TU WIEN Informatics

go2async: A High-Level Synthesis Tool for Asynchronous Circuits Based on Click-Elements

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Sebastian Michael Wiedemann, BSc
Matrikelnummer 01425647

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dipl.Ing. Dipl.Ing. Dr.techn. Jürgen Maier, BSc
Univ.Ass. Dipl.-Ing. Dr.techn. Florian Ferdinand Huemer, BSc

Wien, 26. September 2023

Sebastian Michael Wiedemann | Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

go2async: A High-Level Synthesis Tool for Asynchronous Circuits Based on Click-Elements

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Sebastian Michael Wiedemann, BSc
Registration Number 01425647

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Dipl.Ing. Dipl.Ing. Dr.techn. Jürgen Maier, BSc
Univ.Ass. Dipl.-Ing. Dr.techn. Florian Ferdinand Huemer, BSc

Vienna, 26th September, 2023

Sebastian Michael Wiedemann | Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Sebastian Michael Wiedemann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. September 2023

Sebastian Michael Wiedemann

Danksagung

An diesem Punkt bedanke mich herzlichst bei meinen Betreuern Andreas Steininger, Florian Huemer und Jürgen Maier, die mich während der Erstellung meiner Diplomarbeit begleitet haben. Mit regelmäßiger konstruktiver Kritik haben sie mich großartig und essenziell unterstützt. Natürlich dient mein Dank auch meinen Studienkolleg:innen, mit denen ich über viele Studienjahre einen prägenden Teil meines Lebens verbringen durfte. Außerdem möchte ich tiefe Dankbarkeit meinen Eltern aussprechen, die mir mein Studium ermöglicht haben, sowie meinen Geschwistern und Verwandten, die mich stets motivierten. Sie, mein innerer Freundeskreis und meine Partnerin verdienen eine besondere Erwähnung: Danke für das rege Feedback, die Übersetzungs- und Formulierungshilfen sowohl für die Feiern und die schöne Zeit.

Acknowledgements

At this point, I would like to express my sincere thanks to my supervisors Andreas Steininger, Florian Huemer and Jürgen Maier, who accompanied me during the creation of my thesis. With regular constructive criticism they have given me great and essential support. Of course, I would also like to thank my fellow students, with whom I was able to spend an important part of my life over many years of study. I would also like to express my deep gratitude to my parents, who made my studies possible, as well as to my siblings and relatives, who have always motivated me. They, my inner circle of friends and my significant other deserve a special mention: thank you for the lively feedback, the translation and formulation help as well as for the parties and the good times.

Kurzfassung

Wir haben bereits die Sub-5nm-Technologie für Halbleiterschaltungen erreicht und scheinen die physikalischen Grenzen der kleinstmöglichen Transistorgrößen zu erreichen. Um die Leistung von Computersystemen weiter zu steigern, kann versucht werden, spezielle Hardware für bestimmte Softwareaufgaben zu entwickeln, um diese zu beschleunigen. Dies ist eine äußerst riskante Aufgabe, da der Entwurfsprozess sehr kostspielig und zeitaufwändig ist. Der Einsatz von FPGAs und die Verwendung formaler Beschreibungen in Form von HDLs erleichtern diesen Prozess. Eine zusätzliche Abstraktionsebene und HLS-Tools ermöglichen es sogar, dass Nicht-Hardware-Spezialist:innen Schaltungen zur Beschleunigung von Computeraufgaben entwickeln können.

Grundsätzlich gibt es zwei Arten von digitalen Schaltungen: synchrone und asynchrone. Synchrone Schaltungen stellen die alltägliche Hardware dar. Ein globaler Taktgeber steuert eine Schaltung und bestimmt die Arbeitsgeschwindigkeit. Asynchrone Schaltungen haben vielversprechende Vorteile im Vergleich zu ihrem getakteten Gegenstück in Bezug auf Leistungseffizienz und physikalische Anpassungsfähigkeit. Der Entwurf von Schaltungen ohne Taktgeber ist jedoch weitaus schwieriger und wird oft als nicht machbar angesehen, sofern keine technischen Hilfsmittel zur Verfügung stehen.

Um dieses Problem in Angriff zu nehmen, wird in dieser Masterarbeit ein HLS-Tool für asynchrone Schaltungen entwickelt: go2async. Go2async ist ein HLS-Tool, welches eine Teilmenge der bekannten Programmiersprache Go parst. Die Hardware basiert auf dem Prinzip der Syntaxbaum-gerichteten Übersetzung. Die erzeugte Hardware ahmt die Funktionalität der Eingabesoftware nach und basiert auf vorverifizierten Click-Element-Strukturen, die speziell mit Blick auf FPGAs entwickelt wurden. Go2async ermöglicht es Go-Softwareentwickler:innen asynchrone Hardware zu erstellen, ohne viel Wissen über die Entwicklung asynchroner Hardware zu benötigen.

Das vorgeschlagene HLS-Tool generiert VHDL-Code, das es ermöglicht, die erzeugten asynchronen Schaltungen mit typischen Simulationsprogrammen zu simulieren. Diese Masterarbeit verifiziert die generierten asynchronen Schaltungen mit Hilfe von ausgewählten Eingabe-Funktionen und zeigt, dass go2async in der Lage ist, automatisch und erfolgreich asynchrone Schaltungen auf Basis von Click-Element-Strukturen zu generieren. Zusätzlich, ist es für gängige Synthesewerkzeuge möglich, laufbare Hardware für FPGAs zu erstellen, wodurch die Nutzung der von go2async generierten asynchronen Hardware in einem realen Szenario ermöglicht wird.

Abstract

We have already reached the sub 5nm technology for semiconductor circuits and seem to reach physical limits on smallest possible transistor sizes. To further increase computer system performance, we can try to create dedicated hardware for specific software tasks to speed them up. This is an enormously risky task since the design process is very costly and time-sensitive. The usage of FPGAs eases this problem by using HDLs to formally describe hardware and allowing virtual prototyping. By using an additional abstraction layer and HLS tools it is possible to enable non hardware-specialists to create hardware to speed up computer tasks.

There are basically two types of digital circuits: synchronous and asynchronous. Synchronous circuits represent the everyday hardware. A global clock governs a circuit and determines the operating speed. Asynchronous circuits have promising benefits in comparison to its clocked counterpart in regards to power efficiency and physical adaptability. However, the design of clock-less circuits is way harder and often considered unfeasible without tool support.

To tackle this problem, this thesis proposes a HLS tool for asynchronous circuits: go2async. Go2async is an HLS tool that parses a subset of the well-known programming language Go. The hardware creation is based on a syntax tree directed translation principle. The created hardware mimics the functionality of the input software and is based on preverified click-element structures which were specifically designed with FPGAs in mind.

Go2async enables Go software developers to create asynchronous hardware without needing much knowledge about the asynchronous design process. The proposed HLS tool generates VHDL code which makes it possible to simulate generated asynchronous circuits with typical simulation programs. The thesis verifies generated asynchronous circuits with the help of selected input functions and shows that go2async is able to automatically and successfully generate asynchronous circuits based on click-element structures. Additionally, it is possible for common synthesis tools to create downloadable hardware for FPGAs thus enabling usage of go2async's generated asynchronous hardware in a real world scenario.

Contents

CHAPTER 1

Introduction

The never ending urge for increased circuit performance while maintaining power, resource, and cost efficiency will remain a challenge for the hardware industry [1, 2]. When looking at the specifications of a computer they usually include data about its processing core: the Central Processing Unit (CPU). Usually, there is a clock frequency associated with the CPU that essentially determines the speed of the whole system (e.g. [3]).

However, questions whether a circuit really needs a clock and whether the clock signal is necessary quickly arise. In fact, clock-less circuits, formally known as asynchronous circuits, do exist. Generally, the clock frequency of synchronous circuits governs the maximum time a signal can traverse through computation logic between flip-flops. The asynchronous counterpart is able to operate without the toggling clock signal by employing certain communication protocols between operation components (e.g. handshaking) [4].

Typical hardware structures contain some sort of pipeline. Instead of hardware solving a problem in its entirety at once, so-called pipeline stages allow hardware to perform calculations step-by-step. In synchronous circuits, a pipeline stage typically consists of logic between two flip-flops. The slowest pipeline stage determines the speed of the whole circuit. Usually, pipeline stages can be executed in parallel. A classic example of this structure can be seen in CPUs [5]. A high-performance CPU is able to simultaneously fetch a new instruction while the next pipeline stage can decode the previously fetched instruction.

The asynchronous sibling relies on specific hardware structures to employ similar pipelining capabilities which operate on a specific communication protocol. The click-element design template is an example implementation style for data-driven circuits [6] which are able to deploy pipelined asynchronous circuits operating step-by-step .

1.1 Motivation

Synchronous circuits are basically everywhere with new hardware based on this communication style being released frequently. At a first glance, their counterpart, asynchronous circuits, are basically unheard of in the mainstream and seems like a research-only field with numerous advantages that come with a great price. However, on a closer look asynchronous circuits were always vaguely in the back of the head of the industry. Power efficiency, neuromorphic computing, and generally event-driven circuits given by the rising neural network research field gave this type of circuit another attention spike (e.g. IBM's TrueNorth [7] and Intel's Loihi [8]).

The work of [9, 10] shows that asynchronous hardware can be constructed by using standard library components. This is ideal for Field Programmable Gate Array (FPGA) implementations and thus enables simple testing of asynchronous circuits on broadly available FPGA development boards. However, generating asynchronous circuits based on click-elements is still a very cumbersome task to do manually. This is where the idea of a High-Level Synthesis (HLS) tool for this type of circuit came into fruition.

1.2 Problem Description

Creating hardware to speed up certain software tasks is accompanied with a huge overhead, especially in verification. The design flow from the specification of hardware to an Application Specific Integrated Circuit (ASIC) is risky in regards to design, time to market, and market adoption. In general, ASIC designs are very cumbersome and cost extensive [11]. This makes custom hardware only advantageous if large quantities are needed.

A solution to this problem is using Hardware Description Languages (HDLs) (e.g. Verilog or Very High Speed Integrated Circuit Hardware Description Language (also VHDSIC) (VHDL)) to formally describe hardware. This enables the creation of virtual prototypes which can be simulated and tested virtually. If simulation results are satisfactory, it is possible to load the tested hardware into an FPGA such that a circuit can be tested physically. Besides creating a more convenient dynamic workflow this approach has numerous cost saving aspects in comparison to creating and testing circuits with many iteration cycles of ASICs.

However, the HDL approach still comes with a very complex design effort as well as an extensive verification process and requires specially trained personnel. This problem is hugely magnified when dealing with asynchronous circuits because the management of every single communication channel between components is required. This makes the manual design of large asynchronous circuits completely unfeasible without tool support.

1.3 Research Questions & Expected Results

The overall goal of this thesis is to reduce the complexity of hardware generation. In that regard this project focuses on asynchronous circuits specifically. The main objective is to create a HLS tool that allows untrained personnel to quickly create asynchronous circuits without needing to know much about asynchronous circuits themselves.

To be more precise, the result of this theses shall be the HLS tool called *go2async* written in the Go programming language [12] which parses a subset of Go itself with the help of the built-in Go parser and Go Abstract Syntax Tree (AST). The supported subset of Go consists of multiple sequentially occurring statements including common program flow structures such as if-statements and for-loops as well as nested code scopes (block statements). Other supported programming statements include arithmetic and binary expressions as well as function calls. To ensure a degree of simplicity only binary and integer variable types are allowed. Fixed size arrays can be used to ensure the usefulness of the high-level synthesis tool.

The program takes Go functions as inputs and generates synthesizeable VHDL code based on click-elements [9]. The resulting circuit functionally behaves the same as the input functions. The function parameters are the inputs of the resulting components and the function return variables are the circuit's output. The previously mentioned function call feature additionally enables interoperation with external circuits. The workflow concept is illustrated in Fig. 1.1.

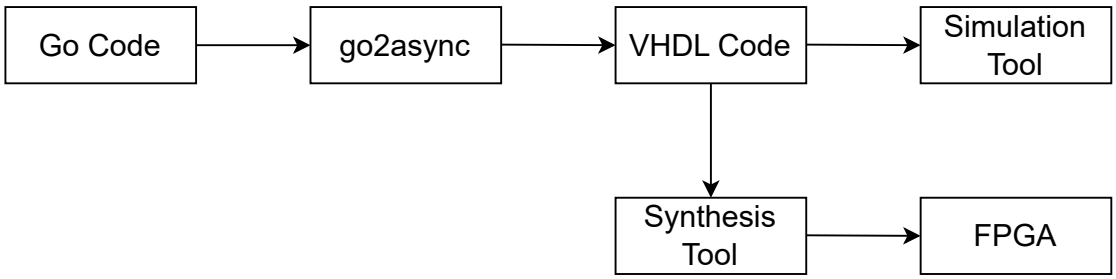

Figure 1.1: Workflow concept of go2async.

To achieve its goal, the thesis has to address following research questions:

- Is it possible to synthesize click-based asynchronous hardware from a high-level design description in Go?
- How must the used capabilities of Go be restricted to keep the code synthesizable?
- Which measures can be taken to synthesize efficient asynchronous hardware?

1.4 Methodological Approach

The methodological approach of this thesis consists of the following steps:

1. *Problem description and extraction of challenges*:

   The first phase of this thesis involves the problem description. From there various challenges are directly implied which will be tackled in the subsequent steps and later chapters.

2. *Literature study*:

   In the state of the art phase a review and analysis of related works and relevant literature will be done to build a knowledge base and provide background context for the work done in this thesis.

3. *Exploration of existing options*:

   Here the results of the literature study will be discussed. In this step the thesis will construct some design ideas.

4. *Implementation*:

   The largest chapter will cover the implementation of the hardware generator. The high-level synthesis tool is actually implemented in this step. The implementation decisions, explanations, discussions and justifications are covered.

5. *Assessment of the implemented concept and exploration of limits by simulation and analysis*:

   In the simulation phase a specific state of the developed tool is analyzed. This will be mostly done by applying certain testcases and monitoring testbenches to verify generated hardware. Additionally, tool outputs will be synthesized on FPGAs and thus even tested physically and practically to ensure usefulness.

6. *Optimization and re-evaluation*:

   The last step of this thesis covers the optimization of the tool. Here the results from the previous phase are evaluated. Depending on the requirement satisfaction status the resulting feedback is collectively applied back to phase 4 restarting an implementation cycle. This can either directly result in the optimization of previously programmed concepts or spring a new feature idea.

The plan is to start with a lightweight version of the tool, which can be improved one step at a time. Especially points 4 and onward can occur in repeating iterations with feedback-loops. This results in a more structured approach and thus yields easier verification.

1.5 Structure of this Work

Chapter 1 presented an introduction for the work of this thesis, containing the motivation, problem description, expected results, as well as a teaser of the methodology which is more thoroughly discussed in Chapter 2. The literature review is covered in Chapter 3

which covers circuit types in more detail, related works, and useful background knowledge. Chapter 4 extracts challenges to be solved as well as drafts solution concepts for identified problems.

The practical part of the thesis starts with the implementation and inner workings of go2async explained in detail in Chapter 5. The thesis continues by showing simulations and verifies go2async's generated hardware by looking at a few examples before showcasing physically used circuits on an FPGA. The chapter ends by naming a few hardware and input software limits go2async has to deal with. The practical part of this thesis ends with Chapter 7 by covering different major versions of the asynchronous HLS tool implemented in this project starting with the initial solution which served as a proof of concept and baseline. Subsequent versions, new features, and ideas are covered here until arriving at the final version of this thesis' project. The thesis ends by summarizing the completed project and stating an outlook as well as ideas for future work in Chapter 8.

CHAPTER 2

Methodology

The applied methodology of this thesis can be seen in Fig. 2.1.

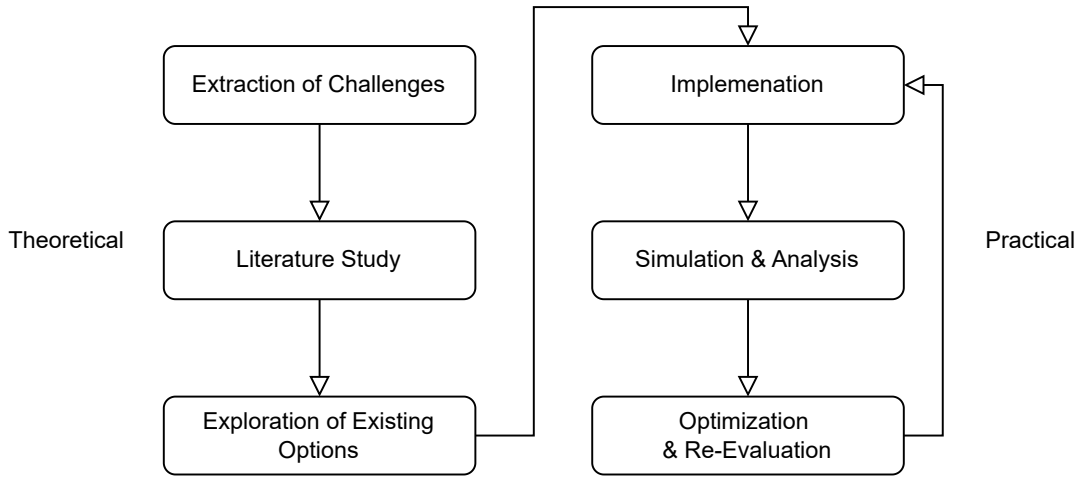

Figure 2.1: Methodology of this thesis.

It consists of the following six phases:

1. *Problem description and extraction of challenges*:

   In the beginning phase of this thesis, the problems are described and the main challenges are extracted from there. Challenges will mainly be directly implied from the defined problems in the field. These can have a wide variety of complexity. One or more solution approaches will be formed for each challenge which will be handled in the later stages of this thesis.

Generally, the tackling of these challenges will be preceded by discussions. The purpose of the discussions is to find alternatives, analyze trade-offs and find the best path leading to the desired solutions. The main goal is to be able to justify the final approaches.

2. *Literature study*:

   This part of this thesis will explore the current state of the art. The corresponding chapter will establish a knowledge base and gather the needed technical background to better understand the subject related to this work.

   More precisely, the literature review will first introduce a discussion about synchronous vs. asynchronous circuits. Especially the drawbacks and known pitfalls of the well known synchronous space will be laid open, as well as how asynchronous circuits can solve various problems while introducing completely different complications and why it might be useful to study asynchronous circuits more thoroughly.

   The field of related high-level synthesis tools for both circuit types is explored mentioning the benefits of using such programs. The general purpose behind such tools is explained. A few practical examples and their usages in the industry will be shown.

   The title of this thesis reveals a focus on asynchronous circuits which of course implies that relevant and important circuit structures of this field are also discussed, namely asynchronous pipelines.

   Related projects working on click-elements [9] will be reviewed which might be useful for implementation approach decisions later on and might even be relevant for comparisons or to draw parallels.

   Since at the core of this thesis lies a software project the remaining puzzle piece before the development can start is finding a suitable programming language. Thus this chapter concludes by reviewing possible programming language candidates.

3. *Exploration of existing options*:

   The findings of the previous phase will be discussed in this step. The assessment of the literature will not be its own chapter in this thesis. It will take place in a paragraph after findings and at the end of the relevant sections and subsections.

   First design decisions and project ideas will be formed here. This includes the reasoning on click-elements as choice in the field of asynchronous pipelines and the two implementation concepts namely sequential vs. dataflow operation processing.

   The discussion about the chosen programming language for this software project takes place in this chapter. Especially why the final decision was made in favor of Go [12].

   This part of the thesis concludes with the analysis of the Go AST and the click-element structures from the click-library [10]. In particular certain patterns and possible parallels are searched which is required for the implementation part.

4. *Implementation*:

   The implementation will be split into two different branches based on operation processing. The first and simpler branch involves the sequential execution of the instructions as defined in order by the code. This branch will be called *sequential branch*.

   The second significant branch of this project involves dataflow processing. Here instructions and data-independent code-blocks are allowed to be executed in parallel. This branch will be called *dataflow branch*.

   The project starts with the implementation of the *sequential branch*. Mainly because this will be much easier and straight forward since this is how single-threaded code is executed and the reference work done on click-elements [10] is based on this instruction processing style.

   In the best case scenario the processing of the input code should work the same in both branches. The big difference will be the internal processing of data dependencies and component wirings.

   Initially, the plan is to start small and gradually make the generator better and more powerful. After the study of the Go AST and found click-element patterns the choice of data structures and handling of component handshakes as well as data-connections will be crucial. The plan is to have easily expandable code in regards to the supported Go input code. Newly supported inputs should mainly not affect previously supported implementation structures. This should make the project seem brick-like where newly added features are independent bricks just added to the program whenever possible.

   This workflow implies that at the start, only a small subset of Go is supported as input. After feedback from the simulation and re-evaluation phases is gathered the supported subset can grow and the program can become more powerful. Alternatively the current implementation can be optimized first (e.g. in regards of hardware usage).

5. *Assessment of implemented concept and exploration of limits by simulation and analysis*:

   After reaching an implementation goal in the previous phase this step will analyze the current implementation. The goal is to verify the functionality of the current state of the tool and explore the limits of the program.

   Firstly, feature exhausting input Go code (i.e inputs that will cover all features or every newly implemented features of the current solution) is used to generate example VHDL code. Afterwards the VHDL output will be put to test in a testbench using ModelSim. Simulations allow to first look at the generated hardware as a black box and whenever there are problems ModelSim makes it possible to look deep into the system by observing single signals at certain timestamps. This process is used to ensure correct and desired functionality of generated hardware. Note

that the generated output makes it possible to even analyze some timing violations by including non-synthesizable VHDL wait statements on relevant assignment operations.

After satisfactory validation of the tool some outputs will be tested on actual hardware. In this case, the used synthesis tool will be Quartus [13] and the target development board will be a Terasic DE0-CV [14] board. This will be done to physically verify the results of the simulation and ensure the usefulness of the developed high-level synthesis tool go2async.

6. *Optimization and Re-Evaluation*:

   In a last step before potentially circling back to the implementation phase, the current state of the tool will be re-evaluated. Solution concepts for found problems and limits of the tool are formed.

   Based on results and findings of the simulation phase possible optimization ideas will arise. This thesis' work will mainly focus on generated hardware optimizations (e.g. hardware exhaustion of target FPGA) rather than optimizing the generation process itself.

   Lastly, whenever problems are fixed and optimizations are completed, depending on the functionality satisfaction of go2async the next implementation step is decided. These are mainly the next to-be supported Go features which will be implemented in the next version of the tool.

CHAPTER 3

Technical Background

This chapter covers the state of the art and related technical topics regarding this thesis. Starting with the dissection of the title of this thesis the *async* part in *go2async* will be tackled first by reviewing problems of synchronous circuits and their asynchronous counterpart. The essence of this project - namely high-level synthesis tools - and similar programs are discussed before diving into the low-level backbone and the inner workings of this thesis' project. This introduces the 'click-element' segment of the title when literature about asynchronous pipelines is reviewed.

Complete the dissection of the thesis project name and Go reveals itself as a chosen programming language where *2* tells us that this certain language is transformed to something *async*(hronous). Lastly, this chapter looks at alternative language candidates for this thesis' project.

3.1 Synchronous Circuits

Synchronous circuits make up the broadly established day-to-day hardware. Almost any purchased hardware has some clock signal in its specification which determines its operation speed. This type of circuit operates on a discrete time variable. Generally, the clock speed is linked to the maximal allowed depth of computation logic between two flip-flops. This is also known as the critical path in a circuit. That is the time a signal is allowed to travel through logic gates between two neighboring flip-flops without violating their setup and hold timings. Synchronous circuits have a centralized clock signal which governs the whole circuit. This is already one of the disadvantages of synchronous circuits as the clock is a single point of failure.

In this type of hardware the top speed is always only dependent on a single critical path even though this path might not even be the most active part of the hardware and the rest of a circuit allows narrower timings. This implies that synchronous circuits rely on

worst-case assumptions for their critical paths to ensure correctness at any time and circumstance. This includes internal and external assumptions. Worst-case assumptions are usually very pessimistic to cover (often rare) edge cases.

Another challenge bigger synchronous circuits face is managing their clock tree [15] especially in modern multiple power designs [16]. Simply using the same clock source in two different signal paths in a circuit means that this clock signal might arrive at slightly different times on two different circuit components. These two signals might merge paths at a later point in time. This time difference is called *clock skew*, which can cause unforeseen timing violations, if not properly considered.

Synchronous circuits generally have to bother with unnecessary high power usage because without special care the whole circuit is supplied with a toggling clock signal making the whole circuit stay active even though there might be only few active relevant computation paths. There are two types of power dissipation: Static and dynamic. Generally, static power dissipation comes from leakage current through transistors whereas dynamic power dissipation comes from switching current. Thus, dynamic power dissipation plays a big part for synchronous circuits. Popular low power design techniques are:

- Clock gating: Ability to turn off the clock for parts of the circuit [17, 18].
- Power gating: Ability to turn off power for parts of the circuit [18].
- Variable voltage: Ability do supply lower voltage for lower performance parts of a circuit [19].

Synchronous circuits' very restricted adaptability to physical properties and technology migration are disadvantageous for them. Whenever parts of a synchronous design change, the whole circuit needs to be re-evaluated. Especially the critical path analysis has to be done after every change. This also implies that it is generally not trivially possible to use old and slower (albeit already verified) parts of hardware in new designs.

3.2 Asynchronous Circuits

Asynchronous circuits are self-timed and operate under a continuous time variable. Instead of pessimistic timing assumptions for calculations between two flip-flops this type of design requires calculation-complete detection to transfer valid data from one node to another. Asynchronous circuits solve all previously mentioned problems of their synchronous counterpart:

The speed is not dependent on some critical path. There are no scalability issues since there is no centralized control unit. The major (dynamic) power usage occurs only in the actively computing path. Asynchronous circuits are more-or-less plug-and-play and thus compatible with similarly behaving hardware by design. However, they have their own disadvantages as they require more delicate care and much more attention to detail

regarding the dynamic state of the circuit to avoid potential hazards. Additionally, each component of asynchronous circuits operates at the same time. The lack of discrete time steps make it impossible to verify every possible behavior.

There are two big asynchronous design principles (and three protocols) which can also be seen in Fig. 3.1:

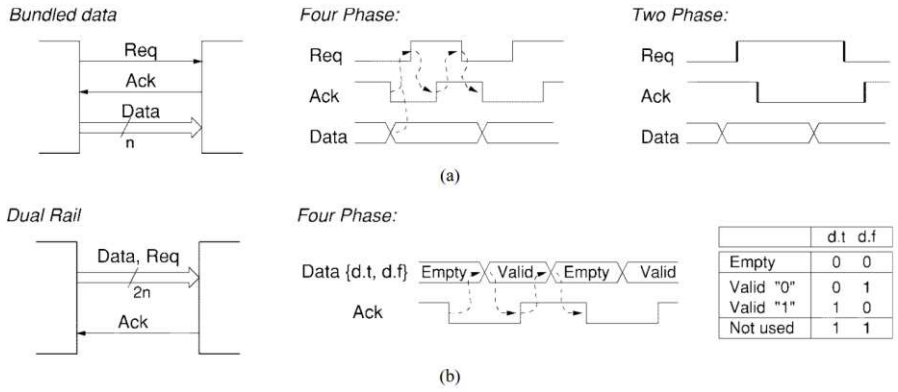

Figure 3.1: The three asynchronous protocols used in practical asynchronous designs: (a) the four-phase and the two-phase bundled data protocols and (b) the four-phase dual-rail protocol. ([4], Fig. 6.)

- **Bundled data/Bounded delay**

  Just like synchronous designs these asynchronous circuits use timing information in some way to detect phase changes. The backbone of this approach is a handshaking protocol between a source of information and an information consuming sink. The goal is to ensure proper coordination between entities.

  Typically, handshakes are initiated by the source to signal the sink that data is stable and ready to be consumed. The sink is able to communicate consumption completeness. The participating signals are usually request and acknowledgement signals with an accompanying data vector. In the general case, the data vector and request signal travel from a source to a sink at the same time. The data however potentially traverses some processing circuit. A delay element has to be inserted into the request signal path which at least matches the worst-case timing behavior of the computation logic to remove a race condition. Recall that the request signal notifies the sink that the data is stable and ready. These delays can be achieved by e.g. inserting inverter chains. The arriving request signal is typically used as a trigger to capture the data vector in a latch or flip-flop on the sink side of the communication channel. After the sink captured the request signal it answers with a corresponding acknowledgement which triggers the next phase on the source side (i.e. ready for more data). These types of circuits need special care during the physical design to ensure timing constraints.

Handshake protocols can either be a four-phase or two-phase protocol. The four-phase (Return-To-Zero (RTZ)) handshake protocol only detects incoming events on certain input state changes (depending on the initial state) and requires handshaking signals to return to an initial state before triggering a new event (see Fig. 3.1 (a) in the middle).

The two-phase (Non-Return-To-Zero (NRTZ)) handshake protocol listens for input state changes only. Thus two-phase handshakes are simply faster but require additional hardware to track the current phase state (see Fig. 3.1 (a) on the right)).

- **(Quasi-)Delay insensitive**

  This design method does not need a sense of time. Instead of using (delayed) handshakes the data vector between a sender and receiver gets encoded which requires additional wires per bit sent. This eliminates the need of a request signal. The simplest case is the 1 bit to 2 wires dual-rail encoding but it is possible to get different levels of complexity and performance by using m-of-n codes [20] (see Fig. 3.1 (b)).

  In principle this method works by introducing valid and invalid (NULL or empty) states for the data vector. The receiving side needs data validity (e.g. no NULL or empty states) or completion detection. This can be done by expecting NULL-states (empty information) on each data-rail after single transmissions. The next set of inputs are only accepted after each rail of an incoming data vector returns to an invalid state before each rail gets to a new valid state. This behavior is similar to the RTZ pattern of four-phase handshaking and cannot be avoided here. While these types of circuits are associated with high robustness they show substantial area overhead [21].

It is obvious that handling communication channels between components individually gets very complex for big circuits. The need of control logic for each calculation to ensure ordered operation is simply solved by placing flip-flops in the synchronous counterpart. This makes hardware design for asynchronous circuits unbearable without tool assistance. The control logic of asynchronous circuits leads to large area usage [4].

Additionally, common Computer Aided Design (CAD) tools and most FPGAs are optimized with synchronized circuits in mind, leaving the development of asynchronous hardware even more disadvantageous [22].

Asynchronous circuits have caught the attention of hardware designers working on event-driven circuits. Neuromorphic computing requires the processing of time and space sparse spikes which make this type of circuit very useful and employ a more natural control flow [23]. IBM's TrueNorth [7] and Intel's Loihi [8] are example projects which show the usefulness of deploying asynchronous circuits to create power-efficient and scalable Spiking Neural Networks (SNNs).

3.3 High-Level Synthesis Tools

The everlasting need for ever more computation power is an endless challenge for the computer industry. An obvious way to tackle this problem on systems that support software execution is to make the heart of the system, the CPU, better and faster. One could also use special acceleration hardware for specific tasks (e.g. Graphics Processing Units (GPUs)) or get rid of the performance loss that software execution systems introduce by designing special circuits for specific applications (also known as ASICs).

While the hardware optimization approach seems appealing, it introduces its own huge overhead, especially in its verification process. Hardware needs to be bug free when it gets to the market, since there usually is no way to fix bugs in the field in an efficient way. Additionally, creating hardware requires highly educated and trained staff. Since hardware solutions are economically risky, hard and very expensive [24], custom hardware designs are only feasible if large quantities are demanded.

An easier and less risky approach to this challenge is using simulation and FPGAs for prototyping. This creates a dynamic workflow by using HDLs (e.g. VHDL, Verilog) which can be used to create simulatable virtual prototypes. After successfully testing a virtual only prototype this approach allows to load the described hardware into an FPGA where it can be physically verified. However, this design flow still requires specially trained personnel and does not get rid of the extensive verification effort.

This is where high-level synthesis tools come into play. These tools are the interface between software and hardware description. They allow untrained personnel to create hardware without knowing anything about hardware design and enable them to quickly achieve satisfying results. In general, an HLS tool takes some standard software code and tries to generate hardware in form of an HDL. The supported inputs are usually subsets of known software programming languages or can be graphical (e.g. flowcharts).

3.3.1 Synchronous High-Level Synthesis

A commercially active HLS tool for synchronous circuits is LegUp [25] (acquired by Microchip[26] also known as SmartHLS) . LegUp's design flow can be seen in Fig. 3.2.

LegUp synthesizes a processor/accelerator hybrid system. This program takes C code inputs and generates Verilog HDL. Without intervention the software runs on a soft-core MIPS processor on an FPGA. LegUp contains a self-profiler which is able to identify parts in the code that could benefit from a hardware implementation. A software programmer can then mark sections in the input code that should be hardware accelerated. LegUp maps C code to an HDL with the help of LLVM Intermediate Representation (IR) [27] instructions. A C statement might generate more than one IR instructions. These can directly be synthesized in an HDL. This allows the tool to create hardware accelerators for specific parts of the input software. The result is hardware on an FPGA where unmarked code will run normally on the given MIPS processor while marked code is hardware accelerated.

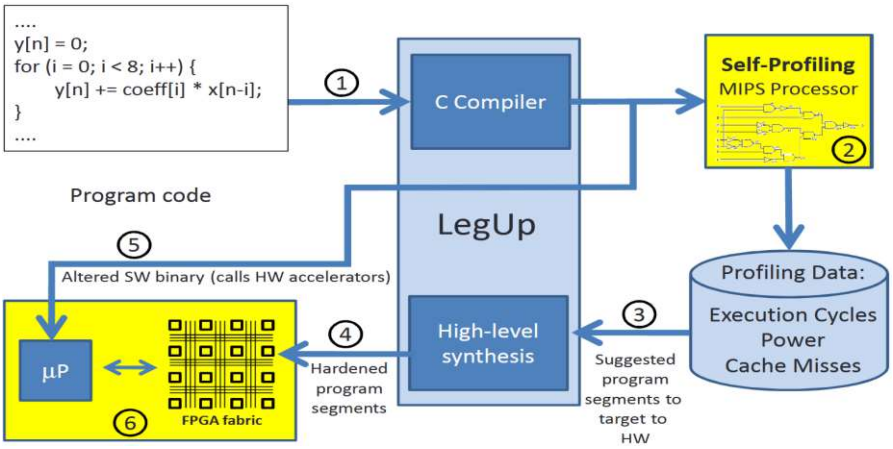

Figure 3.2: Design flow with LegUp. ([25], Fig. 1.)

3.3.2 Asynchronous High-Level Synthesis

In the world of asynchronous circuits many approaches for high-level synthesis, mainly developed by universities, have been proposed in the past. Two big classes of asynchronous HLS are trending. The first being *syntax directed translation* and second being *de-synchronization*. *Syntax directed translation* is the most commonly used. These tools essentially perform a one-to-one mapping of the syntax-tree of the input source into corresponding handshake structure components [28].

A concrete example of a *syntax directed translation* tool is Fluid [29]. This tool basically transforms C input code into an HDL with the help of LLVM IR [27] intructions and Control Flow Graphs (CFGs). The result is a bundled data circuit implementing four-phase handshakes using Muller-C elements [30].

De-synchronization aims as the name implies to convert a synthesized synchronous circuit into an equivalent asynchronous sibling. This is done by employing specific transformation rules for known structures. This method may not enable all advantages of asynchronous circuits but results in a circuit that has the typical asynchronous power-saving properties avoiding unnecessary power loss in inactive calculation paths and makes the circuit less prone to variability [28, 31].

3.4 Asynchronous Pipelines

Pipelining is the essence of high-performance circuits. It vastly improves throughput and parallelism of designed hardware. Synchronous circuits have a rather straight-forward approach to this technique. A simple way is to just split up big computation hardware into so called pipeline-stages which perform simple or parts of complex calculations in each of the created stages. If this process is executed on the critical path it may enable a synchronous circuit to run on a higher clock frequency [32].

For asynchronous circuits this technique is more complicated. A recent investigation of asynchronous pipeline circuits based on bundled-data encoding [23] lists three big 2-phase handshaking protocols: Micropipelines, Mousetrap and Click. Fig. 3.3 shows an abstract view on asynchronous pipelines in linear form (e.g. no forks and joins). Consecutive pipeline stages communicate via handshake channels, usually consisting of Req(uest) and Ack(nowledgement) signals plus a data vector. In each stage a controller handles storage containers and coordinates the handshake signal. The delay in the outgoing Req(uest) signal path has to match the worst-case timing of the optional logic applied on the outgoing data path [23].

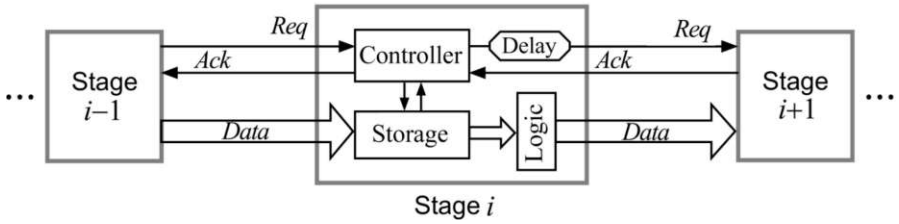

Figure 3.3: An abstract structure of a linear asynchronous pipeline. ([23], Fig. 1)

3.4.1 Micropipelines

One of the first big and classic asynchronous circuits was deployed using Micropipelines [33] which were introduced with Sutherland's Turing Award lecture in 1988. The backbone control of this circuit is the well-known Muller pipeline based on Muller C-elements [30, 9]. Also known as C-gate, it is a latch that has a state holding property. It changes its output if and only if all its inputs reach the same state. This makes C-gates very useful for synchronizing signal transitions in contrast to an AND-gate whose output is HIGH if and only if all its inputs are HIGH and puts its output to LOW if at least one of its inputs is LOW.

The drawback of this design is that C-elements are usually not supported by standard cell libraries and thus need custom implementations with special care. Research on how to implement hazard-free Muller Gates on common Lookup Table (LUT) based FPGAs can be found in [34, 35].

An illustration of the workings of the Mircopipeline with data processing can be found in Fig. 3.4. Each stage has a R(equest) input and an A(cknowledgment) output. Note that ack-signals need to be inverted such that phase changes occur correctly. This gives the circuit an oscillating behavior. The delay has to match the worst-case behavior of the inner-stage control and logic as is usual for bundled-data circuits. The event controlling latch is an unconventional capture-pass register. The input signals C(apture) and P(ass) of the latch have corresponding *done* signals which deliver output events after the latch is done with its action. State changes on C signals causes latches to capture data requiring transitions on P signals to make latches transparent again. [33].

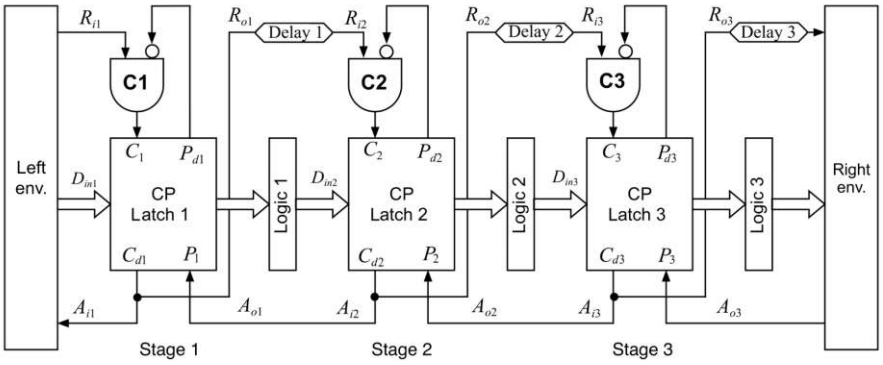

Figure 3.4: 3-stage micropipeline circuit. ([23], Fig. 5)

3.4.2 Mousetrap

Mousetrap [36] (minimal-overhead ultra-high-speed transition-signaling asynchronous pipeline) aims to improve the performance of its predecessor Micropipelines by simplifying the control of handshakes and thus significantly shortening the critical path. Additionally, Mousetrap manages to mainly use standard components (latches and XNORs) which makes this approach very convenient for automated designs or FPGA implementations. However, for non-linear pipeline structures (fork, joins) this pipeline design needs a fall-back to the unfortunate C-elements to synchronize diverged paths back to a single line.

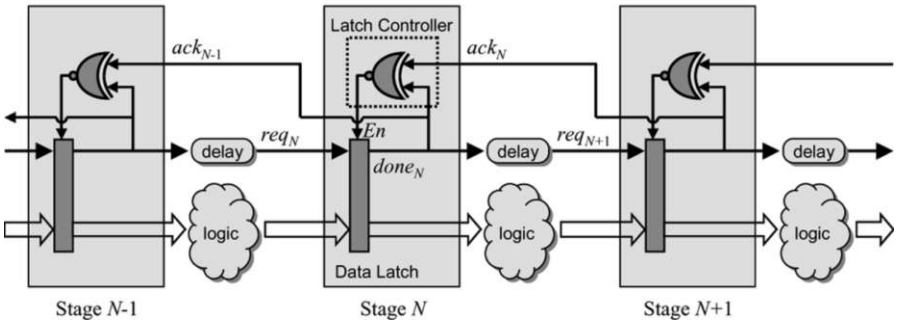

Figure 3.5: MOUSETRAP pipeline with logic processing. ([36], Fig. 4.)

The workings of this design scheme can be seen in Fig. 3.5. As is typical for the bundled data scheme the request signal needs to be delayed such that the data (logic) path successfully arrives at the next stage before the request signal does. XNOR gates are used to create the En(able) signal for the data latches. Even though the event generation behavior is similar to micropipeline's capture-pass register this approach allows the usage of more conventional single-rail blocks in the data-path as is standard for synchronous circuits according to [36].

3.4.3 Click-Elements

The click-elements project's goal was to improve on Mousetrap's shortfalls. In particular they managed to get entirely rid of C-elements and latches that are required in some control and data paths. Instead this design uses edge triggered flip-flops as data storage elements and also for the control circuits. The name *click* derives from the generation of pseudo-clock signals for the used flip-flops as this pipeline style also tries to mimic synchronous circuits as well as possible. In addition, this allows conventional tools to perform optimizations which usually cannot be done on asynchronous circuits. Click-element circuits operate near the speed of Mousetrap circuits [6]. The fact that only standard library components are required makes this type of asynchronous pipeline especially convenient for FPGA implementations.

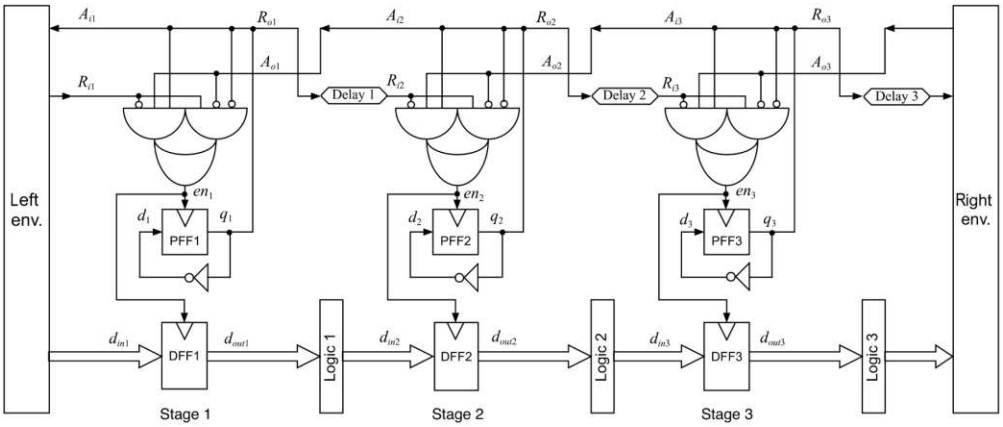

Figure 3.6: A 3-stage pipeline circuit based on the Click template. ([23], Fig. 10)

Fig. 3.6 shows an example circuit based on click-elements. Each stage consists of a controller circuit made out of simple standard library gates only. The control circuit generates a click-signal which is used on the clock-input of the phase (P) and data (D) flip-flops (FF). The changing edge of the control circuit signals a phase change of the handshake state. In the data path, this leads to the capture of the new output of the previous state whereas the state of the phase is simply encoded by inverting the output of the phase flip-flop. The controller essentially implements an arbitrary Boolean function which allows this pipeline scheme to deploy complex pipeline structures. This allows click-elements to mimic the functionality of C-elements if needed [6, 23].

Remarkably, it is possible to make these types of circuits optionally and easily synchronous and scan-testable. This allows for very convenient debugging of single states and stages of this type of asynchronously pipelined circuit which is normally a cumbersome task for this type of hardware. It can be done by OR-type gating a clock signal into the control circuit and adding a multiplexer in the feedback of the phase flip-flop which governs the scan feature. An example can be seen in Fig. 3.7 [6].

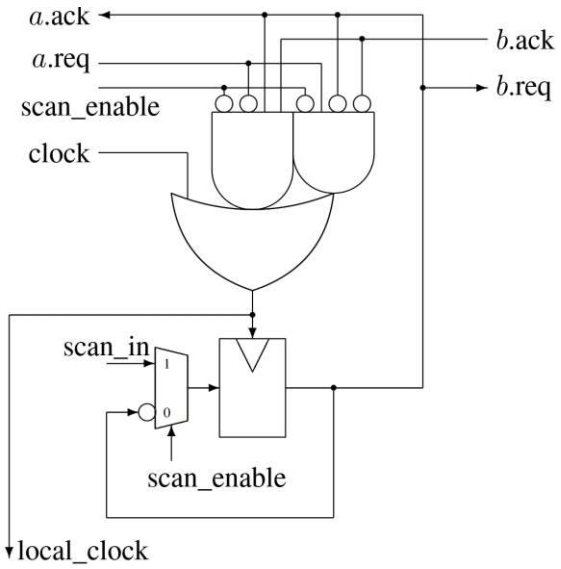

Figure 3.7: Scan-testable control circuit of Click pipeline stage ([6], Figure 18.)

3.5 Work on Click-Elements

Click-elements are the obvious choice for asynchronous HDL implementations on FPGAs. Most importantly, they do not need C-elements and thus are just more convenient to realize. Additionally, they near the performance of Mousetrap [6]. This thesis continues reviewing work done on click-elements.

3.5.1 An Asynchronous Loop Structure Based on Click-Elements

An asynchronous loop structure based on click-elements is presented in [37]. An eight-bit Micro-Control Unit (MCU) was implemented with the aim to verify the loop structures based on click-elements. The designed circuit was simulated and compared against a functionally equivalent synchronous counterpart.

The built MCUs successfully delivered the correct results. On one hand, the performance analysis of the asynchronous version showed a substantially larger use of Logic Element (LE)s than the synchronous counterpart (189%). On the other hand, for the same instructions the asynchronous circuit had 37,5% less working time and it was found that it consumed dramatically less power.

3.5.2 Low Power Asynchronous RISC-V Processor

An FPGA implementation of the Click-based RISC-V Processor is shown in [38]. Li et al. realized an asynchronous RISC-V processor as well as a synchronous one with the same architecture on the same FPGA. By using RISC-V test-tools they verified both implemented CPUs.

To obtain power consumption data, the synthesis tool Vivado is used. As expected for asynchronous circuits, the result shows the asynchronous processor having up to 3x lower dynamic power consumption than synchronous core versions. On average a 200MHz synchronous implementation is 15% faster. However, the adaptive pipeline of the asynchronous design can have the advantage on test cases with few Load/Store operations like in the executed GCD test.

3.5.3 Async Click-library

The contribution of [9] includes the realization of the findings of the click-element proposal in [6] with some additional ideas. The implementation is based on token rings. This work provides FPGA-implementations of click-element pipeline structures in VHDL which can be found in [10]. The implemented structures include the useful components: Register, Join, Fork, Merge, MUX, DEMUX and Function Block. These elements allow to create the functionality any typical software program can do in hardware.

The implemented components have a consistent interface consisting of in/out handshakes plus a data vector, allowing to conveniently link them together in a block-like fashion as needed. The Function Block needs special care since the delay of the outgoing handshake request signal has to match the block's combinational circuit. This is done by following the guidelines of [39] which will be relevant for this thesis' work as well.

So-called peephole optimizations reduce hardware costs by combining certain common handshake components such as Register + Fork which are needed by for-loops [9].

3.6 Conclusion of Pipeline Findings

In addition to click-elements being the obvious choice for this project, the click structure implementions of [10] are already functionally verified and successfully tested in simulations as well as on actual FPGA boards by the authors. Thus, this pipeline structure is even more useful and convenient for this thesis. Consequently this GitHub repository's work will be at the core of go2async. The example circuits (e.g. Greatest Common Divisor (GCD)) presented in [9] whose implementations can also be found in [10] will act as references for the results of this thesis' project. First glances at the async-click library allow this project to form ideas for the automatic generation of such structures.

Fig. 3.8 is a colored version of the schematic of the GCD circuit presented in [9]. The GCD of two input numbers (A, B) is calculated by alternatively subtracting the smaller variable from the bigger one. The loop (and algorithm) ends if the two variables are equal. In this case, either value of both variables can be used as result of the GCD calculation. Conveniently, this circuit contains most important programming structures.

The colored outlines are common program structures. In a) a for-loop is marked. In [9, 10] a for-loop basically consists of a path controlling part (red) and a body (green). The condition component is highlighted in purple. In b) an if-statement is marked. It also

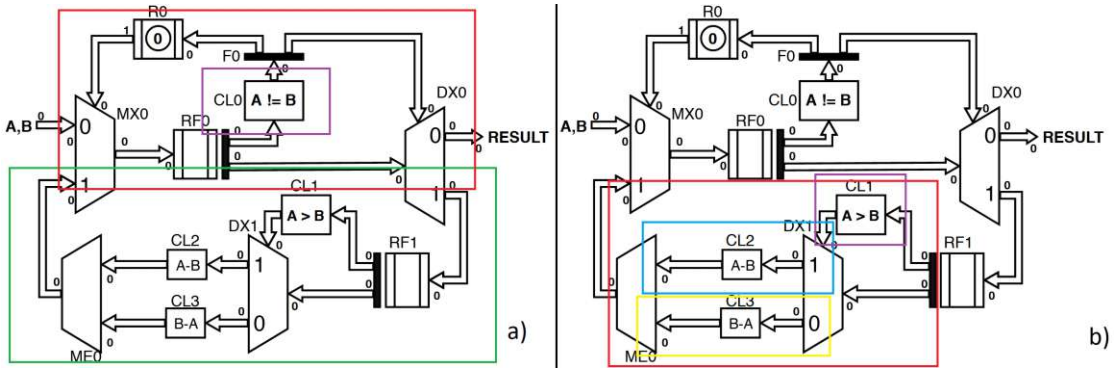

Figure 3.8: Schematic of the GCD circuit. a) marks loop-flow-control (red), condition (purple) and body of the for-loop. b) marks condition (purple) and if-flow-control (red) of an if-statement as well as then (blue) and else (yellow) paths. (Adapted from[9], Fig. 14.)

consists of a path controlling part marked in red. The condition components (purple) are highlighted as well. The bodies are marked in yellow (then-path) and blue (else-path). The GCD circuit from Fig. 3.8 will be more thoroughly discussed in Section 4.2.3.

As is usual for asynchronous circuits it is very cumbersome to manually describe such circuits in an HDL. However, the presence of common structures makes it seem possible to implement a *syntax-directed translation* as mentioned in section 3.3 and thus automate the generation of click-element circuits with the help of e.g. ASTs. This will be more thoroughly analyzed and discussed in Chapter 4.

3.7 Programming Language Candidates

As previously mentioned this project aims to implement an HLS tool that deploys a *syntax-directed translation*. One way to tackle this is by requiring the code parser to abstractly represent the input code syntax. An AST is an example of an abstract syntax representation usual code-parsers are able to generate. This enables mapping of AST-nodes to the desired output format. An ideal language candidate would be a commonly known programming language since the core idea of HLS tools is to allow software-programmers to create hardware. Thus a well-known language is preferred whereas self-made or proprietary input languages are not an option.

Preferably, language candidates have following properties:

1. **There is a parser for the language that is able to create an AST from source code.**

   This is the most important property a *syntax-directed translation* relies on.

2. **The parser of the language is self-hosting.**

That means the parser is written in the same language as it parses. This property is very advantageous because it would make the development more convenient, since there is no switching between two languages for development and hardware generation.

3. **The language is simple.**

   A simple language that is easy to parse and does not allow many alternative variants of valid syntax for the same code is also preferred. This would get rid of the requirement to implement the same output for various inputs that described the same function anyway.

4. **The language is well-known or similar to well-known ones.**

   This enables a lot of potential users and especially addresses the big selling points of HLS tools whose goal it is to enable hardware design without much knowledge about the design process.

With these three requirements in mind the following three programming language are analyzed briefly: Python, Rust and Golang.

3.7.1 Python

Python [40] is an interpreted and dynamically typed programming language which was found in 1991. It is often described as a scripting language. This allows Python programs to skip the compilation step and thus make the edit-test-debug cycle fast. A Python script is simply executed and in case of errors the interpreter raises an exception or prints a stack-trace. Python is also known for its Rapid Application Development (RAD) ability and the short code it requires for tasks in comparison to other languages [41]. According to [42] Python was the #2 used programming language in 2022 on GitHub.

As wished, the Python interpreter can parse Python itself and process Python syntax grammar. However the dynamic typing is rather unpleasant. For instance Python allows to reassign variables with entirely different types later in the code. This property would require additional effort if used in this HLS tool. Additionally, the syntax is relatively complex which is rather disadvantageous with this project's 'start small get bigger' implementation style.

3.7.2 Rust

The Rust programming language [43] was first announced in 2010 by Mozilla. [42] lists Rust as one of the fastest rising languages in recent times. It is most known for its memory and type safety enforcement by employing special rules for dealing with references without requiring a garbage collector. Rust is strongly and statically typed. Thus, types must be known at compilation time and cannot change mid-program. Rust defines built-in types with predefined sizes. This type knowledge is useful for hardware

generation since data-vectors between handshake components have constant sizes once synthesized.

Rust also provides a Rust compiler and an explanation of its AST. However, Rust is a very complex and new language whose concepts are not widely understood. It also allows features from functional programming whose translations to hardware are not feasible for the scope of this work.

3.7.3 Golang

Go is a compiled programming language created by Google with the first stable version in 2012 [12]. The goal of the creators was to create a language that is lightweight, easy to use and easy to parse. The language is statically typed. It also entails predefined types. An analysis of programming languages deemed Go as a potential super-popular language [44]. The compiled code uses a garbage collector.

Like the previous two languages Go ships with a self-hosting parser which is able to generate ASTs. As it turns out Go's clear, simple, and lightweight syntax makes this language a very suitable candidate for this project. The provided AST and parser are easy to work with. Go's simple code structures force specific coding styles which make even subsets of this language seem like ordinary Go code. This leads to few redundant translation mappings other input languages would require. Another nice feature is Go's machine independence. The compiler entirely handles different environments.

Note that the Go language features *goroutines*. This is a way to concurrently run code. Combining this with channels yields event-driven and asynchronous code behavior which could potentially be translated into asynchronous hardware as the channels' send/receive functionalities mimic handshaking behavior.

The simple syntax of Go, while providing rich functionality, makes this language the main choice for this project as long as no dynamically allocated memory is used.

CHAPTER 4

Key Challenges and Solution Concepts

This chapter extracts challenges and uncovers tasks required for the creation of this thesis' high-level synthesis tool go2async. Afterwards, rough solution concepts are drafted for the aforementioned problems. This chapter also explains the project-relevant ins and outs of the Go AST before drawing parallels to the click-library from [10]. The solution concepts will be ultimately tackled in the following implementation section of this thesis (Chapter 5).

4.1 Key Challenges

In general, an HLS tool for asynchronous circuits has to address the following classes of challenges for the design of asynchronous hardware:

1. **Timing and Hazard Analysis**

   The absence of a centralized clock in asynchronous circuits directly introduces additional complexity in timing analysis compared to their synchronous counterpart. It is necessary to eliminate the possibility of hazards such as glitches and race conditions for each computation path.

   Moreover, complex calculation paths (e.g. forks and joins) and the use of unconventional synchronization elements (C-elements) might cause further problems (particularly on FPGAs) depending on the used asynchronous pipeline structures.

2. **Protocol Design**

   Asynchronous circuits rely on handshaking or encoding protocols to ensure the proper order of calculations. The key is defining an efficient and reliable protocol

that can handle complex data dependencies and path synchronization requirements. Furthermore, verifying the correctness of the deployed inter-component communication protocol is an additional huge challenge.

3. **Productivity**

   Designing asynchronous circuits is cumbersome due to the complex analysis task and the lack of standardized design methods. This makes it challenging for hardware designers to develop asynchronous circuits.

4. **Tool Support**

   Synchronous designs have received more attention in terms of tool support compared to their clock-less sibling. The lack of tools and support for asynchronous circuits can make timing analysis and optimization processes more tedious. Therefore, HLS tools for asynchronous circuits need to take extra care for their results to alleviate these challenges.

5. **Hardware Design Automation**

   A system for the automated synthesis of asynchronous circuits must address all the aforementioned challenges. Such a system would significantly improve productivity, simplifying the complexity of asynchronous hardware design and even enabling untrained personnel to develop clock-less circuits. The automation process should be user-friendly to enable widespread usability.

6. **Software Design**

   Lastly, a fresh implementation of an HLS tool is its own complex endeavor. Software projects require a well-defined specification and careful planning. Functional requirements should outline needed functionalities and features the resulting project must possess. Non-functional requirements (such as usability, reliability and scalability) ensure that a satisfying performance is met and allow a convenient way for future and further development. This should serve as the foundation to ensure that expectations are met.

4.2 Solution Concepts

4.2.1 Timing and Hazard Analysis & Protocol Design

As the title of this thesis already mentions, click-elements play a vital role in this thesis' project. This kind of asynchronous pipeline with the help of [6, 9] and the click-elements VHDL implementation of [10] allows go2async to solve the previously mentioned problems 1 & 2.

In more detail, the Click pipeline template defines three constraints that need to be satisfied for the correct operation of such circuits [6]:

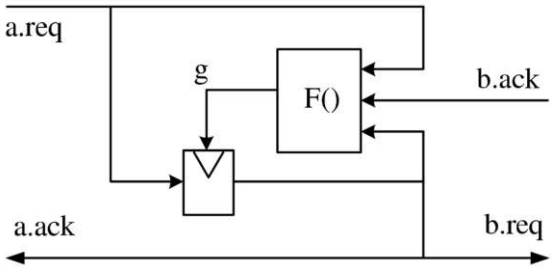

Figure 4.1: Click template two-phase pipeline implementation with feedback-loop based on flip-flop (adapted from [6], Figure 3.)

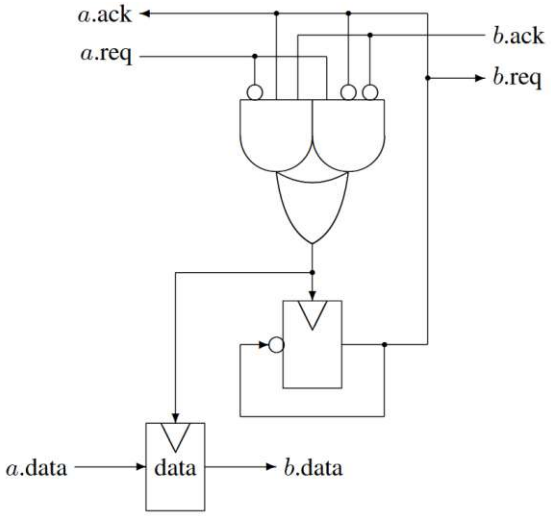

Figure 4.2: Click implementation of simple pipeline stage. ([6], Figure 2.)

1. The click pulse width generated by the control function circuit needs to be larger than the used registers require.
2. Input handshakes need to stay stable during an active clock phase.
3. Hazards such as glitches on handshake signals must be avoided.

The first problem can simply be solved by using common static timing analysis tools to verify whether this condition is met or not since the same verification is needed for synchronous circuits as well.

Fig. 4.1 and Fig. 4.2 show how the remaining constraints are overcome. The output of the flip-flop is fed back into the control circuit, therefore handshake signals are only allowed to toggle after the corresponding output handshake signal is stable. It is claimed that handshake signals are driven by flip-flops, thus these are safe and free from glitches [9].

There are no problems with race conditions because of the construction of the handshake control circuit. It can be verified that potential differences in arrival time of incoming handshake signals do not affect the operation of a circuit using the click template.

The deployed handshaking protocol of the click template is of the two-phase bundled data variant. The behavior of the pipeline design can be described as two-phase token-rings with any number of tokens (often just a single one) where data (a token) is latched from previous pipeline stages and passed to successor stages. Complex pipeline stages are handled by designing a specific control circuit (see *F()* in Fig. 4.1).

4.2.2 Productivity & Tool Support

These two key challenges are at the core of this thesis' project. Productivity gains for asynchronous circuit designs are already vastly enhanced by using the predefined and verified asynchronous pipeline implementations such as the click-library [10]. However, it is still necessary to manually wire all handshakes and to keep track of parallel computation paths. This is obviously a very time consuming and extremely complex process which scales with the size of the target circuit. Maintainability problems will easily arise as an additional challenge.

This thesis' project will not only further improve productivity by completely removing and abstracting the time-consuming process of VHDL implementations, but also take away the need for timing analysis and optimization tools as go2async will deploy its own design methodologies for certain computations.

The use of an easy and widespread user interface for hardware creation is key here. In this case the target group are software developers who aim to speed up processes by utilizing custom hardware. Go2async enables them to do that more conveniently and exploit potential benefits that come with asynchronous hardware. As mentioned in the previous chapter, this project opts to use the rising (C-like) programming language Go and makes use of the light weight parser and its methods to create the target designs.

4.2.3 Hardware Design Automation

The solution concept for this challenge requires researching the available resources, namely the target programming language (Go) and the click-library [10], and looking for patterns. This project aims to develop a syntax-directed translation from Go to VHDL, thus reviewing the Go language and its AST is the next obvious step. Afterwards, repeating patterns in the click-library ([10]) will be searched for.

The Initially Supported Go Language Features and AST

This thesis focuses on Go language features that are relevant for the solution concept and will be supported by the resulting HLS tool. In principle, a valid Go program only requires some package name and a function (main function as an entry point for executables). Go functions are declared with the keyword *func* followed by the function

name, function parameters in parentheses and the function result type. The result types can be a named list just like the parameters. The function body in curly brackets contains Go statements. The project only supports Go functions that contain exactly one return statement to avoid unnecessary complexity during translation. Every function can be converted to such form anyway. The accepted Go subset should at least include functions, assignments, if-statements, for-loops, and binary expressions.

The initial solution supports at least the integer basic types as well as bool. In particular this includes *byte*, *int*, *int8*, *int16*, *int32*, *int64*, and their unsigned counterpart. In addition, it is able to also work with arrays, so that useful and powerful programs can be written as well.

The initial concept considers simple Go input code only because many Go features (especially syntax sugars) are expected to migrate without big complications into future versions once basic implementations work successfully. For instance, if binary expressions are well translated into hardware by go2async, a sum of three numbers will not be a big problem since this particular example can be written in two binary expressions. Additionally, if-statements and for-loops only consider Boolean binary expressions as condition expressions for further simplicity.

```
1  package goexamples
2
3  func sum(a, b int) int {
4
5      a = a + b
6
7      return a
8  }
```

Listing 4.1: Simple sum function in Go.

Listing 4.1 depicts a short and simple example Go code implementing a sum function for two integers. In Fig. 4.3 a slightly simplified AST of the sum function code can be seen. The figure omits additional contents of the Go AST such as source file information since they are not important in the context of this thesis. In general, a node's name, or in case of an array, the array index is followed by a semicolon. After that, the node type's name is written in angle brackets. The Go package name prefix of the node pointer types (**ast.*) and type hierarchy relations are dropped for readability.

The relevant root node of the shown Go AST in Fig. 4.3 is *Decls*. This node contains an array of all declarations in the Go example which contains exactly one function. The function declaration contains the expected name ("sum"), parameter fields, and results fields. Note that the parameters are defined of type *Ident* (identifier) because the parameters are named, as opposed to the function result type, which is simply an integer (type *int)*. The function declaration also contains a *Body* which is an array of *Stmt* (statements). Similar arrays of statements also occur in trees of loop bodies, if-

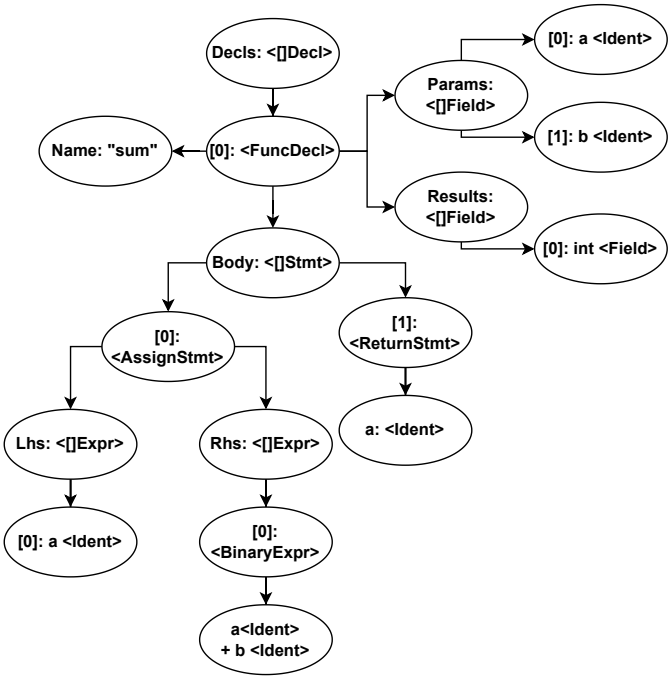

Figure 4.3: Simplified Go AST of the sum function from Listing 4.1

statement bodies (then and else paths) as well as in general code blocks (code specifically bracketed by additional curly brackets). Such commonly occurring AST structures are very convenient for future processing purposes.

The body from Listing 4.1 consists of two statements, namely an assign statement and a return statement (respectively *AssignStmt* and *ReturnStmt* in the tree). As usual an assign statement has a Left-Hand Size (LHS) and a Right-Hand Size (RHS). The RHS resolves as a binary expression (*BinaryExpr*) of the identifiers a and b with the $+$ operation. The return statement simply contains the identifier a.

If-statements and for-loops are represented in the AST as constructs that contain a condition expression that resolves to a Boolean value and a body containing an array of *Stmt* just like *FuncDecl*. As already mentioned, the thesis' initial solution concept focuses on simple binary expressions for the condition expression. For-loop statements with "for clauses" (init statement, exit condition, and post statement) which are usually used to increment a counter variable (often i), as well as if-conditions with assignments are not supported.

The Click-library

The thesis focuses on and analyzes the construction of the click-structures from [9, 10]. For this purpose the GCD example circuit from Fig. 3.8 is used. Note that every arrow

depicts a two-phase handshake plus data from one element to another. The previous chapter already explained parts of this.

Each component has one or more input channels (signals for handshakes and data) and one or more output channels. Every component implements their own inner handshake and data-control (see *F()* in Fig. 4.1). There are many important click-element pipeline structures implemented in [10] (for instance register, fork, MUX, ...). Following relevant big structures consist of and rely on one or more pipeline components:

1. **Statement**

   A statement is simply a function structure that executes some operation(s) on its input(s) and outputs the results. Remarkably, this component has non-constant handshake delays, as opposed to all the other structures, because the handshake management of this component depends on the used operation. For instance an addition requires larger logic and thus more time than an assignment statement. This time-delta has to be determined and dealt with in the handshake paths.

2. **For-Loop**

   The for-loop in the example from Fig. 3.8 consists of an entry-multiplexer (MUX) *MX0*, a for-loop state register (with fork) *RF0*, a loop-condition component (*CL0)*, a condition register *(R0)*, and a loop-body. The body usually starts with a register but the example was optimized so that it was replaced with a Register-Fork combination (*RF1*) (the fork is actually part of the if-statement).

   MX0 (in combination with *R0*) governs whether the loop should get data from outside of the loop structure (e.g. by starting to process a new loop) or accept the input from inside the loop body. Either way, data is passed to *RF0* in which the loop state is stored. The register forks its output to *CL0* and *DX0*. *CL0* calculates the loop condition and forks its result again to *MX0* (over *R0* - this is explained in the next paragraph in more detail) and *DX0*. Depending on the loop condition result the loop either ends and outputs the current loop-state (*R0*) over *DX0*. In this case *MX0* waits for new input from outside the loop. In the event that the loop condition allows the loop to continue, *DX0* outputs the data from *RF0* to the loop body for further processing until it arrives on the "1" input of *MX0* where this triggers a new loop process.

   R0 is special. It is specifically marked with a "0" in a circle at its beginning state (see upper left component in Fig. 3.8) and operates on a different output phase than the rest of the circuit. This is required since a MUX in [10] only accepts inputs if each relevant request signal arrived (condition AND input paths). That means that when initially there is an incoming handshake on the 0-input of *MX0* it is only accepted if and only if the selector data is "0" and the request signal of the handshake channel from the selector path is also already high. In token-ring vocabulary this would translate to: There is a token at R0 at the beginning of the operation and bubbles everywhere else.

```
1  package goexamples
2
3  func GCD(a, b int) int {
4      for a != b /* a) purple */ {
5          // a) green
6          if a > b /* b) purple */ {
7              a = a - b // b) blue
8          } else {
9              b = b - a // b) yellow
10         }
11     }
12     return a
13 }
```

Listing 4.2: Golang implementation of a GCD algorithm.

3. **If-Statement**

   An if-statement in the example from Fig. 3.8 consists of an entry-fork (combined with the register *RF1*), an if-condition component (*CL1*), a path selector DEMUX (*DX1*), two bodies, namely then-path and else-path (arbitrary click-element structures), and an exit-merge (ME0).

   In the beginning, incoming data is forked into the condition component and the DEMUX. The condition component simply decides which path is activated on the DEMUX. The merge component is another special click-element component with two input channels. Instead of relying on a selector like the MUX, it just passes the data of the first handshaking input channel.

4. **Block**

   A block is a collection of one or many sequentially handshaking elements. A block's children can be any of the previously mentioned kinds or another block. Therefore, Click-element circuits consist of recursive block structures. Blocks have exactly one input and one output channel. This component usually occurs as loop and if bodies.

The Go AST and Parallels to the Click-library

Fig. 3.8 shows the schematic of the GCD circuit from [9]. When looking at the Go implementation of the applied GCD algorithm (Listing 4.2), some parallels and patterns can be extracted. The for-loop *a)* contains the loop condition `a!=b` which is marked purple in the previously referenced code and figure. The loop's body is marked green.

Similarly, the if-statement's *b)* condition `a>b` is marked purple as well. The bodies are marked blue and yellow. The whole if-statement is marked red since the whole structure

is the if's flow control (see next paragraph). Even the bodies play a vital role in the flow control of an if-statement because the merge component simply outputs the first incoming channel.

The red parts of Fig. 3.8 do not have an equivalent meaning in Go code. The red structures are just a vital part for the constructions of correctly working if-statements and for-loops in hardware. This does not lead to problems for the software to hardware translation since only the condition and body contents are relevant in this regard.

The previous section already explained the click-element structures in a very code-friendly manner (statement, if, for and block). The parallels to Go code are obvious. The block's recursive behavior is a very useful property. The Go AST, like trees in general, can be recursively described as well. Additionally, click-element blocks also consist of sequentially handshaking elements. This is similar to sequentially occurring statements in Go code. These common features are the keys for the automated generation of of hardware description. The only missing task is the actual code design for the software to hardware mapping process of go2async.

4.2.4 Software Design Principle

The idea of the software concept is to program Go code that parses a subset of Go with the built-in Go parser. This leads to a convenient coding environment with only one programming language to deal with during the software development of go2async and the testing of the HLS tool. Additionally, the machine independence of Go allows migrating the project to other systems and enables go2async to be compiled and executed on common machines without any major hassles.

This thesis project aims to implement two different hardware concepts (both also pitched in [6]). First, a sequential hardware which behaves similarly to a single threaded program in which each component has exactly one predecessor and successor. This approach is similar to the examples of [9, 10]. There is one common data vector for each component which acts like memory. No special processing overhead is needed for this straight-forward approach. This approach is the first and main focus of this thesis.

The second concept implements a dataflow analysis. Depending on the needed input data, each component waits for multiple predecessors to finish their computations. This results in a hardware structure with many join and fork elements for path synchronizations that needs to be dealt with. This approach is much more complex to implement compared to the sequential concept. Data dependencies between all statements need to be resolved. The resulting hardware exploits potential parallelism capabilities of the input code. This design idea is implemented after the sequential approach works sufficiently well.

Data structures that save all the relevant information needed at runtime have to be defined. For this purpose, this project will make good use of Go's interfaces and struct embedding feature called composition. The initial idea for variable handling simply involves a single data vector which acts like Random Access Memory (RAM). Each

component has the same input vector, performs some operation on it and passes it to its successor. It is required to track and map variables to its space in the data vector.

As already mentioned, this thesis makes use of the VHDL click-element structure implementations of [10]. A big challenge of the software project is to extract components from the Go input code, automatically handle dependencies between each component and automatically wire them together as needed.

The concept of go2async's processing consists of parsing Go code (and walking of the Go AST) and mapping it to hardware (described by VHDL). The Go parsing part needs to deal with the recursive structure of the AST, which is done by implementing recursive parsing functions. During this operation, relevant data structures and connections between them are formed. Afterwards, the hardware mapping process checks for potential errors in the built data structures of the parsing part. Lastly, the hardware mapper generates a VHDL file that represents an asynchronous circuit based on click-elements. The resulting hardware has the same functionalities as the input Go code. The implementation is explained in detail in the next chapter.

CHAPTER 5

Implementation

The chapter covers the implementation of the asynchronous HLS tool go2async in depth. Go2async is a program written in the Go programming language. It parses inputs that are a subset of Go and generates asynchronous hardware based on click-elements in the form of VHDL code.

The following sections present software implementation relevant topics of go2async. The chapter starts with general facts around the software environment including the program execution usage, discussion about the supported input Go subset, and illustration of the resulting asynchronous circuit. Afterwards, the inner workings of the program are explained. This entails used data structures, variable handling, connection handling of the generated components, and how the Go input code is parsed. Lastly, it is explicated how the components of the resulting hardware are generated.

5.1 Program Execution

It is suggested to run go2async in a Command Line Interface (CLI) environment. The program usage for standard asynchronous hardware generation is as follows:

go2async generate *GO_FILE* [*OUT_FILE*]		... Minimal syntax with optional output file
	[- - sequential]	... Enable *sequential mode*
	[- - intSize INTEGER]	... Set size of *int* type
	[- - debug]	... Enable debug output
	[- - verbose]	... Enable operational output
	[- - help]	... Print usage

For normal operation, go2async expects the *generate* keyword after the executable name followed by a required Go input file. Optionally, a user can specify an output file to write the generated hardware (VHDL code) to. Without output file specification the program writes everything to *stdout*.

Go2async's VHDL output contains architectures and definitions of every single generated component. With the addition of third-party components consisting of click-element structures from [10] it is possible to synthesize hardware with only two files. However, it might be useful to include another VHDL file containing the usual *top* component wrapper which is common in typical VHDL projects.

Go2async defaults to the dataflow operation mode (see Section 5.6.2). The so-called sequential mode (see Section 5.6.1) can enabled by using the - - *sequential* argument.

Additionally, it is possible to set the size of Go's *int* type in hardware. This is important for enabling smaller integer types which are represented in vectors in VHDL (more details are in section 5.5). For instance a typical seven segment display driver, which is very common on FPGA development boards, requires a 4-bit wide input which has no trivial representational type in Go.

A debug mode can be turned on enabling the program to print runtime state data which aids in development and debugging. The verbose option prints useful information during operation such as function parameter locations in the output's VHDL-vectors.

5.2 The Supported Go Subset

The goal of go2async is to parse a sufficiently large Go subset so that useful programs can be written. A programming language is useful if it can simulate a turing machine, thus can do everything a turing machine [45] can do. This is fairly quick to accomplish by simply supporting conditional branches (such as if-statements and for-loops) and arbitrary memory read/writes (for instance variables and arrays). The second requirement is in theory physically impossible to fulfill because it would require unlimited memory. However, for *turing-completeness* [45] a bounded memory model suffices, which is very convenient for this thesis' project.

With this in mind go2async requires a Go input file with a package name and at least one function. The function should have at least one return variable (Go allows functions to return multiple values). Additionally, the following Go features are supported by go2async:

1. **The supported types are most of the Go primitives with constant size**

   Most important is the constant size property of supported types during runtime of go2async. Hardware stores variables in data vectors which cannot be dynamically sized. In particular the supported basic types are: *bool*, signed and unsigned

integers (*int*/*uint* and their 8, 16, 32, and 64-bit siblings) as well as *byte*. As already mentioned the sizes of *int* and *uint* can be altered via program arguments.

The default size of the *int* (as well as *uint*) type in go2async is 4-bit. The reason is simply a smaller resulting circuit in the common case. The default size can be easily overwritten. This fact needs to be kept in mind when debugging on the software side since 4-bit values overflow very easily.

2. **Fixed-sized arrays of every supported type**

   This feature is very important to enable usefulness of the resulting asynchronous circuits.

3. **Multiple variable declarations at once via *var* keyword without value assignments.**

4. **Single variable declarations via := but only with single value assignments**

   That means one LHS and one RHS variable (or constant) but without binary expressions on the RHS.

5. **Assignment expressions using =**

   One LHS-variable and an expression on the RHS. Nested binary expressions are allowed.

6. **Addition and subtraction operations**

   Only addition (+) and subtraction (-) calculations are allowed. Multiplications (*), divisions (/), and modulo-operations (%) cannot be trivially implemented in hardware, thus are not supported by go2async. If needed, these operations can be subsidized by programming a loop of allowed operations. Note that it is possible to write functions for this task, thus the Go input code does not need to get ridiculously bloated if multiple multiplications or divisions are required.

   Allowed bitwise operators are *or* (|) as well as *and* (&).

7. **Basic comparators**

   These include equals (==), not-equals (!=), smaller (<), smaller-equals (<=), greater (>), and greater-equals (>=).

8. **If-statements with a simple binary expression that results in a boolean value**

   No nested binary expressions in conditions are supported. An empty or omitted *else* code block is allowed. *Else if* blocks are not supported. The alternative are cascading if-statements.

9. **For-loops with a simple binary expression that results in a boolean value**

   No nested binary expressions in conditions are supported.

10. **Exactly one return statement at the end of a function**

11. **Multiple function definitions in a single Go file**

    Each function generates an independent asynchronous circuit. Go2async supports multiple parameters and mulitple result variables. These are encoded in the *in* and *out* data vectors respectively.

12. **Non-recursive function calls**

    Two different types of function calls are implemented in go2async. It is permitted to call another function that is defined in the same Go input file. This is possible because every click-element component has the same interface, implying that even an asynchronous circuit, which was translated from software to hardware via go2async, has the same interface as any other internal statement component. This allows using any circuit resulting from go2async inside click-element components just like any other.

    Additionally, it is possible to provide a function pointer in the function parameter list. The resulting circuit's interface gets extended by an additional typical click-element-like interface. This enables the asynchronous circuit, which results from the given function, to call external interfaces. The external circuits are required to conform with the click-element interface (input/output handshakes and data vectors) whose data vector semantics are defined via the function pointer type.

    Note that the result of a function cannot be used in binary expressions. One needs to store the function result in a variable (see assignment rule). Arrays of function pointers are not supported.

5.3 Abstraction of Resulting Asynchronous Circuits

In principle, asynchronous circuits generated by go2async are composed of following components:

- ***Binary Expression-Component***

  These components represent assignments and expressions of any kind. *Binary Expression-Component*s expect one or more relevant input values. An operation on given values is performed and the result is stored in given memory space (basically the output vector of the component). Nested binary expressions are implemented by chaining binary expression components.

- ***Selector-Component***

  These components are a special case of binary expression components. The result is always a single bit. They are needed for if-statements and for-loops.

- ***Block***

  A general *Block* is a collection of all previously mentioned components as well as other *Block*s. The components inside a *Block* are called children. Concurrently, every component has a parent *Block*. This implies that every *Block* type can be a parent and child at the same time. The result is a recursive tree-like *Block*-structure. A single parent-less root-*Block* exists. Children handshake with each other as is usual with click-element structures. The communication behavior is extracted by the input Go code. The inputs of a *Block* component are all variables read by at least one (grand-)child. *Block* outputs consist of at least every written variable of each child.

- ***If-Block***

  *If-Block*s are a special kind of *Block*s. They represent if-statements in click-element fashion (see previous chapter). All the variables that are read in condition component, the then, and the else path-*Blocks* have to be defined as an input of such a *Block*. Coincidentally, the outputs need to be at least every written variable of the then and else paths combined. Because of internal structuring both paths need to output the same values. This is trivially solved by wiring variables, that are used in one path but not the other, from input to output without any operations.

- ***Loop-Block***

  *Loop-Block*s are special kinds of *Block*s. They represent for-loops in click-element fashion (see previous chapter). The inputs of such components are required to be at least all read variables of the condition-component and body-*Block*. The outputs of for-loop components are required to be at least all variables the body-*Block*s write to.

- ***Scope***

  Scope components represent the Go input function in hardware. In practice they are simply wrapper components for *Block*s combined with a register for some of the *Block* outputs. In particular, the inputs of a *Scope* component are the Go input function's parameters encoded in a single data vector. Similarly, the outputs are the input function's result variables encoded into a single data vector. Usually, a *Block* outputs more variables as needed by a *Scope* component. Therefore, a *Scope* component filters irrelevant outputs and stores relevant *Block* outputs in registers. Thus a *Scope* component can be seen as a wrapper for *Block* components with registers for outputs. Usually, the wrapped *Block* is the parent-less root-*Block*.

- ***Call-Block***

  A *Call-Block*'s purpose is to call another function of the same Go input file. It basically instantiates the *Scope* component of the called function. The biggest challenge is to make sure the data inputs and outputs are wired correctly. The *Call-Block* is essentially a wrapper around a *Scope* component which filters the *Scope*'s I/O.

- ***Function-Block***

  This *Block* is responsible for wiring a black-box that conforms with the click-element structure interface with the variable inputs and outputs defined by the function pointer in the parameter list.

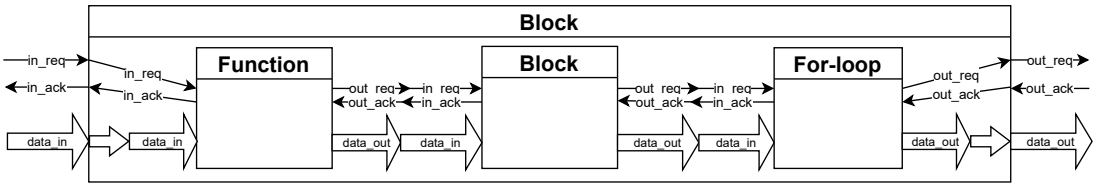

Figure 5.1: An illustration of an example go2async circuit.

An illustration of an example structure of the mentioned components can be seen in Fig. 5.1. The figure contains a root-*Block* with the usual click-element interface (input/output handshakes and data vectors). The insides of the root-*Block* consist of three different *Block* structures including another general *Block*. The insides of the children are not specified in greater detail to illustrate that *Block*s can act as a black-box. Only the consistent interface is important for all used structures in this project.

5.4 Data Structures

This section primarily explains the data structures which describe the resulting hardware of go2async. Typical Object Oriented Programming (OOP) languages have a feature called inheritance. This allows sub-classes to use all methods and fields of its super-class (at least the public parts). Golang does not have classes, thus cannot implement the traditional inheritance functionality per se. However, Go keeps its OOP language status by using a method called composition. Instead of inheriting features from a super-class, Go relies on embedding structs (objects) in one another to allow a struct to gain all the functionalities of its embedded structs. Basically, the embedded struct gets to be a part of the embedding struct and the embedding struct extends the embedded. Similar technique can be applied to interfaces. Embedding interfaces into a struct forces implementing the interface's methods on the struct. This is similar to typical OOP interfaces and also allows structs to be references via multiple (interface) types.

Data strucures are used to represent the resulting asynchronous circuit based on click-element in memory during runtime of go2async. There are four big base struct types defined in the project: *BodyComponent*, *HandshakeChannel*, *VariableInfo*, and *DataChannel*. Additionally, various interface types are defined to enforce specific struct properties. The most important ones are *Component*, *BodyComponentType*, *BlockType*, and *VariableDef*. Based on these structs and interfaces the thesis' software project makes good use of Go's composition, interface, and typing features.

5.4.1 Interfaces

The interfaces are mostly intertwined. *BlockType* embeds *BodyComponentType*. *BodyComponentType* embeds *Component*. The *Component* interface's main job is to enforce low-level properties which are especially needed for the resulting circuit. This entails properties such as a component's name, its architecture name and methods for the actual VHDL representation including VHDL-component, entity, and architecture strings. These interfaces are used to enable using generalized component types in internal processing functions. Otherwise, frequent duplicated code would be required for the various component and block types.

The *BodyComponentType* interface contains all methods needed for variable handling and connection handling (see next sections) as well as various general setters and getters which are required for the internal representation of a component. *BlockType* defines methods needed for the representation for blocks including functions for child/parent relationships. Additionally, blocks also play a part in variable handling and connection handling (especially for its children), thus the interface ensures required functions are implemented in blocks as well.

VariableDef is the main interface for variable representations. In general, variables are encoded in the single data vector between one component to another. This interface enforces variable objects to implement methods needed for valid variable encoding and decoding. Details regarding variable handling are discussed in Section 5.5.

5.4.2 Variable Structs

Variable structs are used to handle variable declarations as well as functionalities for encoding and decoding into and from a data vector. *VariableDecl* is the base variable struct used for a variable's basic definition needs. It contains a variable's name, size (bitwidth), length (in case of arrays), and type.

VariableInfo extends *VariableDecl* to include data for the encoding and decoding process as well as information about its kind of variable. For instance a variable may be a temporary variable, a constant value, a function pointer, or used as an index in an array. If the variable is stored in a data vector, information about its position in the data vector is included.

ScopedVariables is a struct that represents named variables in a data vector as a collection. It includes a map as well as a list of all variables in the data vector for named and positional accesses. This struct also has methods to help in variable handling.

5.4.3 Connection Structs

There are two different connection structs in go2async, namely *DataChannel* and *HandshakeChannel*. Each component is the owner of at least one of those structs to represent incoming and outgoing connections. As the names imply they either represent the data vectors or handshake signals of hardware components.

The *DataChannel* contains direction information (in or out), a reference to its owner, the name of its signal, the variables (in form of *ScopedVariables*) it represents, the references to its targets, and miscellaneous information about its connection state.

The *HandshakeChannel* contains direction information (in or out), a reference to its owner, the references to its targets as well as connection state information.

5.4.4 Component Structs

BodyComponent is the most important data structure of this thesis for the internal representation of a circuit translated from high-level source code. In principle, *BodyComponent* is the base struct of every component and *Block*, which is explained in the previous section. The struct implements the *BodyComponentType* interface's methods and contains all required data to fulfill this task. The member variables entails information for the *Component* interface implementations, connection information arrays, variable informations, and pointers to predecessor as well as successor components. The implemented methods act as default implementation. This is because other structs which embed *BodyComponent* might need to override (it is not exactly OOP overriding but behaves similarly) the defaults. A *BodyComponent* on its own is not sufficient for the representation of a valid click-element structure. This base struct only acts as the basis for all previously mentioned components and *Block*s. It needs to be embedded into another component struct which more accurately defines a click-element structure. The base struct includes a reference to a parent *Block* and the representation of a component's interface. Each component is responsible for its connections to potential predecessors and successors. However, a parent *Block* can help provide vital information for the connection of its children (for more information see Section 5.6).

Simple *BodyComponent*s

The simplest *BodyComponent* embedding objects are all control-flow and register structs. These are most of the components defined in the async-click-library [10]. Important examples are *Fork*, *Join*, *MUX*, *DEMUX*-components as well as the *decoupled handshake register*. The purpose of these implementations is to enable easy coupling with other internal *BodyComponent*s and *Block*s. Their functional potential is exactly what their names imply. For instance a *Fork* component takes one handshake channel and data channel and splits (forks) them into an arbitrary number of output channels (at least one). In principle, since the *BodyComponentType* interface enforces methods for VHDL implementations, the Go implementation of these structures translates the functional needs that were determined during go2async's runtime to parameters that the async-click-library components need to fulfill various requirements.

The *Binary Expression-Component* embeds the base *BodyComponent*. It represents a binary expression in asynchronous click-element hardware. Constructing a *Binary Expression-Component* requires operand and result variable information, the operation to be performed on given operands, and a reference to the parent block (for more details

see Section 5.5). This object's most important ability is to construct the VHDL process which describes the given operation on given operands. Additionally, this object handles different handshake-delays depending on the used operation. The *Selector-Component* is very similar. The main difference are the single bit result value and used operations (arithmetic/boolean vs. comparators).

Blocks

A general *Block* component is basically a *BodyComponent* extended by the *BlockType* interface. The software equivalent is a code block in curly brackets. A *Block* can be described as a black-box of operations with the usual click-element input/output interface. That results in a *Block* struct being able to store and manage its children via a slice of *BodyComponentType*s and contain various maps to store vital variable information for its scope. A *Block*'s responsibility is to manage the resources its children need. In addition a *Block* governs the handshake connections of its children. These obligations lead to a complex VHDL architecture construction task to represent a *Block* component. The architecture has to contain every single child component's definition in addition to all the information required for wiring everything together. This entails signal definitions for every single handshake signal and data vectors. The connection and variable handling process is more accurately described in Sections 5.6 and Section 5.5 respectively. Additionally, *Block*s also gain the ability to handle external interfaces (see Section 5.4.4)

An *If-Block* is a special case of the *Block* component. Firstly, it embeds the basic *Block* component to gain all its methods and variables. The peculiar task of this *Block* is to accommodate and wire the control flow components to realize a valid click-element if-statement. Thus, it contains the children: *Selector-Component*, *DEMUX*, *merge* as well as *then-Block* and *else-Block*. The *If-Block* is non-extendable and the only parent of the aforementioned components. However, it has the usual *Block* responsibilities of its children, especially the variable handling of its two *Block* children. In principle, an *If-Block*'s task is to handle the predefined architecture of an if-statement in hardware. The signal definition and connection handling is easier here because every component is already known at construction of this component.

The *For-Block* is another special *Block* component very similar to the *If-Block*. The *For-Block* also extends the basic *Block* struct. It has a predefined structure and children including a *MUX*, a loop-state register, a *Selector-Component*, a *MUX* register, a *DEMUX*, a body *Block*, a body register, as well as various control flow components (such as *Fork*s). The structure itself is more complex than the *If-Block*. However, this *Block* does not need to handle variables for two different scopes (body *Blocks*) as it only has one loop body. Remarkably, this component contains two differently operating *decoupled handshake register*s. As already mentioned in the previous chapter (as well as in [9]) the *MUX* register needs to operate on a different out-phase than the rest of the for-loop circuit to figuratively *kick-start* this sub-circuit.

A *Call-Block* is a simple extension of the *Block* component. Its only purpose is to instantiate the *Scope* component of another function defined in the same Go input file as well as handle variable and handshake connections to ensure correct functionality.

Lastly, the *Scope* component is a trivial extension of the *Block* struct. The functionality was already described in the previous section. It is a wrapper around a *Block* and a register component.

The Special Function-Block

A *Function-Block* is principally similar to the call-block since it represents a Go function call in hardware. However, instead of instantiating another *Scope* component it instantiates a named component defined by the name of the function parameter of the Go input code method it is used in. It is expected that the VHDL project contains the named component which also conforms with the usual click-element structure interface (handshake and data channels). Go2async cannot verify this external dependency, thus simply generates the previously mentioned component and delegates further responsibility to the user and hardware synthesis tools.

This *Block* has a very complex side-effect. In hardware a simple function call (*Call-Block*) basically needs to initiate a request handshake with the representing component and wait for outgoing handshakes to complete a transaction (as usual for click-element structures). However, a *Function-Block* intends to call an already existing external component. Instead of instantiating the component to call, this component communicates with the external component interface directly. An example visualization of the *Function-Block* object can be seen in Fig. 5.2.

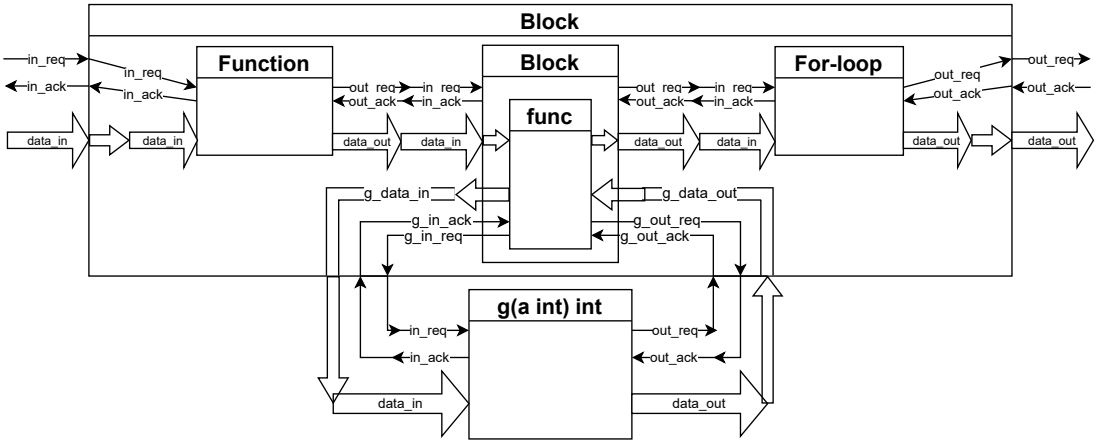

Figure 5.2: An illustration of an example go2async circuit with external function call. (Fig. 5.1 extended)

The illustration extends Fig. 5.1 by including a *Function-Block* inside the inner *Block* of the root-*Block*. The *Function-Block* represents Go code that calls *g(a int) int* (a method

g that takes an integer and returns one). It can be seen that the external component *g(a int) int* conforms with the click-element structure interface. The *Function-Block* requires the usual click-element interface with addition of an inverted interface of the external component. That means that from the perspective of a *Function-Block* the communication signals to the external component are reversed, such that the *Function-Block* can *call* the external component. In this asynchronous hardware-context, *calling* translates to sending a request signal to the called component. Data vectors to and from the external component have to be inverted too. The *Function-Block*'s default I/O are the usual handshake and data channels to communicate with other click-element structures (for instance the parent *Block*).

In the example from Fig. 5.2, the *Function-Block* calls the function-pointer *g(a int) int*. It has to handle both integer inputs as well as outputs and delegate results of the external interface further to its parent *Block*. The illustration also shows that the external interface is passed through the root-*Block* as well as through the inner *Block* before it arrives at the *Function-Block*. Thus, the affected interfaces have to be extended as well.

Type hierarchy illustration

Fig. 5.3 shows the type hierarchies of the relevant data structures used in go2async. Interfaces are illustrated as rounded squares. The figure only shows the type hierarchy. No inner variables and objects are described. As explained in the previous subsections, *BlockType* embeds *BodyComponentType* and is thus drawn inside the embedding interface. Whenever a struct implements a given interface (noted by a colon in the illustration) each child (elements inside a square) also implements the interface (thus the type) by construction.

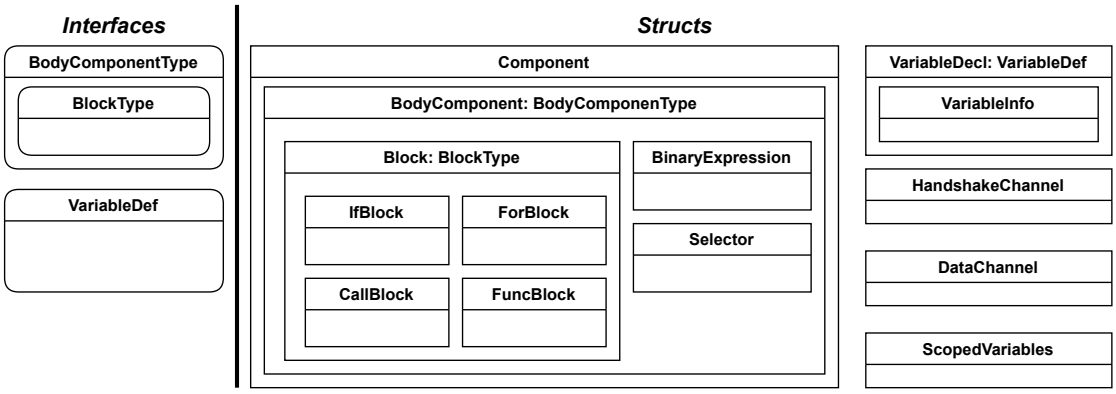

Figure 5.3: Type hierarchy of go2async's interfaces (rounded) and structs (not rounded).

Note that in Go code the definitions of the shown hierarchies are built from the bottom up. An inner drawn square actually embeds the parent square instead of extending it (as is usual for typical OOP languages). However, the illustration would become way bigger and the type hierarchy stays the same anyway.

5.5 Variable Handling

The Go programming language has variable scopes. Scopes are domains in which variables may or may not be valid. These are basically given by curly brackets and as usual for many other programming languages it is possible to have nested scope structures. Each variable is valid after its declarations in the scope in which it was declared and in every nested inner scope thereafter. Variables are not valid outside the scope it was declared in as well as before they were declared.

Go2async's *Block* component's very convenient construction allows to mimic these scopes nearly perfectly. In further context, a *Block* component is deemed equivalent to Go blocks. Thus, *Block*s play the initial vital part in go2async's variable handling.

In principle, variables are encoded in the input and output data vectors of various *BodyComponent*s. A data vector contains a collection of many variables. The usual approach in a go2async circuit for this collection is to pass through various components (operations) in which parts of the collection (one or more variables) get altered. The collection may grow and shrink (temporary variables/end of scopes) during this process. In the end, a *Scope* component filters out the correct output variables. Data is passed as binary values.

The most important struct for variable handling is *ScopedVariables*. As already mentioned this struct represents the data vectors of various click-element components. Particularly, each *BodyComponent* contains two *ScopedVariables* structs: One for the input data vector and one for the output data vector. To recall, *ScopedVariables* is a collection of many variables. It maps unique variable names to variable information. In addition, this object keeps the order in which variables were added. *VariableInfo* contains its size and position in the data vector, thus the *ScopedVariables* struct alone can be used to encode and decode variables from and to a *BodyComponent*'s data vector. Basically, a *BodyComponent*'s input and output data vector sizes are defined by the variables defined in its *ScopedVariables* structs.

*Block*s contain a variable owner map. This data structure maps unique variable names to an owner *BodyComponent*. An owner is either the *Block* itself or a child of the *Block*. The owner map basically contains all the variables that are local to a *Block*. Initially, the *Block* owns every variable that is defined in its input ScopeVariable struct. For instance the root-*Block* is the owner of the input Go code's function parameters. Whenever a component (usually a binary expression component) writes to a variable in an operation, it claims ownership of the result variable. On variable declarations without assignment (via *var* keyword in Go) the *Block* claims ownership. On declaring assignment statements (*:=* in Go) the *Block* immediately declares the statement executing component as owner. In any case, declared variables in a *Block* do not get added to any *ScopedVariables* structs of the *Block*. The variable is local to the declaring *Block* and thus no input variable. The ownership process is especially important for more complex and optimized hardware generation and its use is more accurately discussed in Section 5.6.2).

Whenever a *BodyComponent* needs to access a variable (for instance a binary expression needs to read an operand), the component can request the variable from its parent *Block*. The parent *Block* first searches its owner map. If no owner inside the parent *Block* is found, the parent *Block* searches its own parent *Block* where the same procedure is executed as well. These recursive calls stop successfully if one of the (grand-)parents (aka outer scopes) finds the requested variable. In this case, all the affected *Block*s of the call tree add the variable to their input data vector since the variable came from outside their scopes. In addition, ownership of the variable is claimed by the innermost *Block*. The requesting component adds the variable to its input and connects itself to the owner component of the variable. This simply extends the affected data inputs by the size of the variable. If no owner of the variable is found, an invalid state is reached and go2async panics with an adequate error message. This implies invalid Go input code beforehand. This variable dependency search also handles predecessor and successor definitions of involved *BodyComponent*s.

Remarkably, variable dependencies of extended *Block*s, like *For-Block*s and *If-Block*s, are trivially handled by this recursive variable search. For instance a *For-Block* is simply the parent of its body-*Block*. Thus, whenever the body requests a variable outside a for-loop, the *For-Block* simply delegates the search to its parent *Block*. The variable is added to the *For-Block*'s and body-*Block*'s inputs implicitly.

A successfully fetched variable potentially passes through multiple components. As already mentioned, the *VariableInfo* contains the information to extract the variable from the source data vector. The exact location of a variable in a data vector might not be the same as in input data vectors of other components. *Binary Expression* and *Selector-Component*s are one of the few components that actually need to decode a variable during their execution. These components extract relevant variable information from their inputs. They interpret the binary values as defined by the operation and additional information provided in the *VariableInfo* struct (for instance whether the variable is signed or not). The decoding of input variables takes place in the calculation VHDL-process with the help of VHDL-aliases. These aliases map a VHDL-variable to a specific part of a data vector. This way the decoded variables can be named, thus the decoding is abstracted for the actual operation execution. The output variable is encoded (aliased) in a similar way into the output data vector. With the help of these aliases the operation process is conveniently generated in an easy-to-read way.

For further convenience, outputs of *BodyComponent*s are usually all the variables defined in the input *ScopedVariables* struct. With the exception of the *Binary Expression/Selector-Component*s, root-*Block*s, and *Scope*s. They overwrite the default behavior by specifically defining relevant output variables.

Fig. 5.4 shows an illustration of an example variable handling in go2async. The example contains a *Binary Expression-Component* with the operation `a=b+1`. Thus, the *Binary Expression-Component* needs access to the variables *a* and *b*. The constant 1 is trivially handled by the calculation process. The direct parent *Block* of the example *Binary Expression-Component* does not declare any variables and therefore has to delegate the

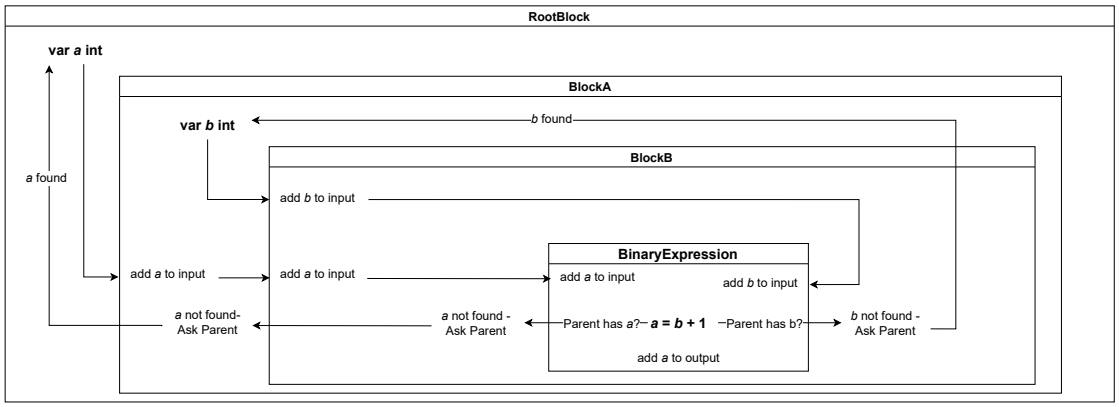

Figure 5.4: Variable handling in go2async.

variable search recursively to its parent. As intended, whenever a variable is found, the variable information travels the arrow path backwards. This process adds the found variable to each input of every affected component which had no prior access and knowledge of the searched variable. Lastly, the *Binary Expression-Component* adds the variable *a* specifically to its output because it is the result variable of the expression.

5.6 Handshake Connection handling

Handshaking is handled by the *HandshakeChannel* struct. This struct handles the handshake signals of click-element structures, specifically *req* and *ack* in both directions. Each *BodyComponent* has at least one input *HandshakeChannel* and at least one output *HandshakeChannel* (for instance a *fork* component requires at least two output *HandshakeChannels*). The actual handling of these channels vastly differs depending on the two big differently operating modes of go2async, namely *Sequential* and *Dataflow* branches. The two branches generate completely differently operating hardware. However, the semantic functionality stays the same.

5.6.1 Sequential Operating Mode

The sequential operating mode of go2async generates hardware in which each operation and component in a *Block* sequentially handshakes in order of occurrences in the Go input code. This makes the resulting hardware operate successively instruction by instruction. This behavior is more or less like the input software would be executed on a CPU. The generated hardware mimics software executions with a single thread. This is the default behavior of go2async.

The order of operation in an asynchronous circuit based on click-elements is given by the connections of the *HandshakeChannels* of each component. A component only starts operating when it receives an incoming request signal. The component outputs

an outgoing request signal when it is done. This can be used to enforce an order of operation.

In this operation mode, a *Block* handles the order of operation of its children. Initially, a *Block* connects its incoming request signal directly to its outgoing request signal. This means that if there are no children (operations) in a *Block*, it is immediately done. The first added child of a *Block* gets the *Block*'s input *HandshakeChannel* as its own input and analogously the child connects its outgoing handshakes to the *Block*'s output. The last added child is important for the next component additions. Each new child gets the last previously added child component's output handshake signals as inputs and replaces the *Block*'s output handshakes with its own. Data dependencies, which are extracted as described in Section 5.5, are completely ignored and ignored in this operating mode. The order of operation is entirely defined by the *Block*'s handling of its children.

5.6.2 Dataflow Operating Mode

The goal of the dataflow operating mode is to make use of a hardware's ability to exploit as much parallelism as possible. To achieve this, the data and variable dependency evaluations described in Section 5.5 are applied. This mode removes a *Block*'s singular responsibility for operations ordering of its children.

The resulting hardware contains components which make use of their predecessors and successors defined by data dependencies. These dependencies are given by read and written variables of components. Successors are components which need to access (read or write) an owned variable. Reversely, predecessors are components which own needed (read or write) variables. Whenever a variable is written to, the writing components claim ownership of the variable.

Recall that a *Block*'s output variables are also its inputs per default. In this operation mode, it is important for a *Block* to adopt ownership of variables their children have written to. This is because a *Block* represents a Go code scope. This implies that through the eyes of a *Block*'s parent-*Block*, the *Block* is the component that has written to a variable because the parent-*Block* does not see a child-*Block*'s inner components.

The handshakes to successors and from predecessors of a component are handled via *Join* and *Fork* components. A *Join* component is able to handle arbitrarily many input handshakes (predecessors) and merges them into one outgoing request if all incoming request signals have the same phase. In practice, the *Join* component's click-function is similar to the *and-reduce* function (see *F()* in Fig. 4.1). The *Join* component is used if multiple dependencies from different sources are needed or for general synchronizations.

Conversely, a *Fork* component translates a single input request handshake into many outgoing requests (successors). Here an *and-reduce* is used to merge the acknowledgement signals of the outgoing requests to a single one that is used to acknowledge the single original input request handshake. The *Fork* component is used to start multiple components which are data-independent to each other.

This leads to a circuit which operates according to data dependencies. Instead of waiting for all previous operations to finish before being able to execute, an operation now only waits for its predecessors. In the best case, the resulting circuit has operations only dependent on function parameters (the root-*Block*), thus each operation can start simultaneously. The worst case is that all computations in a function depend on each other. Therefore, the worst case converges into the sequential operation mode.

The dataflow approach allows faster circuit operation obviously enabled by parallelism. The downside is greater hardware area usage precipitated by the additionally required *Fork* and *Join* components.

5.6.3 Visualization of both Approaches

```
1   package goexamples
2
3   func g(a,b int) int {
4       c := 0
5       a = a + 1
6       {
7           c = c + 1
8       }
9       b = b + 1
10      c = c + c
11      a = a + b + c
12      return c
13   }
```

Listing 5.1: Example Go function: Variable additions.

Listing 5.1 shows an example Go code of random variable additions. In Fig. 5.5, a visualization of the two aforementioned operating methods which were extracted from the Go code example can be seen. The sequential approach is straightforward. Here the components and operations simply are calculated as expected in the order of occurrences of the Go example code. The arrows indicate the handshake direction. Acknowledgment signals are omitted. The code block which includes the instruction `c = c + 1` generates a child *Block*. The child *Block* simply propagates its handshakes to its inner binary expression component. Note that the operation `a = a + b + c` is recognised as nested binary expression and is therefore split into two binary expressions.

The dataflow operating mode exploits parallelism by executing data independent operations. In this case, operations `c := 0` , `a = a + 1`, and `b = b + 1` can be executed in parallel when starting the *Block* and thus save time. *Fork1* forks the root-*Block*'s request signal into each relevant component. Operations requiring only *c* default into a sequential-like behavior.

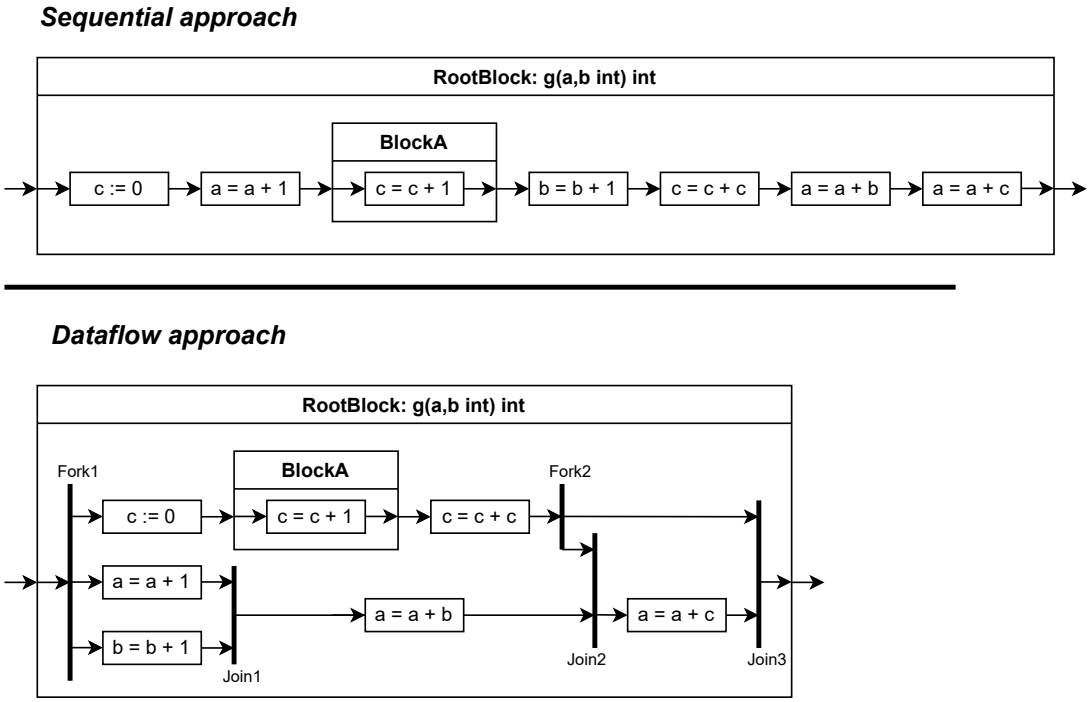

Figure 5.5: Sequential and Dataflow approach visualization example extracted from Listing 5.1.

The nested binary expression `a = a + b + c` triggers a chain of different processing events. Just like in the sequential approach, the operation is split into two binary expressions namely `a = a + b` and `a = a + c`. The first part of the original expression is independent from c and can thus be executed before the upper path of the Dataflow visualization in Fig. 5.5 finishes but obviously after the preceding operations on a and b (synchronized via *Join1*). Operation `a = a + c` has to wait for every other operation to finish before it can correctly calculate its intended result.

On one end, *Fork2* forks the path calculating c to its successor, which happens to be the root-*Block*, because the component calculating $c = c + c$ is the owner of result variable c. On the other end, *Fork2* forks its outgoing handshake to *Join2* which synchronizes the two big independent calculation paths to allow the last instruction to be executed.

Note that *Join3* cannot be omitted even though the last instruction is completely irrelevant for the result of the function. It is important to synchronize every path at the end of a *Block*. The last join is responsible to generate the handshake acknowledgement signal for each path ending with components with no successors. If omitted, these paths would get stuck in a phase, therefore unable to accept more inputs and thus deadlock the asynchronous circuit.

Remarkably, the arrows in the *Dataflow approach* of Fig. 5.5 are equivalent to the data-connections in both operating modes and can be seen as a data dependency graph.

5.7 Parsing Go with the built-in Go AST

Go2async parses Go with the built-in Go AST and performs the source code-to-hardware mapping by constructing relevant components simultaneously on the fly. The adapted Go parser of go2async needs two parameters to do its job: The size of the *int* type (default 4; adaptable via program parameters), and the path of the Go input file. The parsing starts by fetching all declared functions in the file and saving them in memory. This is needed, in case any function calls another. Afterwards, the generations of *Scope* components per function already begins.

As previously mentioned the *Scope* component consists of a *Block* (more precisely: a root-*Block*) and registers. Therefore, the root-*Block* is constructed. Parsing a function's signature provides everything we need to know about a root-*Block*'s interface. The inputs are a function's parameters, and the outputs are the function's result variables. A method for parsing the Go AST's variable expressions is provided to translate its various fields into a uniform variable struct usable for go2async's components. Subsequently, the function body can be parsed to construct the root-*Block*'s children. As mentioned in Section 5.2, exactly one return statement at the end of a function is expected. Its existence is checked before it is deleted from the statement list of the function body for the children generation.

The Go AST's function body basically consists of a list of statements. The *Block* generating method loops over a slice of statements to generate its children. The big benefit of using interfaces and type hierarchies comes into the spotlight. Each statement is translated into some component which implements the *BodyComponentType* interface. A statement could potentially generate any of the components defined in Section 5.4. However, the *Block* does not care which struct type a child has, as long as the generated child implements the *BodyComponentType* interface. A child is generated for assign-statements, for-statements, if-statements, and block-statements. Declare-statements do not generate a child. These statements basically register a variable of a given type and size for later use in the parent-*Block*. Any other Go statements lead to an error-state.

A *Block*-statement just contains a list of statements. This statement simply triggers a recursive call to the same *Block*-generating method as before. The parent of the new *Block* is the *Block* this statement resides in.

The assign-statement is the main driver for a *Block*'s input and output data vector. An assign-statement (=) triggers the variable search of *Block*s. As described in Section 5.5, whenever the parent *Block* cannot find a variable, it searches its (grand-)parents and adds the variable to its input. In the default case, an assign-statement generates a binary expression component. There are two different outcomes: In the case that the RHS is a single variable, constant, or array, the constructed *Binary Expression-Component*

simply assigns one of its inputs directly to its result with no operation. In the case of the RHS being a binary expression, a *Binary Expression-Component* is constructed that mimics the software's behavior in hardware. In the complicated case of nested binary expressions, this process is done recursively and multiple chained *Binary Expression-Component*s are generated. An additionally allowed RHS expression is the call-expression. In this scenario a *CallBlock* or *FunctionBlock* is generated instead of the usual *Binary Expression-Component*.

A for-statement generates a *For-Block* component. The *Selector-Component* for the *For-Block* is constructed by parsing the condition expression of the for-statement. A *Block* is generated by parsing the loop's body statements. This calls the same *Block* generating method as before. Thus this process is recursive. The parent of the block generated by the body statement is the *For-Block*. The *For-Block* already handles its inner components itself.

An if-statement generates an *If-Block* component. Similarly to the for-statement, the condition expression is responsible for the generation of the *Selector-Component*. A *Block* component for the then-path is generated by the usual *Block* generating method. The else-path is optional. Therefore, the default *BodyComponentType* for the else-path is a binary expression with no operation (this cannot be omitted and has to be done for valid handshaking). In case an else-path exists it generates a *Block* just like for the then-path.

Each component generated by this recursive parsing process is saved in a list in memory. The parsing process also handles numerous errors. Whenever unsupported or erroneous Go input is given, go2async's parser part prints the exact location (line and column in the input Go code) in which the unexpected input resides. The error message is also accompanied by an input rejection reason.

5.8 Hardware Generation

This is the last program execution step of go2async and is responsible for the VHDL output generation. As previously mentioned, components described in 5.4 are saved in a list while walking the Go AST. The VHDL generating process iterates through this list to generate hardware.

The *Component* interface enforces methods for VHDL-entity, component, and architecture generations for each component. The generation is split into three parts. First, the output contains library definitions and miscellaneous constant definitions. Afterwards, each generated component's entity is written. An entity is the primary definition of a hardware component. The last step generates various VHDL architectures describing hardware for the previously written entities.

A component's architecture defines its hardware functionality. Many components have a predefined architecture structure with predefined components and connections. These components' architecture is trivial. A few exceptions make this process complicated. The *Binary Expression-Component* and *Selector-Component* need to generate a VHDL

process which decodes its inputs into usable variables, executes a operation, and writes the result to an output variable.

The basic *Block* needs to describe the variable and data dependencies of its children. Even though the *Block* is not necessarily responsible for its children handshake and data connections (as described in previous sections), a *Block*'s architecture needs to instantiate any child component and thus implement and finalize each connection nonetheless. This process entails signal definitions of each handshake and data signal, handshake signal assignments, data/variable signal assignments, and children component instantiating. The signal definitions can be directly fetched from the affected child component. Signal assignments are given by the *HandshakeChannel* and *DataChannel* structs of the *Block* itself as well as its children.

Finally, go2async generates *Scope* architectures at the end of its output. The generated VHDL code, in addition to third-party definitions consisting of click-element structures from [10], can be used to synthesize asynchronous circuits based on click-elements.

5.9 Generation Example

This chapter concludes by showing VHDL of generated hardware for a given Go example input with the dataflow model of go2async. Go2async's outputs get very large very fast. Therefore, the shown VHDL parts are abbreviated and a simple Go input function is used. For this purpose asynchronous hardware corresponding to a short *sum* function (see Listing 5.2) is generated.

```
1  package goexamples
2
3  func sum(a,b int) int {
4       a = a + b
5       return a
6   }
```

Listing 5.2: Example Go function: Calculate sum of two integers.

Essentially, go2async generates one *Binary Expression-Component* corresponding to `a = a + b`, a *Block*, and a *Scope* component. The single (root-)*Block* is inside the *Scope* component where it neighbors a register which holds the output value *a*.

Listing 5.3 shows the generated *Binary Expression-Component* which calculates `a = a + b`. First the discussed variable alias mappings in the I/O data vectors are defined. Information about variable locations can be extracted here. For instance the location of *a* in the output vector is `3 downto 0` (line 8). With the alias mappings the calculation process is able to calculate the desired result (lines 21-28). The process casts the required signals according to the types as defined in the Go input code. The *Binary Expression-Component* also contains a delay component responsible for delaying the input request signal as required by the used operation (lines 12-19).

```
1
2  architecture beh_BEP_0 of binExprBlock_bep_0 is
3      alias x : std_logic_vector(4 - 1 downto 0)
4              is in_data( 4 - 1 downto 0);
5      alias y : std_logic_vector(4 - 1 downto 0)
6              is in_data( 8 - 1 downto 4);
7      alias result : std_logic_vector(4 - 1 downto 0)
8              is out_data( 4 - 1 downto 0);
9  begin
10     in_ack <= out_ack;
11
12     delay_req: entity work.delay_element
13       generic map(
14         NUM_LCELLS => ADD_DELAY  -- Delay  size
15       )
16       port map (
17         i => in_req,
18         o => out_req
19       );
20
21     calc: process(all)
22     begin
23         result <= std_logic_vector(
24                     resize(signed(x) + signed(y),
25                             result'length))
26                         after ADDER_DELAY;
27     end process;
28 end beh_BEP_0;
```

Listing 5.3: *Binary Expression-Component* corresponding to `a = a + b`.

Listing 5.4 contains an abbreviated architecture of the root-*Block* generated from Listing 5.2. The goal of this example is to primarily show the data I/O wirings of the *Binary Expression-Component* child. Signal definitions and assignments of the aggressively generated *Join* and *Fork* components are omitted. Remarkably, the data vector connections are in the same general directions as the handshake connections in go2async's dataflow model.

The root-*Block* starts by defining all required signals. These signals follow a certain naming scheme. The architecture starts with a *signalAssignments* process in which all the signal of the *Block* are assigned. The convenient usage of VHDL processes allows to define default signal assignments which can be overwritten in later lines of a process (see line 15 and 23). The process contains all handshake and data connections. Listing 5.4

```
architecture beh_b_0 of BlockC_0 is
    signal bep_0_in_req : std_logic;
    signal bep_0_in_ack : std_logic;
    signal bep_0_in_data : std_logic_vector(8 - 1 downto 0);
    signal bep_0_out_data : std_logic_vector(4 - 1 downto 0);

    -- Additional signal definitions (fork/joins - omitted)
begin
    signalAssignments: process(all)
    begin
        -- Handshake assignments (omitted)
        ------------------------

        -- Default block out_data is in_data
        out_data <= in_data(out_data'length - 1 downto 0);

        -- Data signal assignments for bep_0
        bep_0_in_data (4 - 1 downto 0)
                        <= in_data (4 - 1 downto 0);
        bep_0_in_data (8 - 1 downto 4)
                        <= in_data (8 - 1 downto 4);

        out_data (4 - 1 downto 0)
                        <= bep_0_out_data (4 - 1 downto 0);
    end process;

bep_0: entity work.binExprBlock_bep_0(beh_BEP_0)
        port map (
          -- Input channel
          in_req  => bep_0_in_req,
          in_ack  => bep_0_in_ack,
          in_data => bep_0_in_data,
          -- Output channel
          out_req => bep_0_out_req,
          out_ack => bep_0_out_ack,
          out_data  => bep_0_out_data
        );

-- Additional management components (fork/joins - omitted)
```

Listing 5.4: Conceptual root-*Block* architecture.

```
1  architecture sum of Scope is
2      -- signal definitions following a naming scheme
3  begin
4      -- Scope input to block
5      b_0_in_req <= in_req;
6      in_ack <= b_0_in_ack;
7      b_0_in_data <= in_data;
8
9      -- Block output to register
10     r_0_in_req <= b_0_out_req;
11     b_0_out_ack <= r_0_in_ack;
12     r_0_in_data <= b_0_out_data;
13
14     -- Register output to Scope output
15     out_req <= r_0_out_req;
16     r_0_out_ack <= out_ack;
17     out_data <= r_0_out_data;
18
19     -- block b_0 instantiation
20
21     -- register r_0 instantiation
22 end sum;
```

Listing 5.5: Conceptual *Scope* component.

shows the data connections of the operation `a = a + b` in lines 18-24. Afterwards all children components of the *Block* are instantiated to complete the signal wirings.

Listing 5.5 depicts the conceptual workings of a *Scope* component. As usual, the *Scope*'s architecture starts with signal definitions following a certain naming scheme. The architecture's body starts by assigning signals to realize the required signal directions. The *Scope*'s inputs are wired to the *Block*'s inputs. The *Block*'s outputs are wired to the register's inputs. Lastly, the register communicates with the *Scope*'s output.

The project source code, along with VHDL and Go examples, can be be found on GitHub [46].

CHAPTER 6

Simulation & Exploration of Limits

This chapter explores practical usages of go2async. Asynchronous hardware is generated from some Go code examples and simulated with VHDL testbenches in ModelSim. The simulation workflow for verification is explained by looking at generated hardware for a well known algorithm. An extensive testcase is also presented to showcase that go2async is capable of generating rather big and very complex circuits. Generated hardware is also downloaded onto an actual FPGA to highlight physical working usage.

6.1 Testing Requirements

Verifying go2async's generated asynchronous hardware requires testing of the features which were described in Chapter 5. This can be done by simulating go2async's results with the help of a simulation program (e.g. ModelSim). To be more precise, the following details need to be addressed in a testbench environment:

- **Certain Go input maps to dedicated hardware components**

  Expressions generate *Binary-Expression* components, if-statements map to *If-Blocks*, for-loops in the Go input code lead to generated *For-Blocks*, and block statements generate *Blocks*.

- **The general operation order is preserved**

  This includes correct handshaking and data connections. The dataflow mode needs components to handshake according to data dependencies.

- **Variable Scoping**

Components only get variables which they are required to read and write. This verifies the optimized variable handling.

- **Correct results are generated**

  Hardware components need to mimic the functionality of their software counterpart.

These bullet points will be addressed by the following sections. This chapter tests the final version of go2async and the dataflow mode only since the sequential approach only represents a more trivial components handshaking approach with the same data dependencies. The general simulation workflow is presented, which showcases the debugging and testing process of go2async with the help of the simple yet demanding greatest common divisor example. The presence of loops shows that generated asynchronous hardware is capable of a dynamic runtime depending on the input.

Furthermore, the exhaustive quicksort example is used to test every feature of go2async at once. This large testcase contains a huge amount of variables as well as multiple cascading scopes in form of blocks, loops and if statements with a network of variable dependencies. While simulation capabilities might be limited in this case it is shown that certain feature verification can still be done.

Additionally, successfully tested hardware is synthesized and downloaded onto an FPGA which enables testing in a physical setting.

6.2 Simulation Workflow

This section describes the general workflow on hardware simulations to test and verify asynchronous circuits based on click-elements generated by go2async. For this purpose, a simple example is chosen to simulate generated hardware. The GCD of two numbers will be calculated. Usually, this is done by solving the term 6.1 (*Euclidean algorithm*).

$$r_{k-2} = q_k * r_{k-1} + r_k \text{ with } r_{k-1} > r_k \geq 0. \tag{6.1}$$

$$q_k = \lfloor \frac{r_{k-2}}{r_{k-1}} \rfloor \tag{6.2}$$

In principle, this term depicts a division with remainder. The variable r_k is the remainder of the division $\frac{r_{k-2}}{r_{k-1}}$. The initial state $k = 2$ can be solved by setting $r_0 = a$ and $r_1 = b$. This recursive term *stops* when some $r_X = 0$ is reached. In this case $r_{X-1} = GCD(a, b)$.

Listing 6.1 shows Go code for calculating the GCD of the two input variables a and b. The code is written in a way such that go2async accepts it as input. If `b = 0` at the beginning, the function returns a which is the default GCD in this scenario. Otherwise the main loop (line 4) iterates as long as b is not equal to 0. The temporary variable t saves the old state of b (line 5). The typical modulo operation needed in this algorithm

```
1   package goexamples
2
3   func GCD(a, b int) int {
4       for b != 0 {
5           t := b
6           //b = a % b
7           c := a
8           for c >= b {
9               c = c - b
10          }
11          b = c
12          a = t
13      }
14
15      return a
16  }
```

Listing 6.1: Example Go function: Calculate the greatest common divisor of a and b.

(line 6) is not directly allowed according to the rules mentioned in Chapter 5. Therefore an inner loop has to be used to calculate the desired result (line 7-11). The outer loop ends by assigning the previous value of b to a and the remainder of the division in b (lines 11 and 12). Altogether this algorithm is equivalent to the recursive term from 6.1. Especially lines 11 and 12 in addition to the outer loop condition $b \neq 0$ conform with the stop requirement $r_X = 0$. In the algorithm particularly $r_X = b$(from line 11) $= 0$ and therefore $r_{X-1} = a$(from line 12) $= GCD(a, b)$. The function returns the GCD by returning a (line 15).

Fig. 6.1 contains a simulation of a testbench on an asynchronous click-element circuit generated from Listing 6.1. The hardware is a result of go2async's *Dataflow* mode. The testbench calculates the GCD of 15 and 6 (see *Input* divider). The inputs and result are shown on the upper part of the figure. The simulated hardware outputs 3 (see *Result* divider at the far right) as result which is in fact the easily verifiable solution to this example. In principle, at this point the structural verification process of resulting hardware structures generated by a particular go2async version is mostly done. The complex handshaking nature of asynchronous circuits operating on the 2-phase bundled data protocol (token rings) used by click-element structures makes it extremely unlikely that a circuit randomly terminates as expected by construction. This is especially the case if results of testbenches conform with results of software executions of the input Go input code. This trend manifests even more if the input Go code gets increasingly complex (see Section 6.3). However, an extensive functional verification process is required for computational components such as the *Binary Expression-Component* and *Selector-Component* because of the vast amount of possible calculations, especially if

arrays with variable indexing are involved. This is done by executing multiple typical software debugging techniques like edge-case testing in addition to manually inspecting generated hardware for a given input.

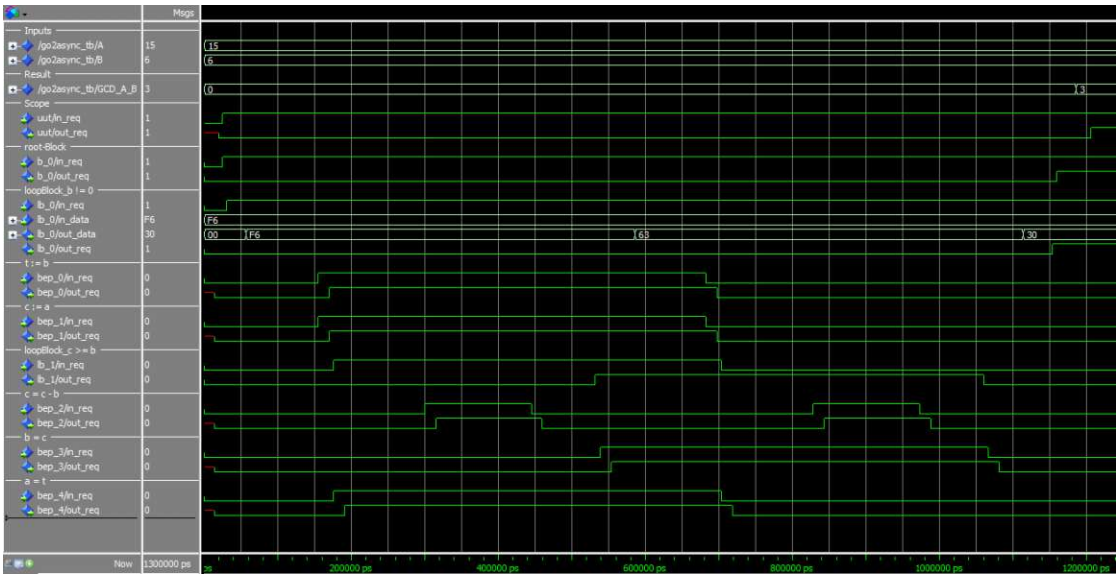

Figure 6.1: Simulation of an asynchronous click-element circuit generated from Listing 6.1 calculating the GCD of (15,6).

If further circuit inspection is required or desired the testbench in Fig. 6.1 also aids in this regard. As already mentioned, generated signal delays are very conservative. Their purpose is purely to generate simulations which enable to functionally verify generated hardware in a convenient manner. The delays ensure order between each component and make sure that there is a significant time difference between incoming and outgoing request signals of a component. The figure contains dividers between different components for better visualization. The dividers have a title describing the operation and component type. Below each divider are the incoming and outgoing request signal of the component described by the divider title. In particular each divided section does not contain other inner signals or data vectors with the exception of the outer loop (see *loopBlock_b != 0* divider). Initially, the hardware is reset and the inputs (15,6) are applied. The asynchronous circuit starts operation on an incoming request signal (*uut_in_req*) on the Unit Under Test (UUT) which is a generated *Scope* component (see *Scope* divider). This input request signal is directly wired to the root-*Block*. The outgoing request signal of the UUT is the last phase changing signal as expected.

Note that the result is valid if and only if the *Scope* component phase changes its output request signal. The time difference between the data change, root-*Block* outgoing phase change, and outgoing phase change of the *Scope* in the testbench figure originates from the additional time delta generated by a register in the *Scope* component. The signal prefix reveals that components of the same kind are called similarly. The prefix numbers

of these components are in order of occurrence in the input code. For example, `t:=b` is the first binary expression in the input thus its component is named *bep_0*.

The exciting part occurs between the phase change of the root-*Block*'s incoming request signal and phase change of its output signal. During this time all its children take their turns for their operations. First, it can be traced that the outer loop (see *loopBlock_b != 0* divider) has the same I/O phase changes as its parent-*Block* (conveniently the root-*Block*). This is because the outer-loop is basically the only instruction in the input function's main code block (return statements only define outputs). The outer loop's divider also contains the *Loop-Block*'s I/O data-vector to visualize the inner state of the loop. The data vector includes two 4-bit wide integer variables (a, b). The convenient hexadecimal radix setting helps to distinguish these variables in the data vectors. Careful readers might realize that the output data of the *Loop-Block* does not exactly conform with the state register of the loop. However, the output data vector is basically a delayed version of that (it also has to go through the loop's exit-DEMUX).

This visualization principle further shows that simulations are a form of abstraction of the tested circuit. If at any point an outer signal appears wrong, one can simply dig further into the circuit by displaying even more signals. Each layer of circuit inspection depth obviously leads to a increasingly complex simulation. As expected, the first executed instruction is the first instruction of the outer loop: `t := b`. The incoming request signal of the affected component occurs rather late because the signals have to traverse the outer loop's various register and controlflow components.

The dataflow mode of go2async allows the resulting hardware to execute the independent instruction `c := a` automatically and concurrently in parallel even though it appears later in code. After `c := a` is executed, the inner loop can start calculating the remainder of $\frac{a}{b}$. Additionally, the loop's last operation `a = t` can start concurrently with the loop, because it only depends on `t := b`. This leads to the last remaining instruction of the outer-loop being `b = c`, which has to wait for the remainder calculating inner loop. The divider for the binary expression `c := c - b` reveals that this instruction is executed twice which is indicated by two phase changes in the request signal paths.

If the implemented *Euclidean algorithm* is followed carefully, it is possible to verify even more operations. After the inner-loop's outgoing request signal changes its phase the instruction `b = c` does so next as expected. Some time after that the *out_data* changes from *F6* to *63* (hexadecimal). *F6* indicates the initial state of $a, b = 15, 6$. After the first iteration a is assigned b and since $15 - 6 - 6 = 3 = 15\%6$ the second vector value conforms with the first GCD algorithm values. The rest of the signals can be traced similarly to verify the calculation $GCD(15, 6) = 3$.

6.3 Exhaustive Simulation

Large and tricky Go examples are used to assess a state of go2async. The used testcases are a combination of many smaller ones. Functionally unnecessary code blocks and

cascaded if-statements are used to test the recursive hardware generation process. In this case, if a certain code block scope depth generates a valid result and so does an input version with an additional scope depth it can be inferred that the generation process is working correctly for an arbitrary scope depth by construction.

A convenient way to test go2async's dataflow mode is to input a GCD calculation version in which the example from the previous section (Listing 6.1) contains multiple independent GCD calculation loops. After the sequential approach verifies correct functionality, these loops should operate in parallel in the dataflow mode, which can be verified with the use of a simulation.

However, these assessment methods are only useful for functional testing for the thesis' author and usually not practically relevant and interesting. It is obviously more exciting to test go2async with a big useful exhaustive input code. For this purpose a quicksort implementation is showcased.

Listing 6.2 contains the go-to state assessment input Go code. The listing shows a non-recursive implementation of quicksort for *quickly* sorting an array consisting of six integers. In principle, queues in the form of arrays are used to save left and right bounds in which the input array has yet to be partitioned. The initial bounds encompass the whole input array (lines 4-7). According to the current queue bounds the input array is partitioned around the pivot element (lines 17-47). After each partitioning, potentially new sorting bounds are determined (lines 50-60). The algorithm stops when the bound queues are processed. This testcase is obviously very complex and contains every functional Go feature which go2async supports.

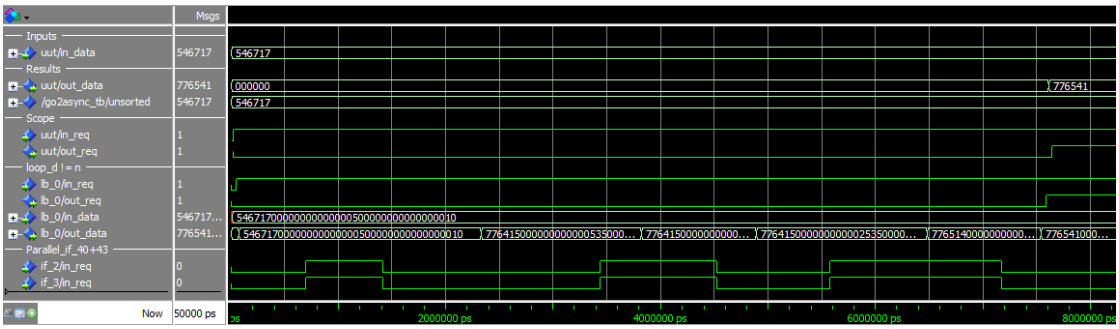

Figure 6.2: Simulation of an asynchronous click-element circuit generated from Listing 6.2 sorting the array $[5, 4, 6, 7, 1, 7]$.

Fig. 6.2 depicts a simulation of quicksort code from Listing 6.2. In the simulation the array $[5, 4, 6, 7, 1, 7]$ is sorted in a descending fashion. Since the input code is obviously rather complex the simulation is so too. Just like in the previous section, if the desired result is encountered most of the structural verification step is completed. At the top left of the figure we see the input array (a vector with six 4-bit wide integers). The right side of the *Results* divider reveals the correctly sorted array $[7, 7, 6, 5, 4, 1]$. Out of curiosity, a few more internal signals are simulated. The *loop_d != n* divider contains

```
1   package goexamples
2
3   func Quicksort(x [6]int) [6]int {
4       var l, r [15]int // Recursive bound states (left bound, right bound)
5       var n, d int     // Queue states
6       l[0] = 0         // First left bound: leftmost element
7       r[0] = 6 - 1     // First right bound: rightmost element
8
9       d = 0        // Current queue position
10      n = 1        // Occupied queue elements
11      for d != n { // loop until all queue elements are processed
12          li := l[d]
13          re := r[d]
14          i := li
15          j := re
16
17          p := j // Pivot position
18
19          // Partitioning
20          for i <= j {
21              {
22                  pivot := x[p]
23
24                  for x[i] < pivot {
25                      i = i + 1
26                  }
27
28                  for x[j] > pivot {
29                      j = j - 1
30                  }
31              }
32
33              if i <= j {
34                  if i < j {
35                      tmp := x[i]
36                      x[i] = x[j]
37                      x[j] = tmp
38                  }
39
40                  if i < 5 {
41                      i = i + 1
42                  }
43                  if j > 0 {
44                      j = j - 1
45                  }
46              }
47          }
48
49          // Determine left and right bounds for further partioning
50          if li < j {
51              l[n] = li
52              r[n] = j
53              n = n + 1 // Increment queue element number
54          }
55
56          if i < re {
57              l[n] = i
58              r[n] = re
59              n = n + 1 // Increment queue element number
60          }
61
62          d = d + 1 // Increment queue position
63      }
64
65      return x
66  }
```

Listing 6.2: Example Go function: Iterative quicksort for six integers.

the I/O request signals of the outermost loop in addition to its I/O data vectors. Here it can be seen how enormous internal data vectors can get depending on a component's data dependencies. Whenever debugging of such large vectors is required, go2async's *debug* flag can be used to help with variable locations of an inner component's I/O data vectors. In this case, the x input array is located at the start of the vector. This allows to easily see what happens after each iteration of the outer loop. It is possible to infer that six iterations were needed to sort the input array. Additionally, Fig. 6.2 contains the input request signals of the if-statements from Listing 6.2 (lines 40-45) to show that the dataflow mode of go2async generated hardware is capable of executing these two deeply nested independent if-statements in parallel.

6.4 The High-Level Synthesis Tool in Practice

It is possible to practically use and test go2async's generated asynchronous hardware based on click-elements with a given FPGA. To prove this, the DE0-CV [14] development board is used to download generated hardware. This thesis presents the two previously explained algorithms in a practical setting. In particular, hardware for the GCD algorithm from Listing 6.1 is generated and shown. Afterwards, six integers will be *quickly* sorted with hardware derived from the Go code from Listing Listing 6.2.

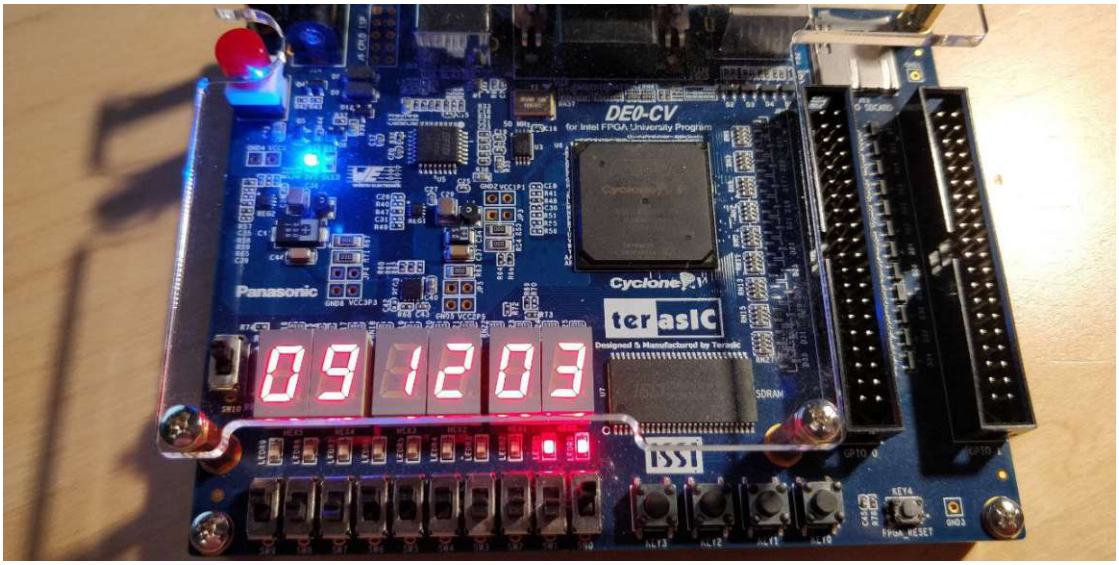

Figure 6.3: Calculating the GCD of 9 and 12 on the DE0-CV.

Fig. 6.3 contains the GCD calculation of 9 and 12. In this setting, 4-bit wide integers can be input via the first eight switches from the left. Four switches are needed for the first input, the next four for the second. Again from the left, the first two seven segment displays show the first input. The next two displays show the second input. The two most right switches are used to handshake with the asynchronous circuit. The right most one is mapped to the input request signal and thus responsible for calculation initialization.

The two most right LEDs are the *input acknowledgement* and *output request* output signals of the circuit to signal input acceptance and calculation completeness. All in all, Fig. 6.3 shows that the input request switch is turned on, the circuit acknowledged the input, requests an output, and the GCD result of 9 and 12, which is 3, can be seen.

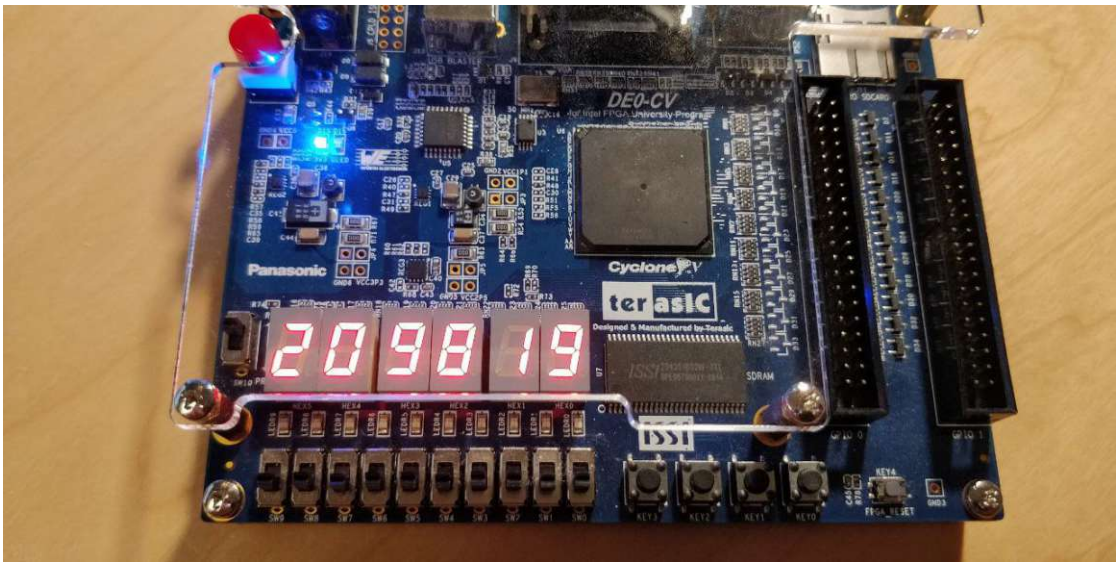

Figure 6.4: DE0-CV displaying an unsorted array.

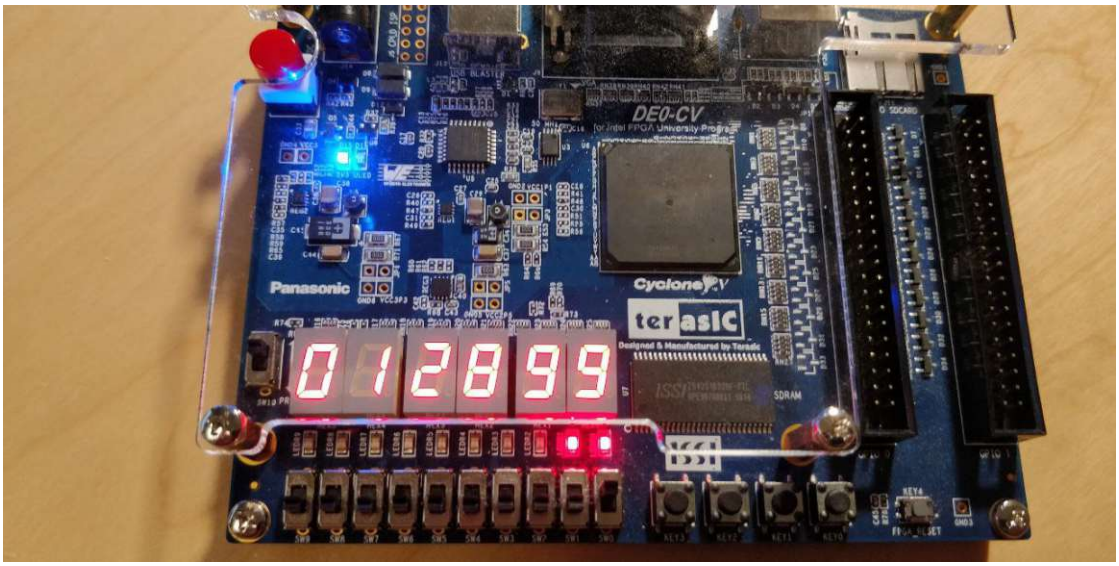

Figure 6.5: DE0-CV displaying a *quickly* sorted array.

Fig. 6.4 shows six unsorted numbers. These are some input numbers for the quicksort algorithm from Listing 6.2. The handshake interface is the same as before. After toggling

the rightmost switch, the status LEDs indicate input acceptance and output request (thus sort completion). The result can be seen in Fig. 6.5.

6.5 Limits of go2async

Remarkably, go2async's goal is to generate asynchronous hardware that functionally mimics its Go input code. However, there are some Go code structures for which no feasible asynchronous click-element circuit equivalent exists. The following list contains Go features which are not accepted by the current version of go2async and most likely never will be in future versions:

- Multiple *returns* in functions

  The implementation of this feature is simply not reasonable. It would require some sort of context and scope breaking in each *Block* up to the root-*Block* with output managing multiplexers for each possible scenario. The complexity would increase enormously for each additional return-statement and scope depth.

- *Goto*-statements

  Theoretically, this feature might be reasonably implementable if the *goto*-statement jumps to a label which is defined in the same scope. However, this case can only lead to endless loops and unnecessary forward jumps (code skipping). In case the jump label is in another scope this feature would become either impossible to implement or at least just as unreasonable as the multiple return feature.

- Loop *continue* and *break*-statements

  The loop *continue* and *break*-statements are very similar to *goto*-statements that jump forward to the end of the loop. However, as previously mentioned, the context breaking from a potentially very deep scope is simply unfeasible.

- *Switch*-statements

  Switch-statements themselves without *break* and *fallthrough* are basically the same as the already working *if/else*-statements. Thus, the current version of go2async simply expects *if/else*-statements instead.

  Go also supports the *break* and *fallthrough* keywords to either jump out of the switch or execute an additional (the neighboring/next) switch case also. Again, this requires complicated scope management. Especially the *fallthrough* case requires principally independent code blocks to be merged or duplicated. However, this idea is disregarded since potential solutions have too many structural disadvantages.

Luckily, each previously mentioned Go feature deemed unreasonable for direct asynchronous hardware generation has alternative Go code which is accepted by go2async's supported Go subset mentioned in Chapter 5.

One go2async pitfall are array index overflows. The Go input code programmer has to make sure that array index variables are never greater than an array's size and never smaller than zero. See Listing 6.2 where the code makes sure that i, j are never overstepping x's size and makes sure that j is never smaller than 0 (lines 6,7, 40-45). If at any point this is potentially the case, a simulation would most likely stop, crash, or abort and the hardware would yield undefined behavior on an FPGA.

Recursive functions are in theory possible with some limitations. However, as even Listing 6.2 shows, it is usually possible to rewrite recursive code into an iterative version and thus not deemed a desirable goal for early versions of go2async. The async-click library [10, 9] contains the idea on what recursive click-element hardware would look like. For this purpose a *Fibonacci* circuit is showcased.

It is not forbidden to input Go code containing an endless loop. However, Go code programmers have to decide for themselves whether their Go code makes sense and allows go2async to generate useful hardware. For instance, if an external interface in an endless loop is used (recall: function pointer parameter) it is actually possible to gain a potentially useful circuit.

CHAPTER 7

Optimization & Design Decisions

In this part of the thesis the main program states are discussed. Starting by explaining the initial solution which serves as a proof of concept for this project the ensuing major versions, which are given by substantial challenges, are presented. Go2async grew bit by bit via carefully chosen design decisions which entails adding big functional features and optimizing generated hardware leading up to the final state of go2async as presented in Chapter 5.

7.1 The Initial Solution

The first working solution acted as a proof of concept to verify that generating asynchronous hardware operating on the two-phase bundled data protocol using click-element structures based on work of [9, 10] works in principle. Differences to the final version is how hardware was generated as well as the supported Go subset was simpler. In particular, the input is restricted by following features:

1. A single function with a single parameter and single return value

   Exactly one function is expected in the Go input file. This implies that no function calls are possible yet.

2. No nested scopes

   This rule disallows the usage of additional code scopes (block statements).

3. Single scope if-else-statements

   If-statements have to include an else-path. Nested if-statements cannot be used.

4. Single scope for-loops

   Similarly to if-statements nested for-loops are not allowed either.

5. Variable declarations via := only

   The *var* keyword is not allowed.

6. Binary expressions only

   In particular, no nested binary expressions are supported. Using constants is allowed.

7. *Int* is the only allowed type.

   This is out of pure simplicity. In particular, arrays are not supported yet.

The goal of these rules was to enable an as easy as possible Go code parsing experience with the help of the Go AST. Obviously, the data structures of the initial solution are not as sophisticated as in the final version (explained in Section 5.4). However, they are similar enough such that a revisit of them is not in the interest of this section. Remarkably, non-nested scopes imply that *Blocks* were not part of the initial solution. An important fact is that the data structure state of the initial solution fit its simple parsing style and enables go2async to generate its first asynchronous hardware.

The initial solution concept implemented the sequential mode only. Generated components sequentially handshake in the order of the given Go input. The key difference to future versions of go2async is how variables are handled. The initial solution has a rather trivial concept: Gather all variable information used in every scope (main scope, if-statement-scopes, and for-loop-scopes in this case) and use a single data width for each component's input and output data vector. In principle, each component passes its potentially enormous input to its output. *Binary Expression-Components* are the only components performing operations on the input and thus have the ability to alter a data vector's state. *Binary Expression-Components* had to pick the correct spaces for corresponding operation variables in its input vector and determine where to alter data in the output vector.

Scope components are the only components with different I/O vector widths to more accurately represent the Go input function in hardware. The *Scope* component's input is as wide as the input function's parameter are and its output data vector's width is as wide as the input function's return values. That's also where the *Scope* component originally got its name. The *Scope* component consists of sequentially handshaking *Binary Expression-Components* and components representing if-statements and for-loops (there are no *Blocks* yet). The input vector of a *Scope*'s first component is initialized with 0-values but the spaces that correspond to the input function's parameter variables. The parameter variable spaces are wired to the *Scope*'s input vector. Similarly, the part of a *Scope*'s last component's output vector which corresponds to the input function's return variable is wired to the *Scope*'s output vector.

Fig. 7.1 shows a visualization of how hardware of Listing 7.1 generated by the initial solution conceptually looks like. As mentioned before, the initial solution generates a

```
1  package goexamples
2
3  func g(a int) int {
4       b := 0
5       c := 0
6       a = a + 1
7       c = c + c
8       return c
9   }
```

Listing 7.1: Example Go function: Simple variable additions.

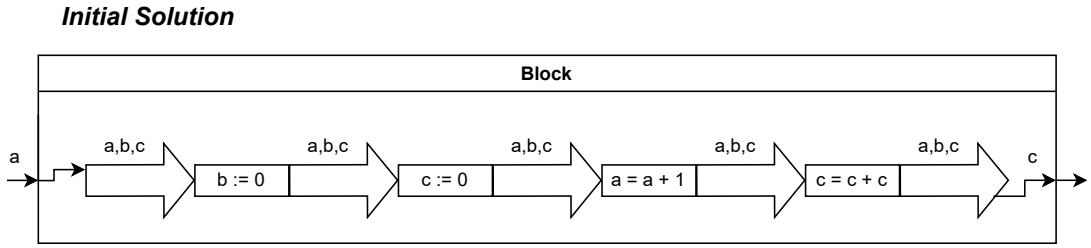

Figure 7.1: Initial solution hardware visualization of Listing 7.1.

sequentially handshaking asynchronous hardware as illustrated by the arrow direction. The figure shows that the *Block*'s I/O is small in comparison to the I/O of the inner components. The *Scope*'s input a is first merged into the input of the component calculating $b := 0$. The data vector between the *Block*'s inner components carries all three (a, b, c) occurring variables at all times. The data vectors stays the same inside the *Block*. Lastly, the return variable c is picked out of the big inner data vector.

The initial concept turns out to be a successful proof of concept that generates correctly operating asynchronous hardware based on click-elements. This should act as a foundation for future versions of go2async.

7.2 Array Support

The first major improvement of the initial solution is increasing go2async's type support and most importantly the capability to handle array variables. For this to work the *VariableInfo* and *ScopedVariables* were extended and improved (see Section 5.4) as well as *Binary Expression-Components* and *Selector-Components* needed broader variable and data vector understanding. This feature enables go2async to parse turing-complete Go input software and thus generates more interesting asynchronous hardware for given inputs.

It turns out that arrays are basically just multiple variables of the same type stuck together into one variable. The size of an array is its base type size times the number of its elements. Accessing an array element requires an index. Hardware requirements force go2async to only support fixed-sized array because hardware needs to know a type's exact size. In the data vector between hardware components an array variable occupies its whole space in one big continuous package. In general, this quickly leads to rather big data vectors if arrays are used in the input code. Array elements are located from an element's index +1 times the array's base type's size *downto* the element's index times the array's base type's size. Equation 7.1 shows how array indexing is done in VHDL.

$$array\_element <= array((index + 1) * type\_size \text{ downto } index * type\_size); \quad (7.1)$$

Remarkably, *Binary Expression-Components* and *Selector-Components* are the only components that need to deal with variables. Additional complexity is added because these components need to fetch an array's position inside their data I/O vectors before they can start indexing an array to use certain elements for their computations.

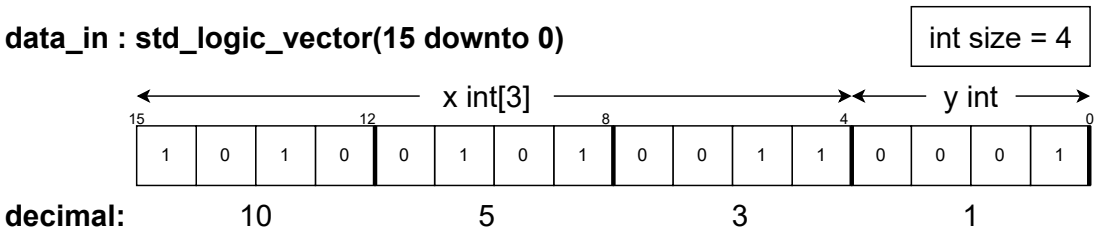

Figure 7.2: Example data vector containing one integer array and one integer variable.

Fig. 7.2 visualizes an example input data vector of a component. The data vector contains two variables: An integer array x which is 3 integers long and the integer y. The *int* size is set to 4 thus the vector is 16 bit wide. For example to access $x[2]$ equation 7.1 has to be executed with an additional consideration that the array in the *data_in* vector starts with an offset equal to 4. In this example $x[2] = 10$.

The biggest challenge here is to get go2async to support non-constant (variable) indexing. It is possible to solve this purely in the VHDL hardware description with the help of process variables to extract the current value of a the index variable from the input data vector as well as variable aliases (see Section 5.4). For example to access $x[y]$ in Fig. 7.2 a *Binary Expression-Component* would have to first decode x with offset from *data_in* as before and then also decode and fetch the value of y. The value of y can be stored in a process variable (for instance by declaring *index* as non-constant in equation 7.1). In this example $x[y]$ can be calculated to be decimal 5.

7.3 Recursive Blocks

This feature enables great programming conveniences for input files. It is finally possible to write nested for-loops, if-statements, and code scopes. For this to work, the *Block* component was introduced. The *Scope* component now only deals with the root-*Block*. The *Block*-parent and its children behavior of components is implemented. This feature also introduces the more sophisticated components *For-Block* and *If-Block* representing for-loops and if-statments respectively. The general theme "*Every code scope is a Block*" is implemented. Additionally, the parent-*Block* and children-components scheme is introduced.

Recursive *Blocks* brought a few additional advantages. Instead gruesomely tracking and declaring signal variables for component connections a *Block* allows to govern its children connections and VHDL signal handling with ease by applying certain signal naming schemes.

Blocks also introduced a more structured hardware. The nested layers of *Blocks* enable a more satisfying simulation workflow. It is now possible to peel the resulting hardware simulation like an endless onion which also greatly aids in debugging.

It turns out that the recursive *Block* structure is also a very convenient and natural fitting design principal for many future feature implementations of go2async.

7.4 Calling Functions

Before implementing function calls go2async added support for multiple function parameters and return values as well as mulitple Go functions in the input file. As explained in Chapter 5 there are two different components introduced to handle function calls:

1. *Call-Block*

   The goal of this feature was mainly to mitigate duplicated code in the input file and to get rid of for-loops required to implement the multiplication and modulo operator functionalities. In theory, this component's implementation is almost trivial. Simply instantiate the *Scope* component generated from a function declared in the same input file. The current variable handling (single width across all components) made it inconvenient to handle the instantiated *Scope*'s I/O data vectors. Luckily, the click-element handshaking behavior allows this feature to just work from the moment a *Call-Block* generates valid VHDL code.

2. *Function-Block*

   Recall a *Function-Block*'s purpose is to call an external interface conforming with the click-element interface. The big challenges with this feature is the capability to parse function pointers in a input function's parameter list. This requires a recursive form of variable parsing which is bug-prone. Additionally, this feature

requires *Blocks* to extend their interfaces by an additional inverted click-element interface (see Fig. 5.2).

While this feature was complex and tedious to implement it again can be considered to work correctly from the moment it generates valid VHDL code. This fact greatly shows the advantage of using the already verified and working click-element interface and structures from [9, 10] this thesis' project is based on.

This feature enables go2async's generated hardware to make use of existing external hardware. For instance it is now possible to communicate with asynchronous RAM or even synchronous hardware (although synchronization techniques are recommended) to make use of a synthesis tool's Intellectual Property (IP)-Catalog (i.e. library of predefined and preverified hardware structures).

7.5 Nested Binary Expressions

Up until now it was only possible to write binary expressions. This often lead to an annoying restructure need of the Go input file. The Nested binary expression support yields a better Go programming experience for input files. This feature requires rewriting parts of go2async's parser and AST handling and thus also triggers a larger refactor process of this thesis' project.

In practice, nested binary expressions are expressions that consist of more than one operator and more than two operands. Such expressions still need to be decoded into multiple interdependent binary expressions (compare Fig. 5.5). For this to work, the parser either needs to chain binary expressions or/and introduce new temporary variables for intermediate results. The first method can be done if a nested binary expression contains only one operator between its operands. To be more exact, the operator needs to be commutative. However, this is the case for all supported operations.

Intermediate results (and thus temporary variables) might be needed if multiple operators are in use in an expression. The main issue here is, that the introduction of additional temporary variables requires the single width data vector to be expanded for the whole circuit even though the new variables might not be relevant for the majority of the circuit. This is especially unfortunate if the circuit consists of many components which contain registers (e.g. *For-Blocks*). Temporary variables might unnecessarily waste numerous hardware components, especially registers which is typically a scarce resource on FPGAs.

7.6 Optimizing Variable Handling

Nested binary expressions triggered the need of a different variable handling technique because of the introduction of temporary variables which unnecessarily widens the uniformly sized internal data vector most of the time. The main goal is to get rid of the single width data vector across all components. The idea is to implement dynamically sized vector depending on the accessible variables of a code scope and code lines (*Block*s

```
1   package goexamples
2
3   func g(a int) int {
4       a = a + 1
5       b := 5
6       {
7           c := 1
8           a = a + c
9       }
10      d := b + a
11      return d
12   }
```

Listing 7.2: Example Go function: More variable additions.

and components in hardware). Up until now variable handling happened globally. An object available at any time during runtime tracked the available variables, defined the size of the single width data vector, and governed variable positions.

The optimization idea is to move the global variable tracking into the very conveniently placed *Blocks*. As mentioned before *Blocks* represent a code scope. As it happens, typical programming languages (Go included) introduce variable scopes and lifetimes. Thus *Blocks* are a near perfect match for this responsibility. Now *Blocks* govern all variables declared in their direct scope (excluding *Block* children) and thus define their children's data vector sizes. *Blocks* start with a data vector given by all variables which come from outside its scope. The vector grows with each variable declaration on the component responsible for the variable declaration (the variable space is prepended to the vector). *Blocks* only output variables they also input since these are the ones a *Block*'s children potentially alter and thus are the variables interesting for the parent-*Block*. A *Block* does not have any insights of newly defined variables in its children *Blocks*. The parent-*Block* needs to merge child-*Block* outputs into its current data vector. The root-*Block* is capable of only outputting the variables defined in the *return* statement of the input function. This approach generally vastly improves hardware resource usage.

This approach allows certain tricks for optimised hardware usage. For instance, a nested binary expression coded in its own scope generates a *Block* which contains the temporary data vector growth in a substantially smaller area. Listing 6.2 contains a similar optimization in Lines 21-31 (*pivot* variable is gone after the code scope).

Fig. 7.3 depicts an illustration of the growing and shrinking data vector given by Listing 7.2. The root-*Block*'s initial data vector is given by the input function's parameter a. The second instruction introduces a new variable, thus the data vector grows with the size of the new variable b. The child-*Block* *Block_A* is generated for the nested code scope. The only read and written variable is a which conforms with *Block_A*'s input

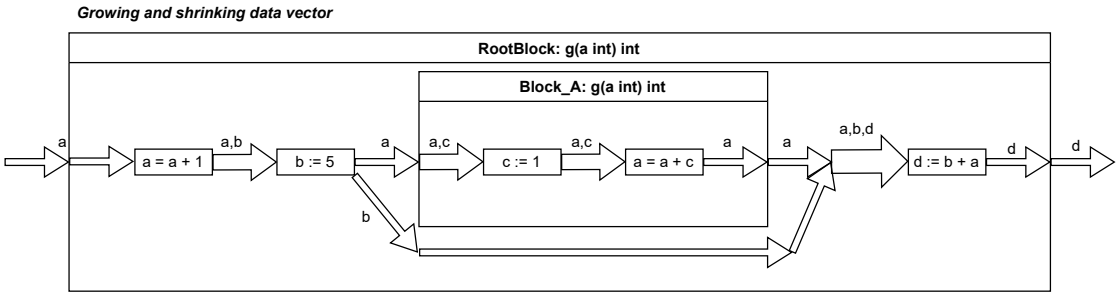

Figure 7.3: Growing and shrinking data vector illustrated for Listing 7.2.

in the figure. *Block_A* grows its internal vector by the size of *c* given by operation `c := 1`. Its output is only *a* as intended and the root-*Block* never gets access to *c*. After *Block_A*'s termination the root-*Block* needs to merge its data vector. In particular, *a* from *Block_A* needs to overwrite its corresponding space in the root-*Block*'s data vector. Additionally, another new variable *d* is needed for the last component. At this point the data vector is at its biggest. Lastly, *d* is the only variable output of the root-*Block* which is given by the return statement.

A very convenient side effect is that existing variable handling (e.g. variable decoding of *Binary Expression-Components*) does not have to be altered at all. Each component still has to deal with the same variable handling data structures internally. The only difference are their sizes. Additionally, this feature fixed a substantial bug. Before, it was not possible to redefine variables in nested scopes. This would simply generate an error since the variable was already registered in the global variable tracker. However, Go actually allows this and so does this new variable handling technique.

7.7 Introduction of the Dataflow Model

The last major version of go2async introduced the dataflow model. The state of this version was explained in Chapter 5. The idea's origin comes from an additional urge to improve variable handling. The growing and shrinking data vector improved go2async's generated hardware a lot. However, variables were still often unnecessarily put into registers and wired, especially if variables were used in deeply nested scopes.

The first optimization idea was to introduce the variable ownership scheme. The work of the previous sections (especially 7.3 and 7.6) did numerous preparation in this regard. The handling of a *Block*'s input and output in addition to the data vector merging of a child-*Block*'s output did a decent preparation for this endeavour. The key was that *Blocks* not only track declared variables of their scope but also their origins. In principle, before the implementation of this feature a *Block* could only tell whether a variable is available or not. After the extension a *Block* is capable of returning a variable's owner component if asked. Instead of trivially wiring a data vector from and to the sequentially handshaking components in the order defined by the Go input code, this approach allows

a component to task its parent-*Block* to find a needed variable's owner and directly connect its input to the variable's owner component's output. Additionally, it is possible for a component to take ownership of a variable if the component writes to it. The variable finding and connection process is covered in Section 5.5 in more detail.

As before, existing inner-component variable handling (e.g. variable decoding of *Binary Expression-Components*) does not have to be altered at all. Each component still has to deal with the same variable handling data structures internally. The only difference are their sizes. Even more so since *Binary Expression-Components* now only usually get exactly two input variables. *Block*'s now handle complex data vector connections between its children. Components have to track their predecessors and successors. This approach further vastly improved FPGA hardware usage as data is never unnecessarily passed around.

Remarkably, this chapter still only described the design decisions and optimizations of the original sequential mode of go2async. Handshake behaviors did not need any significant changes at all yet. What is left is the actual dataflow model. The variable ownership feature essentially generates hardware with data connections conforming with the data dependency graph of the input code. This is exactly what is left for the dataflow implementation. The only truly untouched process of go2async is the handshake signal generation. This has to change.

The dataflow model requires asynchronous hardware that handshakes in the same directions as the data flows. The handshake management requires components to simultaneously wait for predecessor to finish their work and signal its successors operation completions. As covered in Section 5.6.2 this is done by special *Fork* and *Join* components. Luckily, these were basically already provided by [9, 10]. The given components only needed to be extended to accept multiple successors or predecessors generically.

This challenge basically changed go2async's connection handling completely. Before a *Block* was capable of generating connections of its children on demand with the help of its children's parameters. As described in Section 5.6.2, new connection handling objects have to be implemented which are able to generate the needed signals for its parent component. With this feature implemented go2async is capable of generating asynchronous hardware based on click-elements which exploits as much parallelism as the hardware allows coming from the data dependency graph given by the input Go code.

CHAPTER 8

Conclusion & Outlook

This thesis is essentially motivated by two driving factors. First there is a general urge to increase performance of current systems. One solution concept for this problem is to speed up specific tasks by creating dedicated hardware for them. However, this is a very risky endeavor which requires huge design efforts and specialised knowledge. The usage of HDLs and FPGAs hugely reduces design struggles and enables virtual testing. However, specialised knowledge is still required. The next step is to apply an additional abstraction layer to the design process to enable untrained personnel to create desired results by using HLS-tools. This thesis opts to use the Go programming language as interface for hardware creation.

The second big motivation driver are asynchronous circuits. The everyday hardware is synchronous. A global clock governs the circuit and determines its speed. The designs are fairly well researched. However, the design seems unnatural for event-driven or inherently asynchronous environments (e.g. real world) scenarios. Additionally, synchronous circuits are struggling with power efficiency. Asynchronous circuits solve typical problems of their clocked counterpart. Nonetheless, this type of circuit suffers from an enormous design and verification effort which is not viable without tool support.

This thesis presents go2async, an HLS tool to ease the development of asynchronous circuits. Go2async generates asynchronous hardware based on the preverified click-element pipeline structures [6]. Click-elements are one of the few structures which do not rely on Muller-C elements and are thus perfect for FPGA implementations. In principle, click-element circuits are composed of sequentially handshaking components. The deployed asynchronous handshaking protocol is known as the two-phase bundled data protocol. With the additional help of the click-library [9, 10] go2async is able to generate asynchronous hardware by parsing a subset of Go code. The generated hardware mimics the functionality of the input software. This enables Go software developers to generate asynchronous hardware without much knowledge about this type of hardware. The supported Go subset contains multiple functions, assignments, block statements,

if-statements, for-loops, as well as function calls. Most of Go's basic types are supported including arrays. This enables Go software designers to write Turing-complete and thus useful programs as a base for asynchronous hardware generation.

Go2async's development journey and choices were presented. The project started with a small proof of concept which only accepted the most basic Go input. After tests of the first version were successful the subsequent versions of go2async extended the supported Go subset. Many thoughts went into optimizing hardware results especially in regards to variable handling and register usage. A few syntax sugars were implemented such as nested binary expressions.

In the end, go2async now has two operating modes which are able to generate two different types of asynchronous hardware. The sequential mode generates sequentially handshaking hardware from the Go input code in a more direct fashion. Each sequentially occurring statement maps to a sequentially handshaking hardware component. Go2async's dataflow mode generates hardware in which components handshake depending on data dependencies. The dataflow approach exploits numerous parallelization possibilities while introducing hardware area overhead required for synchronization efforts.

Generated hardware was successfully tested. The simulation workflow was presented by showing simulations of a simpler input example (GCD) which enabled a complete overview and visualization of the inner workings of go2async's generated asynchronous hardware with the help of ModelSim. A quicksort implementation was used to edge test go2async's capabilities. This showed that go2async is able to successfully generate hardware for complicated inputs. Additionally, these two testcases were synthesized and downloaded onto an FPGA to showcase that the generated asynchronous hardware indeed works in a physical and real world setting.

With respect to the initially provided research questions, this gives a clear evidence that high-level synthesis from Go to asynchronous hardware is possible (question 1). The required restrictions of the Go functionality have been laid out in the thesis (question 2), and several measures for optimizing the resulting hardware have been introduced (question 3).

8.1 Future Work

Further work ideas mainly regard a larger supported Go subset. Some of the ideas potentially enable go2async to generate optimized hardware. The following list contains Go features which would be interesting for go2async to support in the future:

- **Struct support**

  Structs are essentially a cluster of multiple variables at once. This feature most likely will not improve generated hardware. However, this is a very useful software coding feature and should not be too hard to implement.

- **Assignment lists**

  Go supports multiple variable assignments in one line. For instance `a,b := 1,2` declares a, b and assigns 1 to a and 2 to b in one line. Go2async's dataflow mode actually already parallelizes such assignments automatically. However, it would be beneficial to allow the responsible component to realize these single line assignment lists in one go to save further resources.

- **Variable declarations in conditions**

  Go allows declaring and assigning a variable before writing the condition of an if-statement or for-loop. This is relevant for go2async because in the current version it is required to declare the condition variable before the condition it is used in. The condition variable is visible in the scope it was declared in and therefore potentially wastes precious resources.

- **Complex conditions**

  The current version of go2async only supports simple boolean expressions as conditions in if-statements and loops. Complex conditions are basically chained expressions and should be straight forward to implement in click-element structures. Complex *Selector* component chains should be perfectly fine in loops and if structures since the relevant MUX/DEMUX components wait for click-element structure typical handshakes from the *Selector* components. Therefore a longer path in the selector path would not destroy existing structures.

- **Goroutines**

  In software, goroutines spawn a thread. In asynchronous click-element hardware this would translate to an independently additional handshaking computation path. Sounds easy in principle. However, goroutines run in the same address space as the main thread. This means goroutines should be able to access variables of their parent scopes. Code in goroutines can theoretically alter variables declared in their parent scopes at the same time. Usually, software developers include some synchronization techniques for atomic variable accesses. This would be quite hard to do in asynchronous click-element hardware.

- **Channels**

  Channels are a way to send and receive data from and to different threads. They inherently behave in an asynchronous fashion. The sending and receiving of data is blocking which is similar to handshaking behavior. However, channels only make sense if go2async also supports goroutines.

In addition to better Go support, the following aspects would be beneficial to improve the current state of go2async and the generated hardware:

- **Optimize delays**

  Currently, the used delays are very pessimitic. An obvious hardware improvement path would be to study the needed delays for given operations in more detail. This would help synthesis tools a lot.

- **Sequential mode → Pipelined mode**

  As already mentioned, the sequential mode generates sequentially handshaking components in order given by the Go input code. It would be very interesting to see if it's possible to put registers after some elements to enable a pipelined operation mode. This could potentially vastly improve the throughput of a circuit.

  The dataflow approach could also make use of this. However, this would require analyzing the dataflow graph for nodes in which all dependencies meet. These meeting points could be generated artificially (e.g. annotations in Go code).

- **Hardware optimization via data dependency graph analysis**

  In the current version go2async generates almost everything that is specified in the input software without any additional processing. If programmers write unnecessary or bad performing software the hardware might contain useless components. However, the dataflow graph has the potential to allow go2async to analyze and omit irrelevant computation parts. For instance it would be possible for the example of Fig. 5.5 to determine that each computation in the lower part of the figure is completely irrelevant for the result of the function.

- **Dataflow mode: Fork/Join usage improvement**

  In the current version the usage of *Forks* and *Joins* is very aggressive. Each component always gets a *Join* component for its input handshakes and a *Fork* component for its output handshakes. However, there are many scenarios where this is simply not necessary. For instance, a circuit containing one *Block* which has only one child does not have to fork and join inner handshake signals at all.

List of Figures

Listings

Acronyms

ASIC Application Specific Integrated Circuit. 2, 15

AST Abstract Syntax Tree. 3, 8, 9, 22, 24, 25, 28–30, 33, 34, 52, 53, 72, 76, 85

CAD Computer Aided Design. 14

CFG Control Flow Graph. 16

CLI Command Line Interface. 35

CPU Central Processing Unit. 1, 15, 48

FPGA Field Programmable Gate Array. xi, xiii, 2, 4, 5, 10, 14, 15, 17–21, 25, 36, 59, 60, 66, 69, 76, 79, 81, 82

GCD Greatest Common Divisor. 21, 22, 30, 32, 60–64, 66, 67, 82, 85, 86

GPU Graphics Processing Unit. 15

HDL Hardware Description Language. xi, xiii, 2, 15, 16, 20, 22, 81

HLS High-Level Synthesis. xi, xiii, 2, 3, 5, 15, 16, 22, 23, 25, 26, 28, 33, 35, 81

IP Intellectual Property. 76

IR Intermediate Representation. 15, 16

LE Logic Element. 20

LHS Left-Hand Size. 30, 37

LUT Lookup Table. 17

MCU Micro-Control Unit. 20

NRTZ Non-Return-To-Zero. 14

OOP Object Oriented Programming. 40, 42, 45

RAD Rapid Application Development. 23

RAM Random Access Memory. 33, 76

RHS Right-Hand Size. 30, 37, 52, 53

RTZ Return-To-Zero. 14

SNN Spiking Neural Network. 14

UUT Unit Under Test. 62

VHDL Very High Speed Integrated Circuit Hardware Description Language (also VHDSIC). xi, xiii, 2, 3, 9, 10, 28, 34–36, 41–44, 47, 53–55, 57, 59, 74–76

Bibliography

[1] Vivienne Sze et al. „Hardware for machine learning: Challenges and opportunities“. In: *2017 IEEE Custom Integrated Circuits Conference (CICC)*. 2017, pp. 1–8. DOI: 10.1109/CICC.2017.7993626.

[2] Matthias Maiterth et al. „Power Aware High Performance Computing: Challenges and Opportunities for Application and System Developers — Survey & Tutorial“. In: *2017 International Conference on High Performance Computing & Simulation (HPCS)*. 2017, pp. 3–10. DOI: 10.1109/HPCS.2017.11.

[3] *Wikipedia ESP32*. https://en.wikipedia.org/wiki/ESP32. Accessed: 2023-09-26.

[4] L.S. Nielsen and J. Sparso. „Designing asynchronous circuits for low power: an IFIR filter bank for a digital hearing aid“. In: *Proceedings of the IEEE* 87.2 (1999), pp. 268–281. DOI: 10.1109/5.740020.

[5] Aneesh Raveendran et al. „A RISC-V instruction set processor-micro-architecture design and analysis“. In: *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*. 2016, pp. 1–7. DOI: 10.1109/VLSI-SATA.2016.7593047.

[6] Ad Peeters et al. „Click Elements: An Implementation Style for Data-Driven Compilation“. In: *2010 IEEE Symposium on Asynchronous Circuits and Systems*. 2010, pp. 3–14. DOI: 10.1109/ASYNC.2010.11.

[7] Filipp Akopyan et al. „TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1537–1557. DOI: 10.1109/TCAD.2015.2474396.

[8] Mike Davies et al. „Loihi: A Neuromorphic Manycore Processor with On-Chip Learning“. In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

[9] Adrian Mardari, Zuzana Jelčicová, and Jens Sparsø. „Design and FPGA-implementation of Asynchronous Circuits Using Two-Phase Handshaking“. In: *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 2019, pp. 9–18. DOI: 10.1109/ASYNC.2019.00010.

[10] *Async-Click-Library*. https://github.com/zuzkajelcicova/Async-Click-Library. Accessed: 2023-09-26.

[11] Olga Melnikova, Irina Hahanova, and Karina Mostovaya. „Using multi-FPGA systems for ASIC prototyping“. In: *2009 10th International Conference - The Experience of Designing and Application of CAD Systems in Microelectronics*. 2009, pp. 237–239.

[12] *Go Programming Language*. https://golang.org/. Accessed: 2023-09-26.

[13] *Synthesis tool Quartus*. https://www.intel.de/content/www/de/de/products/details/fpga/development-tools/quartus-prime/resource.html. Accessed: 2023-09-26.

[14] *DE0-CV Hardware design platform*. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=921. Accessed: 2023-09-26.

[15] C.J. Alpert, A. Devgan, and S.T. Quay. „Buffer insertion with accurate gate and interconnect delay computation“. In: *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. 1999, pp. 479–484. DOI: 10.1109/DAC.1999.781363.

[16] Deokjin Joo and Taewhan Kim. „Managing clock skews in clock trees with local clock skew requirements using adjustable delay buffers“. In: *2015 International SoC Design Conference (ISOCC)*. 2015, pp. 137–138. DOI: 10.1109/ISOCC.2015.7401696.

[17] Minh Huan Vo. „The Merged Clock Gating Architecture For Low Power Digital Clock Application On FPGA“. In: *2018 International Conference on Advanced Technologies for Communications (ATC)*. 2018, pp. 282–286. DOI: 10.1109/ATC.2018.8587596.

[18] Harekrishna Kumar, Anjan Kumar, and Vinay Kumar Deolia. „Enabling Concurrent Clock and Power Gating in 32 Bit ROM“. In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2018, pp. 1–6. DOI: 10.1109/ICCCNT.2018.8493779.

[19] Priti Gosatwar and Ujwala Ghodeswar. „Design of voltage level shifter for multi-supply voltage design“. In: *2016 International Conference on Communication and Signal Processing (ICCSP)*. 2016, pp. 0853–0857. DOI: 10.1109/ICCSP.2016.7754267.

[20] W.J. Bainbridge et al. „Delay-insensitive, point-to-point interconnect using m-of-n codes“. In: *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.* 2003, pp. 132–140. DOI: 10.1109/ASYNC.2003.1199173.

[21] Shanlin Xiao et al. „A Data-Driven Asynchronous Neural Network Accelerator“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.9 (2021), pp. 1874–1886. DOI: 10.1109/TCAD.2020.3025508.

[22] A. Kondratyev and K. Lwin. „Design of asynchronous circuits by synchronous CAD tools“. In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*. 2002, pp. 411–414. DOI: 10.1109/DAC.2002.1012660.

[23] Yu Zhou. „Investigation of asynchronous pipeline circuits based on bundled-data encoding: Implementation styles, behavioral modeling, and timing analysis“. In: *Tsinghua Science and Technology* 27.3 (2022), pp. 559–580. DOI: 10.26599/TST.2021.9010089.

[24] Olga Melnikova, Irina Hahanova, and Karina Mostovaya. „Using multi-FPGA systems for ASIC prototyping“. In: *2009 10th International Conference - The Experience of Designing and Application of CAD Systems in Microelectronics.* 2009, pp. 237–239.

[25] Andrew Canis et al. „From software to accelerators with LegUp high-level synthesis“. In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2013, pp. 1–9. DOI: 10.1109/CASES.2013.6662524.

[26] *Microchip LegUp*. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler. Accessed: 2023-09-26.

[27] *Wikipedia LLVM*. https://en.wikipedia.org/wiki/LLVM. Accessed: 2023-09-26.

[28] Jens Sparsø. „Current trends in high-level synthesis of asynchronous circuits“. In: *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*. 2009, pp. 347–350. DOI: 10.1109/ICECS.2009.5411011.

[29] Rui Li et al. „Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures“. In: *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 2021, pp. 1–8. DOI: 10.1109/ASYNC48570.2021.00009.

[30] Idan Schwartz et al. „Near-threshold 40nm Supply Feedback C-element“. In: *2011 3rd Asia Symposium on Quality Electronic Design (ASQED)*. 2011, pp. 74–78. DOI: 10.1109/ASQED.2011.6111705.

[31] J. Cortadella et al. „Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.10 (2006), pp. 1904–1921. DOI: 10.1109/TCAD.2005.860958.

[32] Argyrios Sideris, Theodora Sanida, and Minas Dasygenis. „High Throughput Pipelined Implementation of the SHA-3 Cryptoprocessor“. In: *2020 32nd International Conference on Microelectronics (ICM)*. 2020, pp. 1–4. DOI: 10.1109/ICM50269.2020.9331803.

[33] I. E. Sutherland. „Micropipelines“. In: *Commun. ACM* 32.6 (June 1989), pp. 720–738. ISSN: 0001-0782. DOI: 10.1145/63526.63532. URL: https://doi.org/10.1145/63526.63532.

[34] Cuong Pham-Quoc and Anh-Vu Dinh-Duc. „Hazard-free Muller Gates for Implementing Asynchronous Circuits on Xilinx FPGA“. In: *2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications*. 2010, pp. 289–292. DOI: 10.1109/DELTA.2010.40.

[35] Quoc Thai Ho et al. „Implementing Asynchronous Circuits on LUT Based FPGAs“. In: *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Ed. by Manfred Glesner, Peter Zipf, and Michel Renovell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 36–46. DOI: 10.1007/3-540-46117-5_6.

[36] Montek Singh and Steven M. Nowick. „MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15.6 (2007), pp. 684–698. DOI: 10.1109/TVLSI.2007.898732.

[37] Yuxuan Liu et al. „An asynchronous loop structure based on the click element“. In: *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*. 2017, pp. 1–2. DOI: 10.1109/EDSSC.2017.8126448.

[38] Zhiyu Li et al. „A Low-Power Asynchronous RISC-V Processor With Propagated Timing Constraints Method“. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 68.9 (2021), pp. 3153–3157. DOI: 10.1109/TCSII.2021.3100524.

[39] Jon Neerup Lassen. „FPGA prototyping of asynchronous networks-on-chip“. In: *M. Sc. thesis* (2008).

[40] *Python Programming Language*. https://www.python.org/. Accessed: 2023-09-26.

[41] Arun Kumar and Supriya.P. Panda. „A Survey: How Python Pitches in IT-World“. In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. 2019, pp. 248–251. DOI: 10.1109/COMITCon.2019.8862251.

[42] *The Top Programming Languages*. https://octoverse.github.com/2022/top-programming-languages. Accessed: 2023-09-26.

[43] *Rust Programming Language*. https://www.rust-lang.org/. Accessed: 2023-09-26.

[44] Dongdong Lu et al. „Analysis of the popularity of programming languages in open source software communities“. In: *2020 International Conference on Big Data and Social Sciences (ICBDSS)*. 2020, pp. 111–114. DOI: 10.1109/ICBDSS51270.2020.00033.

[45] *Wikipedia Turing Machine*. https://en.wikipedia.org/wiki/Turing_machine. Accessed: 2023-09-26.

[46] *go2async GitHub page*. https://github.com/SeWiede/go2async. Accessed: 2023-09-26.