Gegensatz zu C nicht in der Lage dynamische Ressourcenallokation zu verwalten. Darüber hinaus kennt MiniC nur drei Variablentypen natürliche Zahlen, Ganze Zahlen und Boolean (nat, int und bool). MiniC kann ebenso wie C Vektoroperationen, Multireading und predicated execution, jedoch im Gegensatz zu C keine rekursiven Funktionen. Der Compiler übersetzt den eingegebenen MiniC Code in Abacus, damit lassen sich einige

Graphen anzeigen. Für den von mir entwickelten Algorithmus war die Funktion „print controll dataflow graph“ essentiell. Rekursive Funktionen können von MiniC nicht übersetzt werden, da Funktionesaufrufe von dem Compiler durch die Implementierung der Funktion ersetzt werden(inlineing).

Der Folgenden Codeausschnitt stellt ein vollständiges MiniC Programm dar, welches die n-te Fibonacci Zahl ausgibt, wobei n eine natürliche Zahl ist.

```
function fibonacci(nat n) : nat{
    nat i,f1,f2,fn;
    if(n==1 | n==2){
        fn=1;
    } else{
        f1=1;
        f2=1;
        i=2;
        while(i<n){
            i=i+1;
            fn=f1+f2;
            f2=fn;
        }
    }
    return (fn);}}
```

3.1.2. Kontrolldatenflussgraph

Ein Kontrolldatenflussgraph ist eine hybride Form aus einem Datenflussgraph und einem Kontrollflussgraph und vereint sowohl die Eigenschaft des Datenflussgraphs zur bildlichen Veranschaulichung sequenzieller Abläufe sowie die Kontrollfunktion des Kontrollflussgraphs, die eine nach Reihenfolge geordnete Darstellung der Variablen ermöglicht [Gaj+12].

Ein Kontrolldatenflussgraph hat Knoten und Kanten. Verschiedene Knoten werden in sogenannte Basicblocks gruppiert. Erst wenn alle Knoten in einem Basicblock abgearbeitet sind, kann sukzessive der nächste Basicblock abgearbeitet werden. Die ausgehenden Kanten eines Knotens, die eine Fallunterscheidung betreiben, sind schwarz gefärbt. Die grünen Kästen sind die Basicblocks, die gelben Rechtecke sind die Knoten. Knoten bestehen aus einer eindeutigen ID und einem Value (siehe Abbildung 3.1). Es gibt 11 verschiede Aktionen, die ein Konten ausführen kann; diese CMDtypes haben für die spätere Übersetzung maßgebliche Bedeutung [Gaj+12].

Ziel ist es, diesen Kontrolldatenflussgraphen zu generieren, indem man das Tool ”MiniC Compiler”nutzt, um sich diesen Kontrolldatenflussgraphen ausgeben zu lassen. Mit den gegebenen Informationen aus unserem Kontrolldatenflussgraphen generieren wir nun eine äquivalente Beschreibung in Verilog. Die BBs in einem Kontrolldatengflussgraphen werden als endliche Automaten (Final State Machine) betrachtet. BBs enthalten in sich keine Zyklen, dies

macht die Übersetzung in sequenziellen Verilog Code möglich. Die Datenstruktur ist wie folgt repräsentiert: Map $<$ int, Basicblock $>$. Hierbei ist der Key die ID des ersten Knotens im BB. Der Datentyp Basicblock besteht aus folgenden Attributen:

```
type BasicBlock= {rdVars: Set<string>
                  wrVars: Set <string>
                  lineBegin: int
                  lineEnd: int
                  cmdProg: Map<int,CmdType>
                  actDepGraph: Map<int,Set<int>>
                  nextBBs: Set<int>}
```

- rdVars : Set $<$ string $>$ ist ein Set aus allen zu lesenden Variablen in einem BB.
- wrVars : Set $<$ string $>$ ist ein Set aus allen beschreibbaren Variablen in einem BB.
- lineBegin : int ist die ID des ersten Knotens in einem BB.
- lineEnd : int ist die ID des letzen Knotens in einem BB.
- cmdProg : Map $<$ int, CmdType $>$ Die Keys sind die IDs der Knoten. CmdTypes sind die Befehle, die ein Knoten ausführen kann. In unserem Tool gibt es 11 verschiedene CMDTypes. Nun kommen wir zur Erläuterung der CmdTypes.
  - CmdAssert(x) prüft ob der Wert x ungleich null ist.
  - CmdCopy(y, x) weist y den Wert x zu.
  - CmdSglAssign(y, x1, op2, x2) Der binäre Operator Ops2 wird auf die Operanden x1 und x2 angewendet und in y geschrieben , die Ops2 werden im Folgenden näher erklärt. z.b Addition zweier Zahlen x1 + x2 = y
  - CmdDblAssign(y1, y2, x1, op2, x2) Der binäre Operator Ops2 wird auf die Operanden x1 und x2 angewendet und in y1, y2 geschrieben z.b x1 + x2 = y1,y2
  - CmdAccessArr(y, x1, x2) weist y den Wert im Array x1[x2] zu.
  - CmdAssignArr(y, x1, x2) weist y[x1] den Wert x2 zu.
  - CmdAssignCnd(y, c, x1, x0) weist y den Wert x1 zu, falls die Bedingung c wahr ist , ansonsten den Wert x0.
  - CmdGoto(i) ist ein Sprungbefehl und symbolisiert, dass zu Knoten i gesprungen werden soll.
  - CmdIfGoto(x, i) ist ein bedingter Sprungbefehl und symbolisiert, dass zu Knoten i gesprungen werden soll, falls die Bedingung x erfüllt ist.

  - CmdReturn(x) Gibt den Wert x zurück; dies wird nicht verwendet da wir keine Funktionen haben.
  - CmdSync signalisiert dem Thread, dass er warten soll, bis die geteilten Variablen sichtbar gemacht werden für andere Threads.

- actDepGraph : Map < int, Set < int >> Enthält die Abhängigkeiten zwischen den Knoten in einem BB.
- nextBBs : Set < int > nextBBs ist ein Set aus den nachfolgenden BBs. Das Set kann maximal 2 Zahlen enthalten.

Ops2 sind binäre Operatoren; der Ergebnistyp der Operation hängt von dem Operator ab:

- Add : Steht für eine Addition.
- Sub : Für eine Subtraktion.
- Mul : Für eine Multiplikation.
- Div : Für eine Division.
- Mod : Für die Modulo Rechnung.

Nun hängt man noch einen Buchstaben an, um die einzelnen Operanden zu unterscheiden und damit die Values dementsprechend anzupassen. z.B. steht AddN für eine Addition zweier natürlicher Zahlen (nat); das Ergebnis ist hierbei ebenfalls eine natürliche Zahl. Den Buchstabe Z verwendet man für Integer. Zusätzlich gibt es Vergleichsoperationen:

- Les : Kleiner
- Leq : Kleiner oder gleich
- Eqq : Gleich
- Neq : Ungleich

Hier hängt man auch wieder einen Buchstaben Z oder N an, um zu unterscheiden ob die einzelnen Operanden integer oder nat sind. Nun gibt es noch 4 weitere Operationen:

- AndB : Dies ist eine Konjunktion zweier Booleans
- OrB : Disjunktion zweier Booleans
- EqqB : Prüft, ob die Booleans gleich sind
- NeqB : Prüft, ob die Booleans ungleich sind

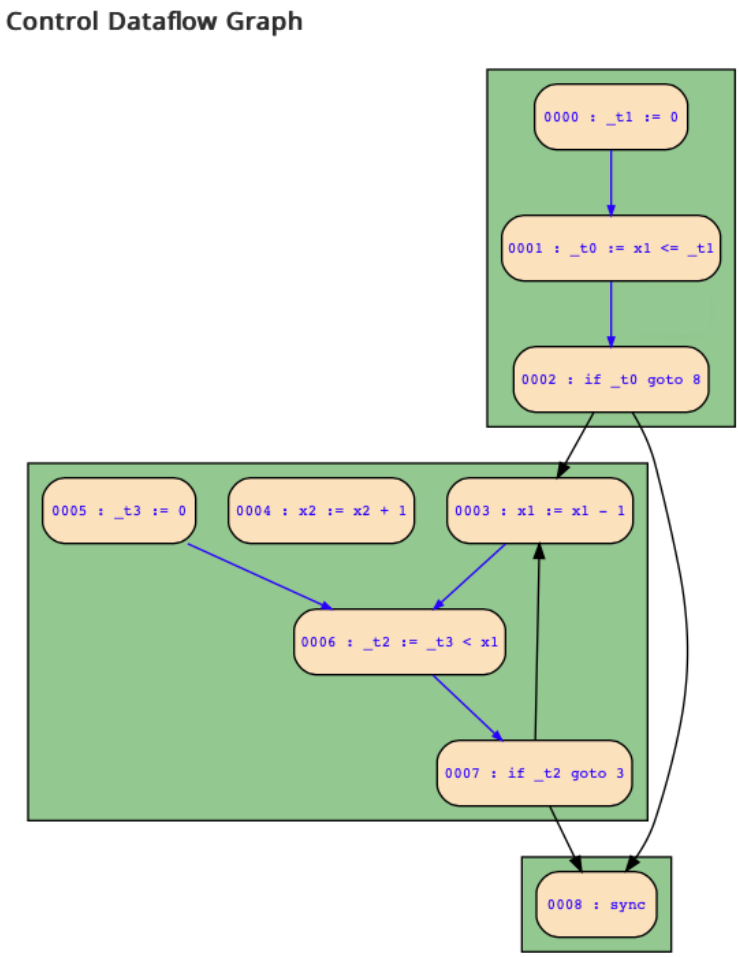

Abbildung 3.1.: *Graphische Darstellung des Kontrolldatenflussgraphen.*

3.1.3. Übersetzung in Verilog

Zudem hatten wir das Programm ursprünglich in ein Steuerwerk und ein Operationswerk aufgeteilt, wobei das Steuerwerk die BBs und die Sprünge zwischen diesen darstellte. Das Steuerwerk sollte die einzelnen BBs als einen Automaten darstellen. Die BBs sind die einzelnen Zustände, hier konnte man die Übergänge problemlos übernehmen. Dies haben wir aber letzendlich mit einem Switch-Case im Verilog Code gelöst, um die Fehleranfälligkeit zu minimieren. Die Kommunikation zwischen Steuerwerk und Operationswerk wäre aufwändiger und unübersichtlicher als der von mir gewählte Lösungsansatz; Das Steuerwerk hätte die Sprünge zwischen den BBs koordiniert und das Operationswerk die Aktionen im BB ausgeführt.

Die koordinatorische Leistung des Steuerwerks war abgestimmt auf die Exekutivfunktion des Operationswerks, das für die Informationsspeicherung und -verarbeitung der in den BBs enthaltenen Daten zuständig war. Jetzt

konnte die Infomationsweitergabe bzw. Übertragung zwischen diesen beiden Systemen durch den Switch-Case in Verilog simplifiziert werden, indem diese vormals autonomen Einheiten fusioniert wurden. Die Entscheidung, das Steuerwerk und das Operationswerk zusammenlegen, wurde nach gründlicher Recherche getroffen, exemplarisch sei hier die Dissertation von Klaus Schneider(1996) genannt [Sch96].

Die einzelnen BBS werden als Case im Code repräsentiert, die Information und Verarbeitung finden nun in den einzelnen Cases statt (dies war vormals Aufgabe des Operationswerkes). Nun haben wir eine grobe Übersicht der qualitativen Inhalte des Verilog Codes kurz erläutert, was uns zur spezifischeren Erläuterung der Übersetzung in Verilog führt. Mit dem berechneten Kontrolldatenflussgraph haben wir die Ausgangssituation für die weitere Übersetzung in Verilog geschaffen.

Im nächsten Schritt generieren wir mittels der gegebenen Schnittstellen einen äquivalenten Verilog Code. Hierfür benötigen wir folgendes Programmunspezifisches Codestück: Es beinhaltet ein Verilog Modul, welches nicht synthetisierbar ist; dieses Modul benötigen wir ausschließlich zur Ausgabe und zur Bereitstellung der Clock, die normalerweise von einem anderen Modul bereitgestellt wird. Im Verilog Modul „program“ befindet sich die Übersetzung des ursprünglichen MiniC Programms.

```
printfn "module main";
printfn "reg[0:0] clock = 0;"
printfn "wire done;"
printfn "program p (clock, done);"
printfn "always #10 clock = ~clock;"
printfn "always @ (posedge done) begin"
printfn "$display (\"Finished\");"
printfn "$finish;$"
printfn "end"
printfn "endmodule"

printfn "module program("
printfn "input wire clock,"
printfn "output wire done"
printfn ");"
```

Der nächste Schritt besteht darin, alle Variablen aus dem generierten KDFG zu extrahieren und diese in ein Set einzufügen. Zuerst hab wir alle rdvars und wrvars ausgelesen; hier war eine Typisierung hinsichtlich der Variablen nicht möglich. Daher extrahieren wir alle Variablen, die der Kontrolldatenflussgraph benötigt, samt des zugehörigen Typens der Variablen, aus locDecls.
Der Name der Variable ist als String abgespeichert und der Typ als TypeC; TypeC wird unten näher erläutert.
Diese erhalten wir mit dem Aufruf:

```
MiniC.CodeGenAbacus.CompiledThread.locDecls: (string * TypeC) []
```

Jetzt muss lediglich noch unterscheiden werden, welchen Typ die Variable hat. Bool, int und nat können ohne weitere notwendige Schritte übersetzt werden wie folgt:

```
TypeC.Cnat -> printfn "reg[16:0] %s = 0;" variable
TypeC.Cint -> printfn "reg[16:0] %s = 0;" variable
TypeC.Cbool -> printfn "reg[1:0] %s = 0;" variable
```

Arrays müssen gesondert betrachtet werden, da sie die zuvor behandelten primitiven Typen enthalten können oder auch mehrdimensional sein können. Dies sieht dann wie folgt aus:

```
| TypeC.Carr ((Some len), elementType) ->
    let mutable elementType = elementType
    let mutable length = len
    let mutable finished = false
    let mutable command = ""
        while not finished do
            match elementType with
                | TypeC.Carr ((Some len),
                   elementType2) ->
                    length <- length * len
                    elementType <- elementType2
                | TypeC.Cnat ->
                    command <- sprintf "reg[16:0] %
                       s [0:%A];" variable (length -
                        1)
                    finished <- true
                | TypeC.Cint ->
                    command <- sprintf "reg[16:0] %
                       s [0:%A];" variable (length -
                        1)
                    finished <- true
                | TypeC.Cbool ->
                    command <- sprintf "reg[1:0] %s
                        [0:%A];" variable (length -
                       1)
                    finished <- true
        printfn "%s" command
```

Im nächsten Schritt iterieren wir über die zuvor extrahierten Variablen und schreiben jede Variable in ein Register. Jetzt benötigen wir noch ein Register für den instruction Pointer IP. Der instruction Pointer hat den Wert der ID des aktuellen BBs.

Zudem brauchen wir einen Platzhalter für einen IP, der zeigt, dass das Programm terminiert. Dann wieder zwei Zeilen programmunspezifischen Code; bei

jeder positiven Änderung der Clock wird der weitere Code ausgeführt. Hier wird die Fallunterscheidung des endlichen Automaten begonnen. Dies sieht im Code so aus:

```
printfn "always @ (posedge clock) begin"
printfn "case (ip)"
```

Wie weiter oben in Abschnitt 3.1.2 erwähnt, mussten noch die Operationen in Verilog übersetzt werden:

```
let operationToString operation =
        match operation with
            | Ops2.AddN -> "+"
            | Ops2.SubN -> "-"
            | Ops2.MulN -> "*"
            | Ops2.DivN -> "/"
            | Ops2.ModN -> "%"
            | Ops2.LeqN -> "<="
            | Ops2.LesN -> "<"
            | Ops2.EqqN -> "=="
            | Ops2.NeqN -> "!="
            | Ops2.AddZ -> "+"
            | Ops2.SubZ -> "-"
            | Ops2.MulZ -> "*"
            | Ops2.DivZ -> "/"
            | Ops2.ModZ -> "%"
            | Ops2.LeqZ -> "<="
            | Ops2.LesZ -> "<"
            | Ops2.EqqZ -> "=="
            | Ops2.NeqZ -> "!="
            | Ops2.AndB -> "&"
            | Ops2.EqqB -> "=="
            | Ops2.OrB  -> "||"
            | Ops2.NeqB -> "!="
```

Für jeden BB generieren wir einen Case, der ausgeführt wird falls der aktuelle instruction Pointer gleich der ID des BBs ist. Danach werden alle Befehle des BBs in diesem Case übersetzt.

```
printfn"begin"
 let basicBlock = entry.Value
        for cmdEntry in basicBlock.cmdProg do
            match cmdEntry.Value with
            | CmdType.CmdAssert var -> printfn "%s
               != 0;" var
            | CmdType.CmdCopy (var1, var2) ->
               printfn "%s = %s;" var1 var2
            | CmdType.CmdAccessArr (resultreg,
               arrayreg, indexreg) -> printfn "%s =
```

```
                %s[%s];" resultreg arrayreg indexreg
            | CmdType.CmdAssignArr (arrayreg,
                indexreg, valuereg) -> printfn "%s[%s
                ] = %s;" arrayreg indexreg valuereg
            | CmdType.CmdAssignCnd (y,c,x1,x0)->
                printfn "%s = %s ? %s : %s;" y c x1
                x0
            | CmdType.CmdGoto gotoi -> printfn "ip
                = %A;" gotoi
            | CmdType.CmdIfGoto (bvar, gotoi) ->
                let nextBBs = Set.toList basicBlock
                    .nextBBs
                if List.length nextBBs <> 2 then
                    failwith "unexpected"
                let altinstruction =
                    if nextBBs.[0] = gotoi then
                        nextBBs.[1]
                    elif nextBBs.[1] = gotoi then
                        nextBBs.[0]
                    else
                        failwith "following
                            basicblock not present"
                printfn "ip = %s ? %A : %A;" bvar
                    gotoi altinstruction
            | CmdType.CmdSync -> printfn "ip = %d;"
                doneIp
            | CmdType.CmdSglAssign (result,
                operand1, operation, operand2) ->
                let operation =
                    operationToString operation
                printfn "%s = %s %s %s;" result
                    operand1 operation operand2
            | CmdType.CmdDblAssign (result1,
                result2, operand1, operation,
                operand2) ->
                let operation =
                    operationToString operation
                printfn "{%s, %s} = %s %s %s;"
                    result1 result2 operand1
                    operation operand2
    printfn "end"
```

Wie oben bereits schon erläutert, haben wir die Übersetzung der Sprünge zwischen den BBs mithilfe des Switch-Case Konstruktes (wie bereits oben beschrieben) in Verilog gelöst. Um einen Sprung auszuführen, wird der Instruction Pointer auf die ID des nächsten BBs gesetzt. Nun finalisieren wir wie in der Syntax von Verilog gefordert wie folgt:

```
printfn "endcase"
printfn "end"
printfn "endmodule"
```

Dadurch, dass alle Knoten und Kanten im KDFG sequentiell übersetzt worden sind wurde sukzessive der MiniC Code vollständig übersetzt. Damit haben wir die Übersetzung in Verilog komplettiert.

3.2. Erklärung des Algorithmus anhand eines Beispiels

3.2.1. Inc/Dec in MiniC

Zum besseren Verständnis gehen wir den Algorithmus mit dem nachfolgenden Beispiel(Incrementer/Decrementer) durch:

```
nat x1,x2;

thread AddIncDec {
    while(x1>0) {
        x1=x1-1;
        x2=x2+1;
    }
}
```

Dieser Code bekommt als Eingabe zwei natürliche Zahlen x_1, x_2 . In Zeile 3 wird geprüft, ob x_1 größer 0 ist; falls diese Bedingung erfüllt ist, zählt das Program x_1 runter und x_2 hoch. Diese beiden Schritte werden so lange wiederholt, bis die Bedingung $x_1 > 0$ nicht mehr erfüllt ist.

3.2.2. Kontrolldatenflussgraph des Inc/Dec

Der dazugehörige Kontrolldatenflussgraph zum oben genannten Code sieht wie folgt aus:

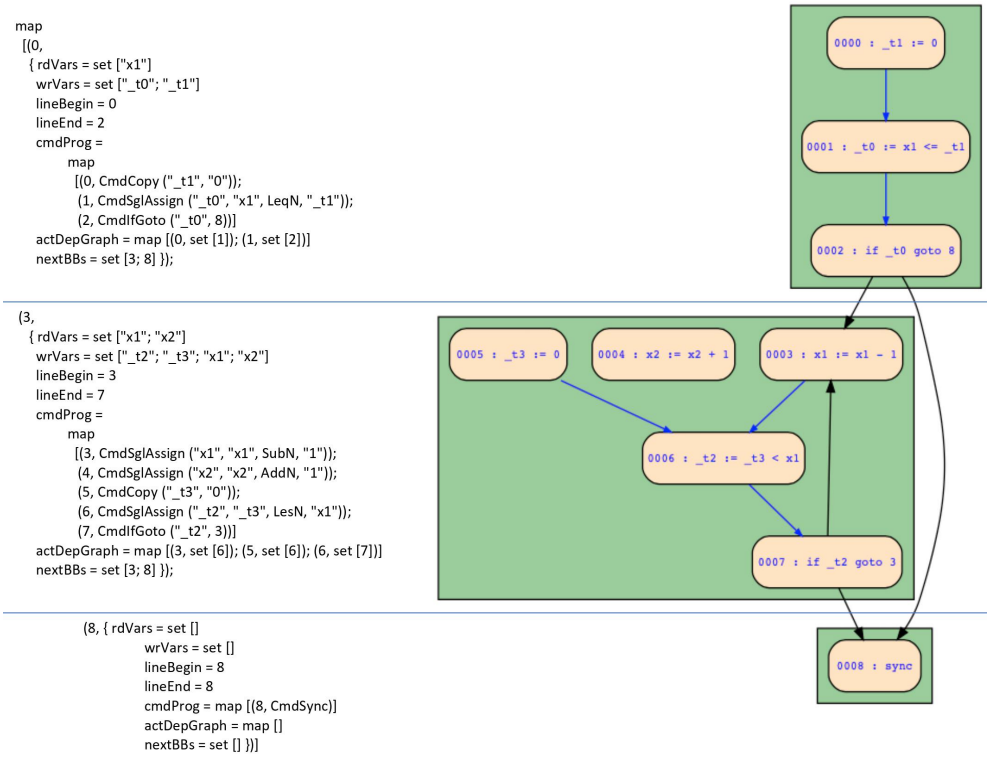

Abbildung 3.2.: *Kontroll Datenfluss Graph (Incrementer/ Decrementer).*

Wir visualisieren das Ganze anhand des INC/DEC Beispiels.
Der Kontrolldatenflussgraph hat 8 Knoten, die in 3 BBs aufgeteilt sind. Die BBs werden nach der Nummer des ersten Knotens im BB benannt. Beispiel Werte $x_1 = 1$ und $x_2 = 3$.

- Basicblock 0
  - Knoten 0: Man nimmt eine Hilfsvariable _t1 und setzt diese auf null. Dies erfolgt mittels des Befehls CmdCopy.
  - Knoten 1: _t0 ist ein Boolean; wenn x1 (in unserem Fall 1) kleiner oder gleich null ist, ist _t0 true. Da x1 = 1 ist, ist _t0 := false (CmdSglAssign[”_t0”,”x1”,LeqN,”_t1”])
  - Knoten 3 Da _t0 false ist, springen wir zu Knoten 3 der im BB3 liegt. Wäre _t0 true würden wir zu BB8 springen und das Programm würde terminieren.
- Basicblock 3
  - Knoten 3: x_1 wird um eins verringert

  - Knoten 4: x_2 wird um eins erhöht
  - Knoten 5: t_3 wird auf null gesetzt
  - Knoten 6: Hier wird dasselbe gemacht wie in Knoten 1: es wird geprüft, ob $x_1 > 0$ gilt _t2 nimmt den Wert „true“ oder „false“ an
  - Knoten 7: wenn _t2 true ist, kehren wir wieder zu Knoten 3 zurück und wiederholen Knoten 3, 4, 5, 6 solange bis $x_1 = 0$ ist. Da in unserem Fall $x_1 = 0$ ist (da es um eins verringert worden ist in Knoten 3) sind wir fertig und das Programm terminiert.

- Basicblock 8: Hier terminiert das Programm.

3.2.3. Übersetzung in Verilog anhand eines Beispiels

Jetzt haben wir einen programmunabhängigen Code generiert, der als Gerüst für den spezifischen Code dient. Nun geht es an die Übersetzung der einzelnen Knoten. Wir haben alle Variablen in Register angelegt. Zudem haben wir die „doneip“ von 9, da der größte Knoten in unserem Graph den Wert 8 hat und wir ihn um eins erhöhen.Das Ganze sieht dann mit unserem Beispiel so aus:

```
module main;
    reg[0:0] clock = 0;
    wire done;
    program p (clock, done);
    always #10 clock = ~clock;
    always @ (posedge done) begin
        $display("Finished");
        $finish;
    end
endmodule

module program(
    input wire clock,
    output wire done
);
reg[16:0] _t0 = 0;
reg[16:0] _t1 = 0;
reg[16:0] _t2 = 0;
reg[16:0] _t3 = 0;
reg[16:0] ip = 0;
assign done = ip == 9;
always @ (posedge clock) begin
```

Jetzt fehlt uns nur noch der Switch-Case, dafür geben wir den Verilog Befehl „case(ip)“ aus und beginnen mit der Key des ersten BB, welcher den Wert 0 hat.

```
CmdCopy ("_t1", "0");
CmdSglAssign ("_t0", "x1", LeqN, "_t1");
CmdIfGoto ("_t0", 8)
```

Das sind die Values der Knoten des BBs0

Die ersten beiden Knoten können wie oben bereits beschrieben, einfach übersetzt werde und sehen dementsprechend folgendermaßen aus:

```
_t1 = 0;
_t0 = x1 <= _t1;
```

Nun kommt der CmdIfGoto Befehl. Wenn _to true ist, setzt man Ip auf 8, ansonsten setzt man dei Ip auf 3. Dies sieht dann aus wie folgt:

```
ip = _t0 ? 8 : 3;
```

Analog zu der Übersetzung der ersten drei Knoten vefahren wir mit den verbleibenden Knoten; hier ist das Ergebnis folgendes:

```
case (ip)
0:
    begin
        _t1 = 0;
        _t0 = x1 <= _t1;
        ip = _t0 ? 8 : 3;
    end
3:
    begin
        x1 = x1 - 1;
        x2 = x2 + 1;
        _t3 = 0;
        _t2 = _t3 < x1;
        ip = _t2 ? 3 : 8;
    end
8:
    begin
        ip = 9;
    end
9:
    begin
```

3.2.4. Auswertung

Das Ziel ein beliebiges MiniC Programm in eine äquivalente Schaltung zu transformieren wurde mit dem oben genannten Algorithmus erreicht. Wenn wir das anhand unseres Beispiels durchdeklinierte MiniC Beispiel nehmen und es als Eingabe für den Algorithmus verwenden, wird uns der äquivalente Verilog Code ausgegeben. Mithilfe des Tools „Yosys Open Synthesis Suite“ können wir uns die Schaltung in codierter Form ausgeben lassen [oAc].

Visualisieren lassen kann man sich besagte codierte Schaltung auf der Webseite: [oAb]. Die Schaltung zu unserem MiniC Programm in Abbildung 3.3. Zwecks einer quantitativen Auswertung des Algorithmus wurde das Programm in Kapitel 4 mehrfach getestet. Diese MiniC Beispiele, die für die Tests verwendet werden, wurden von der „Arbeitsgruppe Embedded Systems“ des Fachbereichs Informatik der TU Kaiserslautern geschrieben. Die Angabe wo diese zu finden sind, befindet sich in der Sektion Test.

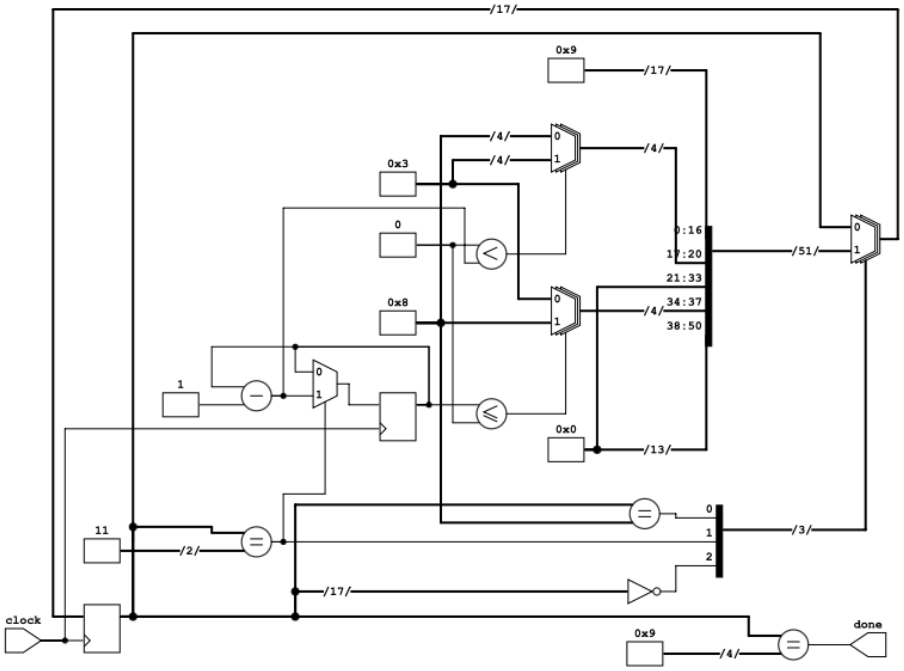

Abbildung 3.3.: *Graphische Darstellung der Schaltung zum MiniC Programm.*

Eben erwähnte Tests haben gezeigt, dass der Algorithmus mit verschiedenen Eingaben eine korrekte Übersetzung vornimmt und diese fehlerfrei funktioniert. Darüber hinaus haben wir noch einige Maßzahlen miteinander verglichen die im Folgenden dargestellt werden, mit folgender Konvention:

- Programm A: INCDEC
- Programm B: Insertion Sort
- Programm C: Fibonacci No.
- Programm D: Computing Square Roots
- Programm E: Euclid's Algorithm

	A	B	C	D	E
Number of wires	18	149	32	22	24
Number of wire bits	239	1760	431	308	342
Number of public wires	8	37	21	11	12
Number of public wire bits	88	597	325	155	172
Number of memories	0	0	0	0	0
Number of memory bits	0	0	0	0	0
Number of processes	0	0	0	0	0
Number of cells	13	128	14	14	16
$add	0	2	1	1	0
$dff	2	15	2	2	3
$div	0	0	0	0	2
$eq	3	45	6	4	4
$le	1	3	0	0	0
$logic_not	1	5	1	1	2
$lt	1	1	1	1	0
$mux	3	37	1	1	4
$pmux	1	16	2	2	1
$reduce_bool	0	0	0	0	1
$reduce_or	0	2	0	0	0
$sub	1	2	0	0	0

4. Tests

An dieser Stelle sei anzumerken, dass die folgenden Beispiel MiniC Programme aus der Ärbeitsgruppe Embedded Systems"des Fachbereichs Informatik der TU Kaiserslautern entnommen wurden. Diese sind hier abrufbar [oAa].

4.1. Insertion Sort

4.1.1. Minic

```
thread InsertionMaxSort {
    [10]nat x;
    nat i,j,yind,yval;
    for(i=0..9) {
        x[i] = 10-i;
    }
    for(i=0..9) {
        j = 0;
        yval = x[j];
        for(j=1..9-i)
            if(x[j]>yval) {
                yval = x[j];
                yind = j;
            }
        x[N-1-i]
        j = 9-i;
        x[yind] = x[j];
        x[j] = yval;
    }
}
```

4.1.2. Verilog

```
module main;
    reg[0:0] clock = 0;
    wire done;
    program p (clock, done);
    always #10 clock = ~clock;
    always @ (posedge done) begin
        $display("Finished");
        $finish;
```

```
    end
endmodule
module program(
    input wire clock,
    output wire done
);
    reg[16:0] x [0:9];
    reg[16:0] i = 0;
    reg[16:0] j = 0;
    reg[16:0] yind = 0;
    reg[16:0] yval = 0;
    reg[16:0] _t0 = 0;
    reg[16:0] _t1 = 0;
    reg[16:0] _t10 = 0;
    reg[16:0] _t11 = 0;
    reg[16:0] _t12 = 0;
    reg[16:0] _t13 = 0;
    reg[16:0] _t14 = 0;
    reg[16:0] _t15 = 0;
    reg[16:0] _t16 = 0;
    reg[16:0] _t17 = 0;
    reg[16:0] _t18 = 0;
    reg[16:0] _t19 = 0;
    reg[16:0] _t2 = 0;
    reg[16:0] _t3 = 0;
    reg[16:0] _t4 = 0;
    reg[16:0] _t5 = 0;
    reg[16:0] _t6 = 0;
    reg[16:0] _t7 = 0;
    reg[16:0] _t8 = 0;
    reg[16:0] _t9 = 0;
    reg[16:0] ip = 0;
    assign done = ip == 42;
    always @ (posedge clock) begin

        case (ip)
        0:
            begin
                i = 0;
                _t1 = 9;
                _t0 = _t1 < i;
                ip = _t0 ? 11 : 4;
            end
        4:
            begin
                _t3 = 10;
                _t2 = _t3 - i;
                x[i] = _t2;
                i = i + 1;
                _t5 = 9;
                _t4 = i <= _t5;
                ip = _t4 ? 4 : 11;
```

```
    end
11:
    begin
        i = 0;
        _t7 = 9;
        _t6 = _t7 < i;
        ip = _t6 ? 41 : 15;
    end
15:
    begin
        j = 0;
        yval = x[j];
        j = 1;
        _t9 = 9;
        _t8 = _t9 - i;
        _t10 = _t8 < j;
        ip = _t10 ? 32 : 22;
    end
22:
    begin
        _t11 = x[j];
        _t12 = _t11 <= yval;
        ip = _t12 ? 27 : 25;
    end
25:
    begin
        yval = x[j];
        yind = j;
    end
27:
    begin
        j = j + 1;
        _t14 = 9;
        _t13 = _t14 - i;
        _t15 = j <= _t13;
        ip = _t15 ? 22 : 32;
    end
32:
    begin
        _t16 = 9;
        j = _t16 - i;
        _t17 = x[j];
        x[yind] = _t17;
        x[j] = yval;
        i = i + 1;
        _t19 = 9;
        _t18 = i <= _t19;
        ip = _t18 ? 15 : 41;
    end
41:
    begin
        ip = 42;
```

```
            end
        42: begin end
        endcase
    end
endmodule
```

4.2. Fibonacci Numbers

4.2.1. MiniC

```
function fibonacci(nat n) : nat {
    nat i,f1,f2,fn;
    if(n == 1 | n ==2) {
       fn = 1;
    } else {
       f1 = 1;
       f2 = 1;
       i = 2;
       while(i<n) {
          i = i+1;
          fn = f1+f2;
          f1 = f2;
          f2 = fn;
       }
    }
    return(fn);
}

thread Fibonacci {
    nat n;
    n = fibonacci(10);
}
```

4.2.2. Verilog

```
module main;
    reg[0:0] clock = 0;
    wire done;
    program p (clock, done);
    always #10 clock = ~clock;
        always @ (posedge done) begin
        $display("Finished");
        $finish;
    end
endmodule
module program(
    input wire clock,
    output wire done
);
reg[16:0] n = 0;
```

```
reg[16:0] i__FC1 = 0;
reg[16:0] f1__FC1 = 0;
reg[16:0] f2__FC1 = 0;
reg[16:0] fn__FC1 = 0;
reg[16:0] _t0 = 0;
reg[16:0] _t1 = 0;
reg[16:0] _t10 = 0;
reg[16:0] _t11 = 0;
reg[16:0] _t12 = 0;
reg[16:0] _t2 = 0;
reg[16:0] _t3 = 0;
reg[16:0] _t4 = 0;
reg[16:0] _t5 = 0;
reg[16:0] _t6 = 0;
reg[16:0] _t7 = 0;
reg[16:0] _t8 = 0;
reg[16:0] _t9 = 0;
reg[16:0] ip = 0;
assign done = ip == 27;
always @ (posedge clock) begin
    case (ip)
    0:
        begin
            _t2 = 10;
            _t3 = 1;
            _t1 = _t2 == _t3;
            _t5 = 10;
            _t6 = 2;
            _t4 = _t5 == _t6;
            _t7 = _t1 || _t4;
            _t8 = 0 == _t7;
            ip = _t8 ? 11 : 9;
        end
    9:
        begin
            fn__FC1 = 1;
            ip = 24;
            end
            11:
            begin
            f1__FC1 = 1;
            f2__FC1 = 1;
            i__FC1 = 2;
            _t10 = 10;
            _t9 = _t10 <= i__FC1;
            ip = _t9 ? 24 : 17;
        end
    17:
        begin
            i__FC1 = i__FC1 + 1;
            fn__FC1 = f1__FC1 + f2__FC1;
            f1__FC1 = f2__FC1;
```

```
                f2__FC1 = fn__FC1;
                _t12 = 10;
                _t11 = i__FC1 < _t12;
                ip = _t11 ? 17 : 24;
            end
        24:
            begin
                _t0 = fn__FC1;
                ip = 26;
            end
        26:
            begin
                n = _t0;
                ip = 27;
            end
        27: begin end
    endcase
    end
    endmodule
```

4.3. Computing Square Roots

4.3.1. MiniC

```
function heron(nat a) : nat {
   nat xold,xnew;
   xnew = a;
   do {
      xold = xnew;
      xnew = (xold + a/xold)/2;
   } while(xnew < xold)
   return xold;
}

thread Heron {
   nat z;
   z = heron(121);
}
```

4.3.2. Verilog

```
module main;
    reg[0:0] clock = 0;
    wire done;
    program p (clock, done);
    always #10 clock = ~clock;
    always @ (posedge done) begin
        $display("Finished");
        $finish;
    end
```

```
endmodule
module program(
    input wire clock,
    output wire done
);
    reg[16:0] z = 0;
    reg[16:0] xold__FC1 = 0;
    reg[16:0] xnew__FC1 = 0;
    reg[16:0] _t0 = 0;
    reg[16:0] _t1 = 0;
    reg[16:0] _t2 = 0;
    reg[16:0] _t3 = 0;
    reg[16:0] _t4 = 0;
    reg[16:0] ip = 0;
    assign done = ip == 11;
    always @ (posedge clock) begin
        case (ip)
        0:
            begin
                xnew__FC1 = 121;
            end
        1:
            begin
                xold__FC1 = xnew__FC1;
                _t2 = 121;
                _t1 = _t2 / xold__FC1;
                _t3 = xold__FC1 + _t1;
                xnew__FC1 = _t3 / 2;
                _t4 = xnew__FC1 < xold__FC1;
                ip = _t4 ? 1 : 8;
            end
        8:
            begin
                _t0 = xold__FC1;
                ip = 10;
            end
        10:
            begin
                z = _t0;
                ip = 11;
            end
        11: begin end
        endcase
    end
endmodule
```

4.4. Euclid's Algorithm

4.4.1. MiniC

```
function euclid(nat a,b) : nat {
```

```
    nat t;
    while(b!=0) {
       t = b;
       b = a % b;
       a = t;
    }
    return a;
}

thread t2 {
    nat x,y,z;
    z = euclid(x,y);
}
```

4.4.2. Verilog

```
module main;
    reg[0:0] clock = 0;
    wire done;
    program p (clock, done);
    always #10 clock = ~clock;
    always @ (posedge done) begin
        $display("Finished");
        $finish;
    end
endmodule
module program(
    input wire clock,
    output wire done
);
    reg[16:0] x = 0;
    reg[16:0] y = 0;
    reg[16:0] z = 0;
    reg[16:0] t__FC1 = 0;
    reg[16:0] _t0 = 0;
    reg[16:0] _t1 = 0;
    reg[16:0] _t2 = 0;
    reg[16:0] _t3 = 0;
    reg[16:0] _t4 = 0;
    reg[16:0] ip = 0;
    assign done = ip == 12;
    always @ (posedge clock) begin
        case (ip)
        0:
            begin
                _t2 = 0;
                _t1 = y == _t2;
                ip = _t1 ? 9 : 3;
            end
        3:
            begin
                t__FC1 = y;
```

```
                y = x % y;
                x = t__FC1;
                _t4 = 0;
                _t3 = y != _t4;
                ip = _t3 ? 3 : 9;
            end
        9:
            begin
                _t0 = x;
                ip = 11;
            end
        11:
            begin
                z = _t0;
                ip = 12;
            end
        12: begin end
        endcase
    end
endmodule
```

5. Fazit

Das Ziel dieser Arbeit war es, ein beliebiges MiniC Programm in eine äquivalente Schaltung zu übersetzen. Zunächst wurden mehrere Ansätze veranschaulicht, die zielführend sind, um dann in einem zweiten Schritt anhand verschiedener Faktoren zu vergleichen welcher Lösungsansatz sich am optimalsten präsentiert. Der gewählte Lösungsansatz wurde zunächst theoretisch erläutert, darüber hinaus wurde das Grundgerüst des Codes mitsamt Funktionsprinzipien beschrieben, um diesen dann anhand eines Beispiels zu veranschaulichen. Einen besonderen Stellenwert nahm die Erläuterung zur Verwendung eines Kontrolldatenflussgraphen ein, da dieser die Qualität des von mir entwickelten Codes maßgeblich verbesserte. Gerade für Studierende ist die theoretische Konzeption und deren praktische Umsetzung für das Verständnis wesentlich.

Hier leistet mein Code einen elementaren Beitrag zur Visualisierung der gebauten Schaltungen und kann so den Lernprozess verständnisorientiert begleiten. Die Übersetzung von Programmiersprachen in Hardwarebschreibungssprachen ist für jeden Anwendungsbereich relevant, der den Bau von Chips voraussetzt, beispielhaft sei hier die Arbeit mit FPGAs die in Rechenzentren zur Leistungs- und Effizienzsteigerung eingesetzt werden. Aus den breiten Applikationsmöglichkeiten und dem sukzessive gesteigerten Bedarf resultiert nicht nur eine praktische Anwendungsoptimierung sondern auch die Notwendigkeit der theoretischen Fundierung in der Wissenschaft der Informatik. Um das Grundprinzip der Übersetzung nachvollziehen zu können ist mein Code geeignet, ebenso bietet die von mir vorgestellte Arbeit Anknüpfungspunkte für weitere Forschungen und forschungspraktische Entwicklungen.

Bei der Wahl meines Ansatzes wurde der Reduktion der Komplexität und der Minimierung der Fehleranfälligkeit besondere Bedeutung beigemessen, da es in der praktischen Umsetzung von immanenter Relevanz ist, anwendungsorientierte Lösungsmöglichkeiten zu präsentieren die universal einsatzfähig sind. Bei der Konzeption des Codes wurde, wie in Abschnitt 3.1.3 beschrieben die Entscheidung getroffen, Steuerwerk und Operationswerk zu fusionieren. Dies bedeutet, dass der Code nicht nur in der Komplexität reduziert wurde, sondern auch effektiv weniger Code vorhanden ist, was darüber hinaus für bessere Nachvollziehbarkeit sorgt.

Die Fusion und damit einhergehende Reduktion sorgt darüber hinaus für eine bessere Übersichtlichkeit. Dadurch, dass die Übersetzung vereinheitlicht ist, ist zusätzlich eine bessere Verständlichkeit gewährleistet. Dies macht auch ei-

ne eventuelle Fehlersuche einfacher. Bei dem von mir entwickelten Code haben wir für jede Variable ein Register angelegt. Hier bestünde noch Entwicklungspotential hinsichtlich der Optimierung des Speicherplatzes. Hier könnte man die anderen Register abgehen und nach Registern Ausschau halten, in denen noch Speicherplatz vorhanden ist. In der Praxis hat man nicht unendlich Ressourcen zur Verfügung, deshalb ist die Optimierung von Speicherplatz ein optionales Themengebiet für weiterführende Arbeiten. Darüber hinaus wäre es eine weitere interessante Weiterentwicklungsmöglichkeit, ein Tool zu implementieren welches prüft, ob der gegebene Verilog Code valide ist.

Ein automatisiertes Tool, hätte den Vorteil, dass die Äquivalenzprüfung nicht mehr manuell mit Tests erfolgen müsste. Hierzu weiterführend Leung, Alan, Bounov und Lerner (2015) in ihrer Abhandlung "C to Verilog translation validation", welche sich allerdings auf die Sprache C bezieht, allerdings einen guten Überblick über die Herangehensweise bietet [LBL15].

Der Mehrwert der Arbeit besteht unter anderem darin, dass ein voll funktionsfähiger Chip mittels des MiniC Compiler Tools für ein MiniC Programm gebaut werden kann. Jede Programmiersprache benötigt ein Compiler Tool um diese entsprechend auch zu verwirklichen.

Da der Prozessor/Chip Bau sehr kostenintensiv ist und theoretische Fehler in der Praxis irreparabel sein können oder explodierende Kosten nach sich ziehen können. Hierbei muss beachtet werden, dass der von mir konzipierte/-entwickelte Code immer genauso qualitativ ist wie das entsprechende MiniC Compiler Tool, hier ist also eine Interdependenz vorhanden. Der anwendungspraktische Mehrwert des von mir entwickelten Codes besteht vor allem darin, Schaltungen vor der materiellen Umsetzung zu modellieren, um etwaige Fehler frühzeitig zu erkennen und durch die gegebene Visualisierung beheben zu können. Damit erleichtert man darüber hinaus die Entscheidungsfindung hinsichtlich des Für oder Widers einer materiellen Umsetzung, indem man zu einer Kosten-Nutzen-Abwägung beiträgt.

Literatur

[Ayg+07] Eduard Ayguadé, Gerald Baumgartner, J Ramanujam und Ponnuswamy Sadayappan. *Languages and Compilers for Parallel Computing: 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers*. Bd. 4339. Springer, 2007.

[De 99] Giovanni De Micheli. "Hardware synthesis from C/C++ models". In: *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*. IEEE. 1999, S. 382–383.

[Gaj+12] Daniel D Gajski, Nikil D Dutt, Allen CH Wu und Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[LB15] Jiang Long und Robert Brayton. "A Simple C to Verilog Compilation Procedure for Hardware/Software Verification". In: (2015). Berkeley Verification and Synthesis Research Center (BVSRC).

[LBL15] Alan Leung, Dimitar Bounov und Sorin Lerner. "C-to-verilog translation validation". In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, S. 42–47.

[oAa] o.A. *MiniC Examples*. `https://es.cs.uni-kl.de/tools/teaching/MiniC-Examples.html`. Accessed: 2021-08-01.

[oAb] o.A. *Netlistsvg*. https://neilturley.dev/netlistsvg/.

[oAc] o.A. *Yosys Open Synthesis Suite*. `http://www.clifford.at/yosys/`. Accessed: 2021-07-18.

[Sch12] Patrick R Schaumont. *A practical introduction to hardware/software codesign*. Springer Science & Business Media, 2012.

[Sch96] Klaus Schneider. "Ein einheitlicher Ansatz zur Unterstützung von Abstraktionsmechanismen der Hardware-Verifikation". Sankt Augustin 1996. (DISKI. Dissertationen zur Künstlichen Intelligenz. 116.) Fak. f. Informatik, Diss. v. 7.7.1995. Diss. 1996.

A. Mein Code

```
open MiniC.Compiler
open MiniC.Types
[<EntryPoint>]
let main argv =
    let exampleProg = "
        nat x1;
        nat x2;
        thread AddIncDec{
            while((0 < x1)) {
                x1 = (x1 - 1);
                x2 = (x2 + 1);
            }
        }
        "
    let compiled = MiniC.IO.ParseMiniCFromString (
       Some System.Console.Out) exampleProg
    let (\_, \_, compiledThreads) = MiniC.
       CodeGenAbacus.CompileMC None "test" (false,
       false, false, false) false compiled
    let compiledThread = compiledThreads.[0]
    let dataflow = MiniC.Compiler.
       ComputeControlDataFlowGraph compiledThread.
       cmdProg
    let basicBlocks = [for basicBlock in dataflow
       -> (basicBlock.Key, basicBlock.Value)]
    printfn "module main;"
    printfn "reg[0:0] clock = 0;"
    printfn "wire done;"
    printfn "program p (clock, done);"
    printfn "always #10 clock = ~clock;"
    printfn "always @ (posedge done) begin"
    printfn "$display(\"Finished\");"
    printfn "$finish;"
    printfn "end"
    printfn "endmodule"
    printfn "module program("
    printfn "input wire clock,"
    printfn "output wire done"
    printfn ");"
```

```
let mutable locals = Set.empty;
for (variable, vartype) in compiledThread.
   locDecls do
    if not (Set.contains variable locals) then
        locals <- Set.add variable locals
        match vartype with
        | TypeC.Cnat -> printfn "reg[16:0] %s =
            0;" variable
        | TypeC.Cint -> printfn "reg[16:0] %s =
            0;" variable
        | TypeC.Cbool -> printfn "reg[1:0] %s =
            0;" variable
        | TypeC.Carr ((Some len), elementType)
           ->
            let mutable elementType =
               elementType
            let mutable length = len
            let mutable finished = false
            let mutable command = ""
            while not finished do
                match elementType with
                    | TypeC.Carr ((Some len),
                       elementType2) ->
                        length <- length * len
                        elementType <-
                           elementType2
                    | TypeC.Cnat ->
                        command <- sprintf "reg
                           [16:0] %s [0:%A];"
                           variable (length - 1)
                        finished <- true
                    | TypeC.Cint ->
                        command <- sprintf "reg
                           [16:0] %s [0:%A];"
                           variable (length - 1)
                        finished <- true
                    | TypeC.Cbool ->
                        command <- sprintf "reg
                           [1:0] %s [0:%A];"
                           variable (length - 1)
                        finished <- true
            printfn "%s" command
for variable in compiledThread.tmpVars do
    printfn "reg[16:0] %s = 0;" variable
```

```
printfn "reg[16:0] ip = 0;"
let doneIp = List.max (List.map (fun (line, _)
   -> line) basicBlocks) + 1
printfn "assign done = ip == %d;" doneIp

printfn "always @ (posedge clock) begin \n"
printfn "case (ip)"

let operationToString operation =
    match operation with
        | Ops2.AddN -> "+"
        | Ops2.SubN -> "-"
        | Ops2.MulN -> "*"
        | Ops2.DivN -> "/"
        | Ops2.ModN -> "%"
        | Ops2.LeqN -> "<="
        | Ops2.LesN -> "<"
        | Ops2.EqqN -> "=="
        | Ops2.NeqN -> "!="
        | Ops2.AddZ -> "+"
        | Ops2.SubZ -> "-"
        | Ops2.MulZ -> "*"
        | Ops2.DivZ -> "/"
        | Ops2.ModZ -> "%"
        | Ops2.LeqZ -> "<="
        | Ops2.LesZ -> "<"
        | Ops2.EqqZ -> "=="
        | Ops2.NeqZ -> "!="
        | Ops2.AndB -> "&"
        | Ops2.EqqB -> "=="
        | Ops2.OrB  -> "||"
        | Ops2.NeqB -> "!="

for entry in dataflow do
    printfn "%d: " entry.Key
    printfn "begin"
    let basicBlock = entry.Value
    for cmdEntry in basicBlock.cmdProg do
        match cmdEntry.Value with
        | CmdType.CmdAssert var ->
           printfn "%s != 0;" var
        | CmdType.CmdCopy (var1, var2) ->
            printfn "%s = %s;" var1 var2

        | CmdType.CmdAccessArr (resultreg,
           arrayreg, indexreg) ->
```

```
                printfn "%s = %s[%s];" resultreg
                    arrayreg indexreg
            | CmdType.CmdAssignArr (arrayreg,
                indexreg, valuereg) ->
                printfn "%s[%s] = %s;" arrayreg
                    indexreg valuereg
            | CmdType.CmdAssignCnd (y,c,x1,x0)->
                printfn "%s = %s ? %s : %s;" y c x1
                    x0
            | CmdType.CmdGoto gotoi ->
                printfn "ip = %A;" gotoi
            | CmdType.CmdIfGoto (bvar, gotoi) ->
                let nextBBs = Set.toList basicBlock
                    .nextBBs
                if List.length nextBBs <> 2 then
                    failwith "unexpected"
                let altinstruction =
                    if nextBBs.[0] = gotoi then
                        nextBBs.[1]
                    elif nextBBs.[1] = gotoi then
                        nextBBs.[0]
                    else
                        failwith "following
                            basicblock not present"
                printfn "ip = %s ? %A : %A;" bvar
                    gotoi altinstruction
            | CmdType.CmdSync ->
                printfn "ip = %d;" doneIp
            | CmdType.CmdSglAssign (result,
                operand1, operation, operand2) ->
                let operation =
                    operationToString operation
                printfn "%s = %s %s %s;" result
                    operand1 operation operand2
            | CmdType.CmdDblAssign (result1,
                result2, operand1, operation,
                operand2) ->
                let operation =
                    operationToString operation
                printfn "{\%s, \%s} = \%s \%s \%s;"
                    result1 result2 operand1
                    operation operand2
            | _ -> failwith "todo"
        printfn "end"
    printfn "\%d: begin end" doneIp
```

```
    printfn "endcase"
    printfn "end"
    printfn "endmodule"

    0
```