Imperial College London
Department of Electrical and Electronic Engineering

Formal Verification of High-Level Synthesis

Yann Herklotz Grave

April 2024

Supervised by John Wickerson

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London and the Diploma of Imperial College London.

Supervisor:	Dr John Wickerson
Internal Examiner:	Prof George Constantinides
External Examiner:	Prof Xavier Leroy

Statement of Originality

I, Yann Herklotz Grave, declare that the work presented in this thesis is my own, and that any other work has been appropriately referenced.

Abstract

Latency, throughput, and energy efficiency are becoming increasingly important, leading to custom hardware accelerators being designed for numerous applications instead of using less efficient general processors. Alas, designing these accelerators can be an error-prone process, especially when using hardware description languages (HDLs) such as Verilog, which operate at the register transfer level.

An attractive alternative is *high-level synthesis (HLS)*, where hardware designs are automatically compiled from software written in a high-level language like C. This way, hardware designers can benefit from mature software development tools. HLS tools promise designs with comparable performance and energy-efficiency to those hand-written in HDLs, reducing the time needed to design new accelerators. Reasoning about behaviour at a higher level should also make the process less error-prone. Unfortunately, HLS tools are unreliable; Vivado HLS produces incorrect designs in 1.2% of randomly generated C programs, undermining testing performed at the higher level of abstraction.

In an attempt to improve this situation, I propose a formally verified HLS tool called Vericert, providing a computer-checked proof that ensures it only generates hardware designs that behave like the input software program. Vericert extends CompCert, an established formally verified C Compiler, with a hardware back-end.

One expects a verified tool to produce significantly worse hardware than existing optimising HLS tools, as each transformation has to be simple enough to be proven correct. Indeed, an initial version of Vericert was up to 8× slower than a state-of-the-art HLS tool called Bambu. However, by verifying *hyperblock scheduling* in Vericert, a transformation which parallelises instructions in regions of code without loops, hardware produced by Vericert becomes only 1.6× slower than Bambu without optimisations and 3.6× slower than optimised Bambu. This is encouraging, showing that a verified HLS tool is comparable with an existing HLS tool, while being guaranteed to generate correct hardware designs.

Acknowledgements

First and foremost I would like to thank my supervisor John Wickerson for his continued guidance and patience throughout the PhD. He has allowed me to explore freely, and was generous with his time whenever I needed help or advice. He has also always encouraged me to present my work in diverse venues, leading to many interesting discussions and connections. His wonderful ability to understand technical descriptions and expose the core contribution clearly and creatively has been invaluable, and has taught me how to communicate my ideas clearly. I would also like to thank my examiners George Constantinides and Xavier Leroy for the many thought-provoking discussions and helpful suggestions that lead to a much clearer dissertation and many new ideas to explore.

Next, I would like to thank my colleagues in the Circuits and Systems group for many interesting discussions and creating an enjoyable atmosphere in the lab: Jianyi for a joint journey through the PhD, Nadesh for introducing me to high-level synthesis and helping me run them in the first place, Aditya and Diederik for many discussions about compilers over drinks, my coauthors James Pollard, Zewei and Michalis for all their help on our publications, and Kate, Roy, Divyansh and Mariano for being amazing flatmates. Finally, I would like to thank Alex Dalton, Alex Montgomerie-Corcoran, Ben Biggs, Ben Chua, Cheng, Dan, Erwei, Guoxuan, Ian, Marta, Quentin, Sina and Zhewen for many insightful discussions and advice, as well as help working with FPGAs and hardware tools.

I also gratefully acknowledge the Cyber Security Centre (NCSC) for funding this PhD through the Research Institute on Verified Trustworthy Software Systems (VeTSS).

I have also had the great pleasure to be able to work in the Celtique group at Irisa in Rennes as a visiting researcher over the summer in 2022. I would like to thank Sandrine Blazy and Delphine Demange for their deep insights into the CompCert compiler and for countless interesting discussions on different verification techniques and how these could be integrated into CompCert.

I would also like to sincerely thank Sandro Stucki and Bor-Yuh Evan Chang for their invaluable guidance during my internship at Amazon Prime Video, as well as my colleagues

during my time there with whom I had enlightening discussions with: Daniel, Francesco, Franco, Horia, Ilina, Ioannis, Pauline, Philipp, Sarek, Stefan, and Vlad. Additionally, I would like to thank Stefan Zetzsche for all his help understanding Dafny.

I would also like to thank my family and friends for their invaluable support and understanding during the PhD. Thank you to my father and especially my mother for their continued support, helping me get through stressful times.

Finally, I would like to thank my partner and best friend Nikita for her continued support.

Contents

List of Figures

List of Tables

Abbreviations

ALAP as late as possible

ASAP as soon as possible

ASIC application-specific integrated circuit

Asm CompCert assembly language

AsmBlock CompCert KVX assembly block language

AST abstract syntax tree

BRAM block random-access memory

Btl block transfer language

C#minor CompCert intermediate language

CDFG control- and data-flow graph

CFG control-flow graph

Clight CompCert intermediate language

Cminor CompCert intermediate language

CminorSel CompCert intermediate language

CPU central processing unit

DFG data-flow graph

DRAM dynamic random-access memory

DSL domain-specific language

DSP digital signal processor

FPGA field-programmable gate array

FSM finite-state machine

FSMD finite-state machine with data path

GPU graphics processing unit

HDL hardware description language

HLS high-level synthesis

IP core intellectual property core

IR intermediate representation

LP linear programming

LSQ load-store queue

Ltl linear transfer language

LUT look-up table

Mach CompCert intermediate language

Rtl register transfer language

SAT satisfiability

SDC system of difference constraints

SMT satisfiability modulo theories

SSA static single assignment

VLIW very large instruction word

1 Introduction

As latency, throughput, and energy efficiency are becoming increasingly important, we are seeing companies move towards designing their own application-specific hardware accelerators tailored to their workloads instead of relying on general-purpose central processing units (CPUs) or graphics processing units (GPUs). By specialising the hardware to the application, the hardware can be optimised further than general purpose processors, unlocking better performance while often using less power. Apple and Google, for example, are integrating machine learning accelerators into consumer hardware to allow models to run more efficiently than if they used the CPU or GPU [Apple 2022; Gupta 2023]. Machine learning is an example of an application that benefits greatly from having dedicated and specialised hardware accelerators designed for it [Reuther et al. 2020].

Alas, designing these accelerators can be a tedious and error-prone process. Hardware is normally designed using a hardware description language (HDL) such as Verilog or VHDL, which operates at the register-transfer level where the hardware needs to be described manually. As the complexity of hardware designs increases, designing hardware at this level becomes increasingly difficult, because the low-level description of the hardware makes it time-consuming and expensive to thoroughly test to ensure that it behaves as expected. An attractive alternative is *high-level synthesis (HLS)*, where hardware designs are automatically compiled from software written in a high-level language like C. This way, hardware design can benefit from mature software development tools while working on the general functionality of the hardware, and then use a modern HLS tools such as LegUp [Canis et al. 2013], Vitis HLS [AMD 2023b], Intel i++ [Intel 2020a], Stratus HLS [Roane 2023] or Bambu HLS [Pilato and Ferrandi 2013] to produce the register-transfer level description. These HLS tools promise designs with comparable performance and energy-efficiency to those hand-written in an HDL [Gauthier and Wadood 2020; Homsirikamol and Gaj 2014; Pelcat et al. 2016]. This reduces the time needed to design new hardware accelerators and as the design is performed at a higher level, the process should also be less error-prone.

Verifying the functionality of HLS designs Compared to software, it is even more important to ensure that hardware functions as it is supposed to, because once the hardware has been taped-out into an application-specific integrated circuit (ASIC), it becomes impossible to properly fix the issue except through workarounds in software. This may come at a great cost in terms of energy usage and the performance of the system compared to fixing the issue in hardware itself [Bowen and Lupo 2020; Herzog et al. 2021]. These hardware faults can also often be exploited and can be hard to detect, even using state-of-the-art hardware verification methodologies [Dessouky et al. 2019], either because the correctness properties themselves can be hard to express, or because the state-space that needs to be explored by the tools is too large.

HLS should simplify the process of verifying the functionality of the hardware design. Verifying designs at the register-transfer level requires large engineering efforts because of the level of detail and the size of the design. Even testing such large designs can be problematic, because the size of the design often means one cannot simulate running the hardware for more than a few seconds. Instead, HLS moves the verification of the functionality of the design to a higher level, where less detail is exposed, making it possible for software tools to reason about the behaviour of the program instead. Although a recent survey by Lahti et al. [2019] describes that verification remains a time-consuming part of the design process even with the use of HLS, it finds that in general it still reduced the verification effort by half.

Unfortunately, there are reasons to doubt that HLS tools actually preserve the behaviour of the design, increasing the chance of there being exploitable hardware faults in the resulting accelerator, and making verification at the level of the high-level language less useful. Some of these reasons are general: HLS tools are large pieces of software, they perform a series of complex analyses and transformations, and the HDL output they produce has superficial syntactic similarities to a software language but a subtly different semantics. Other reasons are more specific: for instance, Vivado HLS has been shown to apply pipelining optimisations incorrectly[1] or to silently generate wrong code should the programmer stray outside the fragment of C that it supports.[2,3] Meanwhile, Lidbury et al. [2015] had to abandon their attempt to fuzz-test Altera's (now Intel's) OpenCL to hardware compiler since it 'either crashed or emitted an internal compiler error' on many

[1] https://support.xilinx.com/s/question/0D52E00006jt7LfSAI/crtl-cosimulation-failed-caused-by-pragma-hls-pipeline
[2] https://support.xilinx.com/s/question/0D52E00006hpMZSSA2/pointer-synthesis-in-vivado-hls-v201
[3] https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Pointer-Limitations

of their test inputs. More recently, Herklotz et al. [2021a] fuzz-tested three commercial HLS tools using Csmith [Yang et al. 2011], and despite restricting the generated programs to the C fragment explicitly supported by all the tools, they still found that on average 2.5% of test-cases were compiled to designs that behaved incorrectly.

Existing workarounds Aware of the reliability shortcomings of HLS tools, hardware designers routinely check the generated hardware for functional correctness. This is commonly done by simulating the generated design against a large test-bench. But unless the test-bench covers all inputs exhaustively – which is often infeasible – there is a risk that bugs remain.

One alternative is to use *translation validation* [Pnueli et al. 1998] to prove equivalence between the input program and the output design. Translation validation has been successfully applied to several HLS optimisations [Banerjee et al. 2014; Chouksey and Karfa 2020; Chouksey et al. 2019; Karfa et al. 2006; Youngsik Kim et al. 2004]. Nevertheless, it is an expensive task, especially for large designs, and it must be repeated every time the compiler is invoked. For example, the translation validation for Catapult C [Siemens 2021] may require several rounds of expert 'adjustments' [Chauhan 2020, p. 3] to the input C program before validation succeeds. And even when it succeeds, translation validation does not provide watertight guarantees unless the validator itself has been mechanically proven correct [e.g. Tristan and Leroy 2008], which has not been the case in HLS tools to date.

My position is that none of the above workarounds are necessary if the HLS tool can simply be trusted to work correctly. This dissertation explores the implementation of a mechanically verified and optimising HLS tool built on the CompCert verified C compiler [Leroy 2006, 2009b; Leroy et al. 2016]. The main thesis of this dissertation is therefore the following:

> **Thesis** A realistic and optimising high-level synthesis tool can be proven correct using an interactive theorem prover, guaranteeing the correctness of the hardware while also remaining practical and efficient.

1.1 Research Contributions

The main contributions of this dissertation is Vericert, a formally verified and optimising HLS tool. Vericert is written in the Coq theorem prover and comes with a machine-checked

proof that any output design it produces always has the same behaviour as the input C program. Vericert is automatically extracted to an OCaml program from Coq, which ensures that the object of the proof is the same as the implementation of the tool. Vericert is built by extending the CompCert verified C compiler with a new hardware-specific intermediate language and a Verilog back end. It supports many C constructs, including integer operations, function calls (which are all inlined), local arrays, structs, unions, and general control-flow statements, but currently excludes support for case statements, function pointers, recursive function calls, non-32-bit integers, floats, and global variables. The main research contributions of Vericert are the following:

Formulate overall correctness theorem with Verilog semantics First, I state the correctness theorem of Vericert with respect to an existing semantics for Verilog due to Lööw and Myreen [2019]. The key challenge here involved integrating the hardware semantics within CompCert's model of computation and calling convention. This required specifying the external module interface used to interact with the final hardware produced by Vericert, for example specifying how the hardware can be reset, and how the final return value is extracted. Another challenge was extending the Verilog semantics with support for arrays, which is necessary to model hardware memory interfaces. Lastly, one particular difficulty that had to be overcome is proving that the function stack frame could be modelled by this finite Verilog array.

First mechanisation of general if-conversion CompCert does already perform limited if-conversion, removing branches that contain a single instruction and replacing them with a conditional move instruction, because predicated instructions are unsupported. I describe the formalisation of a general if-conversion transformation in CompCert used to generate hyperblocks, which are sequences of possibly branching predicated instructions, where the only incoming edges are to the start of the block. The key challenge was to generalise the if-conversion pass so that any external unverified heuristic could be used to inline blocks, while keeping the correctness proof conceptually simple. It is also flexible enough to allow for light loop transformations like loop unrolling and loop peeling.

Formal verification of hyperblock scheduling Next, I present a verified implementation of hyperblock scheduling, a critical optimisation for any HLS tool, taking advantage of the parallel nature of the hardware that is generated. I implement and validate the system of difference constraints (SDC) scheduling algorithm [Cong and

Zhang 2006], which is the base of the scheduling algorithm used by most HLS tools. Prior work verifying scheduling algorithms in CompCert [Six et al. 2022; Tristan and Leroy 2008] either were not general enough, being unable to schedule two branches of an if-statement together, or were inefficient because of unnecessary duplication when checking the correctness of the schedule. Instead, the key insight for this contribution is that predicates can be used to share otherwise duplicate expressions along different paths through the program.

SAT and SMT solvers for translation validation in CompCert I also present a novel use of a satisfiability (SAT) solver and a three-valued logic solver, implemented using an satisfiability modulo theories (SMT) solver, during translation validation to reason about the equivalence of functions before and after the scheduling transformations. The key challenge here is integrating an existing SMT proof checker, meant for assisting with interactive Coq proofs, and using it as a checker that can be used by the compiler at runtime to check that, for example, a three-valued predicate always holds. This can be used as a general validator for any transformation pass where the correctness depends on dynamically generated properties expressible in three-valued logic that need to be checked.

Evaluation of Vericert on PolyBench/C Finally, I evaluate different versions of Vericert against the state-of-the-art open source and unverified HLS tool Bambu HLS [Pilato and Ferrandi 2013] on a standard C benchmark suite called PolyBench/C. One might expect a fully verified tool to perform significantly worse than a more optimised, unverified tool, however, I show that Vericert produces designs that have around the same cycle count as Bambu without optimisations, with a slightly worse maximum clock speed, leading to an overall execution speed of around 1.6× that of Bambu designs. However, when comparing against optimised Bambu, Vericert is 3.6× slower, which can be explained by the extensive loop optimisations present in Bambu.

Companion material Vericert is fully open source and available on GitHub at:

`https://github.com/ymherklotz/vericert`

A snapshot of the Vericert development is also available in a Zenodo repository [Herklotz et al. 2024].

1.2 Dissertation Outline

This dissertation is organised into the following chapters.

Chapter 2 provides background for the rest of the dissertation and also discusses related work around verification of high-level synthesis.

Chapter 3 introduces Vericert itself, giving an overview of how it is structured, as well as what kind of transformations are performed. Design choices made during the development of Vericert are also described and compared against other possible approaches.

Chapter 4 then describes the Verilog semantics and the final correctness theorem of Vericert, which covers the main trusted computing base.

Chapter 5 then describes the front end of Vericert, which hooks into the middle end of CompCert. This chapter describes the implementation of hyperblock scheduling.

Chapter 6 then describes how the hyperblocks optimised by the scheduling algorithm are then turned into a hardware design in Verilog.

Chapter 7 evaluates different versions of Vericert in a number of ways on certain metrics comparing it against Bambu.

Chapter 8 finally gives a description of the limitations of Vericert as well as a discussion of the formalisation. In addition to that, many possible future directions are discussed.

1.3 Publications

The research in this dissertation has also been presented in the following three publications.

FCCM 2021 This first paper evaluates the reliability of HLS tools and motivates the need for a more reliable HLS tool, as well as a more robust verification flow for HLS designs. This paper is described in section 3.1.

Yann Herklotz, Zewei Du, Nadesh Ramanathan and John Wickerson. 2021a. 'An Empirical Study of the Reliability of High-Level Synthesis Tools'. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 219–223. DOI: 10.1109/FCCM51124.2021.00034.

OOPSLA 2021 Next, we introduce Vericert and describe an initial translation from C to Verilog using CompCert, without optimisations. This article is the basis for the dissertation, making up parts of chapters 3, 4 and 6.

Yann Herklotz, James D. Pollard, Nadesh Ramanathan and John Wickerson. Oct. 2021b. 'Formal Verification of High-Level Synthesis'. *Proceedings of the ACM on Programming Languages*, 5, OOPSLA, (Oct. 2021). DOI: 10.1145/3485494.

PLDI 2024 Finally, we describe a critical optimisation in the HLS flow called hyperblock scheduling. This article underpins chapter 5.

Yann Herklotz and John Wickerson. June 2024. 'Hyperblock Scheduling for Verified High-Level Synthesis'. *Proceedings of the ACM on Programming Languages*, 8, PLDI, Article 225, (June 2024), 25 pages. DOI: 10.1145/3656455.

The following publications did not directly contribute to the dissertation.

FPGA 2020 This paper describes bugs that were found in hardware synthesis tools by generating random, deterministic hardware designs.

Yann Herklotz and John Wickerson. 2020. 'Finding and Understanding Bugs in FPGA Synthesis Tools'. In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. Seaside, CA, USA. ISBN: 978-1-4503-7099-8. DOI: 10.1145/3373087.3375310.

FCCM 2022 This short paper describes an implementation of function calls in Vericert allowing function bodies to be shared between calls.

Michalis Pardalos, **Yann Herklotz** and John Wickerson. 2022a. 'Resource Sharing for Verified High-Level Synthesis'. In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1–6. DOI: 10.1109/FCCM53951.2022.9786208.

CPP 2023 Finally, this paper described an implementation of a control-flow based semantics for gated-SSA in CompCertSSA, a first step towards a pure data-flow intermediate language in CompCert.

Yann Herklotz, Delphine Demange and Sandrine Blazy. 2023. 'Mechanised Semantics for Gated Static Single Assignment'. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2023). Association

for Computing Machinery, Boston, MA, USA, 182–196. ISBN: 9798400700262. DOI: 10.1145/3573105.3575681.

2 Background

This chapter briefly describes field-programmable gate arrays (FPGAs) followed by introducing high-level synthesis (HLS) and the current state-of-the-art optimisations used by HLS tools, focusing in particular on static scheduling. Next, common testing and verification workflows for HLS are also described. Finally, an overview of CompCert is given, on which Vericert is built.

2.1 Field Programmable Gate Arrays

This section introduces field-programmable gate arrays (FPGAs), which is assumed to be the final target for the hardware produced by Vericert, as well as the HLS tools that Vericert is directly compared against.

FPGAs are programmable hardware chips that can be used to implement and run custom hardware without having to tape-out an ASIC, which may take years of development time. FPGAs instead provide a platform to test custom hardware quickly without these long turnaround times, and can be reprogrammed at will in case the hardware ever needs to change. Because they still allow for reprogrammability, they can never be as efficient as an equivalent ASIC design, however, for many applications having the chance to reprogram the hardware is an advantage. In addition to that, an FPGA will still generally be more efficient and more performant than running the same workload on a general-purpose processor. FPGAs comprise the following four main components [Boutros and Betz 2021], which are also shown in figure 2.1.

Look-up table (LUT) A LUT can implement any kind of logic with a set number of inputs and an output. On an FPGA, LUTs are often grouped into larger programmable logic units called *slices* that can handle multiple inputs and outputs. LUTs are also normally paired with an optional register at the output so that they can also be used as a memory.

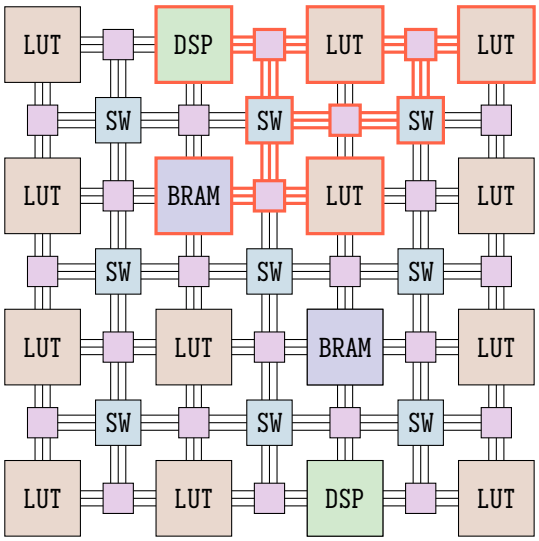


Figure 2.1: FPGA layout, showing an example of a design that is placed and routed on the FPGA highlighted in red. The beige blocks correspond to LUTs, followed by green blocks being DSPs and purple blocks being BRAMs. The programmable interconnects are made up of connection blocks and switches shown in pink and blue respectively.

Programmable interconnect The LUTs are connected using programmable interconnects, so that these arbitrary logical units can also be connected in arbitrary ways, making it possible to implement any kind of hardware design.

Block random-access memory (BRAM) Instead of relying on implementing memories to store a large amount of data using LUTs, there is often BRAM on the FPGA, which provides efficient storage for data.

Digital signal processor (DSP) Finally, FPGAs also often contain DSPs, which can be used to implement common arithmetic functions efficiently, that may otherwise take up a lot of space if implemented using LUTs. Some common arithmetic functions that are often implemented using DSPs include integer multipliers and multiply-accumulate operations.

The standard process to translate a hardware design from an HDL, such as Verilog or VHDL, to be placed onto an FPGA is to first *synthesise* the hardware design, which generates a lower level netlist description of the hardware in terms of the resources that are available on the FPGA. Next, the netlist is place-and-routed on the FPGA, which assigns a physical location to each resource and programs the interconnects so that all the components are connected properly. This low-level description of the hardware is then turned into a bit

stream that will program all the individual resources on the FPGA. The result can be seen in figure 2.1 by looking at the highlighted paths in red, as the place-and-route process placed logical functions into LUTs and connected them together making use of a BRAM and a DSP.

2.2 An Introduction to Verilog

This section will introduce Verilog for readers who may not be familiar with the language, concentrating on the features that are used in the output of Vericert. Verilog [IEEE 2006] is an HDL and is used to design hardware ranging from complete CPUs that are eventually produced as integrated circuits, to small application-specific accelerators that are placed on FPGAs. Verilog is a popular language because it allows for fine-grained control over the hardware, and also provides high-level constructs to simplify development.

Verilog behaves quite differently to standard software programming languages due to it having to express the parallel nature of hardware. The basic construct to achieve this is the always-block, which is a collection of assignments that are executed every time some event occurs. In the case of Vericert, this event is either a positive (rising) or a negative (falling) clock edge. All always-blocks triggering on the same event are executed in parallel. Always-blocks can also express control flow using if-statements and case-statements.

A simple state machine can be implemented as shown in figure 2.2. This state machine implements a pulse generator that generates a pulse every three clock cycles. At every positive edge of the clock (`clk`), the always-blocks will trigger. First, the `rst` input is checked, and if it is high the `state` variable is set back to `2'b0`. Otherwise, the state machine is implemented by checking the current value of `state` and defining the next `state`, including setting a value for the output `out`. No explicit value is set for the output in state `2'd1`, however, as `out` is a register, it will keep the same value from the last state, which in this case is `1'b0`. Assignments are performed using nonblocking assignment (`<=`), which assigns registers simultaneously at the end of each clock cycle. This state machine has the same general structure as the state machines discussed in this dissertation.

2.3 High-Level Synthesis

High-level synthesis is the transformation of software directly into hardware. There are many different types of HLS, which can vary in terms of the languages they accept or the

```
module counter(clk, rst, out);
  input clk, rst;
  output logic out = 1'b0;
  logic [1:0] state = 2'b0;

  always @(posedge clk)
    if (rst == 1'b1) state <= 2'b0;
    else
      case (state)
        2'd0: begin
          out <= 1'b0;
          state <= 2'd1;
        end
        2'd1: state <= 2'd2;
        2'd2: begin
          out <= 1'b1;
          state <= 2'd0;
        end
      endcase
endmodule
```

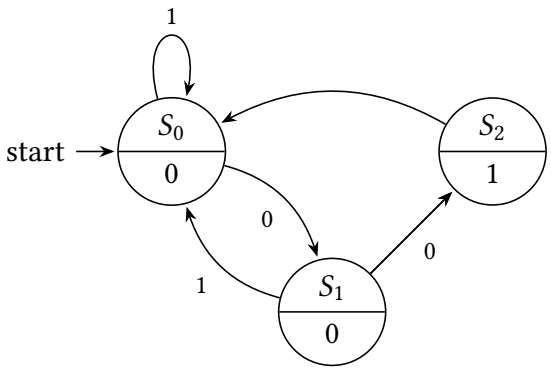


Figure 2.2: A simple pulse generator implemented using a state machine in Verilog, with its diagrammatic representation on the right. The only input to the state machine is the reset `rst`, and the output in each state is represented by the lower half of the nodes in the finite-state machine diagram.

devices that are targeted, however, they often share similar steps in how the translation is performed, as they all go from a higher level, behavioural description of the algorithm to a timed hardware description. In this dissertation, I will assume that I am targeting FPGAs instead of ASICs, which means there are often different constraints in terms of available resources, as well as clock frequency.

The main steps performed in the translation of an HLS tool is the following [Canis et al. 2013; Coussy et al. 2009]:

Compilation of input language First, the program or specification written in the input language to the HLS tool is compiled into an intermediate language that is more suitable to be transformed by optimisations. The input language for most traditional HLS tools is a restricted version of C or C++, however, HLS tools such as Google XLS [Google 2024] can use a domain-specific language (DSL) based on communicating sequential processes [Hoare 1978] as an input specification as well. There are also HLS tools based on OpenCL [Intel 2020b] or Matlab through the Simulink block diagram language [Constantinides et al. 2001; Ou and Prasanna 2005]. This specification is then turned into some intermediate language that is easier to

manipulate through optimisations and transformations to hardware. This includes intermediate languages such as the LLVM intermediate representation (IR) [Lattner and Adve 2004], MLIR [Lattner et al. 2021] or a custom representation of the code. The structure of these intermediate languages is further discussed in section 2.3.2.

Hardware resource allocation Depending on if the hardware target is a specific FPGA or an ASIC built on a specific technology library, the HLS tool will have to allocate resources differently. For example, on FPGAs there are only a limited number of LUTs, DSPs and BRAMs available. The HLS tool therefore often decides ahead of time which resources the program will need based on the operations that are present in the program. A few resources are often assumed to be infinite to simplify the resource allocation, examples being registers, which are normally cheap especially on FPGAs, or simple logic circuits that are cheap enough to be duplicated such as integer adders or multiplexers. Other circuits may require more resources in which case it would make sense to only have a few instantiations of the circuit and share it as much as possible. Examples of these circuits could be integer division modules or floating point arithmetic units that will most likely not have dedicated hardware on the FPGA and would therefore have to be implemented in logic. These circuits also often have trade-offs between area, latency and throughput which should be considered, and will be important in the operation scheduling step. Finally, memory and multiplication units lie in the middle, where there are usually enough BRAMs or DSPs resources on the FPGA, but they might introduce different challenges such leading to designs that are more difficult to place-and-route. In particular, if one uses too many of these resources, BRAMs and operations in the DSPs can be implemented using LUTs.

Operation scheduling Once the available resources have been chosen, the operations in the intermediate representation need to be scheduled into a clock cycle based on these resource constraints, creating a timed representation of the program. The goal of the operation scheduling step is to maximise the instruction-level parallelism of the program while also honouring the various resource constraints that are imposed by the available resources. Scheduling is further described in section 2.4. As part of the scheduling step, operations are also often preliminarily bound to specific resources.

Resource binding After scheduling, each operation is assigned a concrete instantiation of

its resource. For example, an integer divide operation will be assigned to the integer divider resource, and if this resource is used by another divider in the design, the inputs and outputs will have to be multiplexed. The resource constrained scheduling step should have ensured that two divisions are not taking place in the same cycle, and that the result of the division will only be used when the divider resource has finished computing the result.

Hardware description generation Finally, the hardware description is generated from the code that was described in the intermediate language and from the states and resources that each operation and register was assigned to.

There are many examples of existing high-level synthesis tools, the most popular ones being Bambu HLS [Pilato and Ferrandi 2013], LegUp [Canis et al. 2013], Vitis HLS [AMD 2023b], Catapult C [Siemens 2021], Google XLS [Google 2024] and Intel's OpenCL SDK [Intel 2020b]. These HLS tools all accept general programming languages such as C/C++ or OpenCL.

The concept of HLS has also evolved over time, from describing the transformation of the behavioural level of Verilog and VHDL to the register-transfer level in the 90s, to describing the transformation of C code into hardware automatically like tools do today, as synthesis tools have accepted increasingly larger subsets of Verilog and VHDL that include the behavioural level. In addition to that, languages like Bluespec [Nikhil 2004] are also considered HLS tools despite having different goals to more traditional HLS tools accepting C code. Bluespec provides a high-level hardware specification language that can be used to define concurrently running rules. These are automatically scheduled by the Bluespec synthesis tool and converted to a traditional synthesisable HDL. Handel-C [Aubury et al. 1996; Bowen 1998] is in a similar position, being a C-like language for hardware development. Handel-C and its translation to hardware is based on occam [Page and Luk 1991]. It supports many C features such as assignments, if-statements, loops, pointers and functions. In addition to these constructs, Handel-C also supports explicit concurrency as well as sequential or parallel assignments, similar to blocking and nonblocking assignments in Verilog. It therefore was popular as a hardware/software co-design language as the language could naturally be used to define sequential code running in software and parallel code which could be synthesised to hardware. However, it is inherently timed, and the Handel-C synthesis tool does not usually perform automatic parallelisation of the code as is the case with HLS tools that accept C as input.

In this dissertation I will be focusing on the more traditional HLS conversion from software languages into hardware designs.

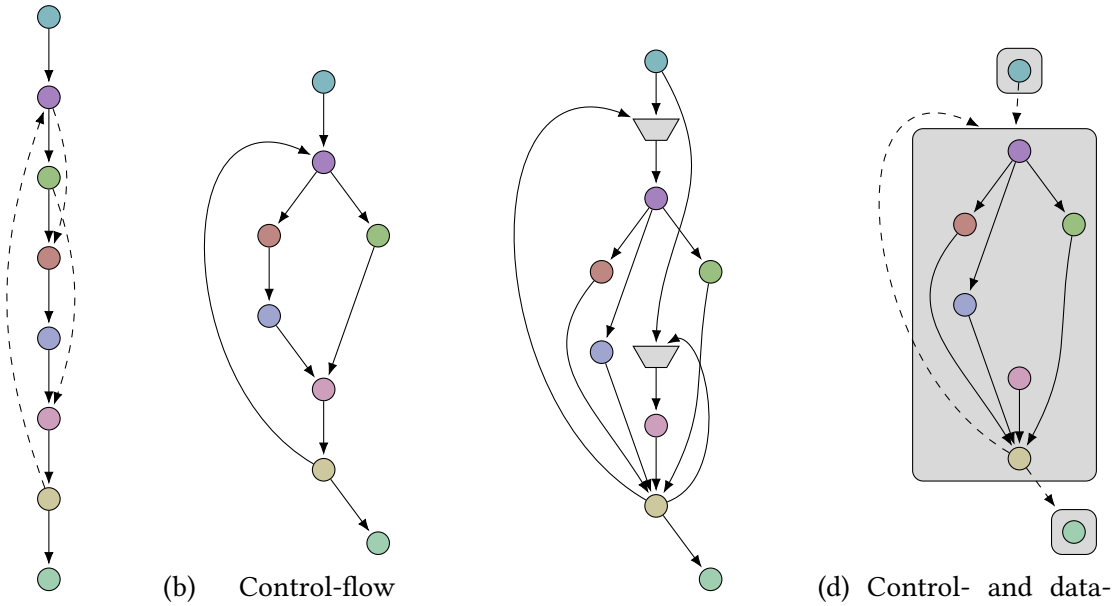
2.3.1 Data structures for intermediate languages

This section describes how intermediate languages in the HLS flow are usually represented and the differences in these approaches. Next, in section 2.3.2 I will describe techniques used to group instructions into a contiguous blocks to simplify analyses and transformations of instructions within the blocks.

There are many ways to represent the code that makes up a function or a program. High-level languages that are written by the programmer are normally represented as an abstract syntax tree (AST) in the front end of the compiler, which is a tree representing the parsed source file. An AST is a good representation for high-level optimisations that may need information about the intent of the programmer. An example being a loop which is represented as a structured for-loop instead of unstructured goto statements. On the opposite end, the assembly that will run on the processor can simply be represented by a list of instructions. This simple representation of the program is useful because individual instructions can directly be stored contiguously in memory and are then loaded by the processor to be executed. However, storing instructions as a list means that much of the original structure of the program is lost, which makes program analysis and transformations more difficult. In between these two representations, there is often one or more intermediate languages that can be represented in a variety of ways, and are either used purely for analysis or also as an intermediate transformation step.

Lists

An example of a list representing code is shown abstractly in figure 2.3a. In the figure, the solid arrows between nodes show how the instructions are stored as a linked list, one instruction feeding to the next. However, programs may have loops in them, which cannot directly be expressed in the list representation, as each instruction can only have one successor. Instead, instructions that are represented as a list will have labels attached to them, and goto instructions can then jump to those labels. These jumps are represented using the dashed arrows in the figure. This is the simplest representation of the code as it is completely linear, however, analysing such a program will be more difficult because a lot of information, like information on many of the edges, is implicit in the representation.



(a) List (b) Control-flow graph (CFG) (c) Data-flow graph (DFG) (d) Control- and data-flow graph (CDFG)

Figure 2.3: Comparison of lists, control-flow graphs, data-flow graphs and control- and data-flow graphs.

Every analysis pass would have to reconstruct the control-flow edges between the nodes, which would have to be stored as a graph, so lists are likely only the final representation of the code.

Control-flow graphs

A control-flow graph (CFG) representation of the code is a graph instead of a list, and allows any instruction to be connected to any other instruction explicitly by a control-flow edge. This signifies that after executing an instruction, the execution will then move to one of the successors of the current node. This turns control-flow analysis into a graph problem [Allen 1970], simplifying many program analysis passes and making it more natural to traverse the program with graph search algorithms. Figure 2.3b shows how the list representation of the code would be represented as a CFG, where nodes can now have multiple outgoing or incoming edges. Because the edges represent control flow, only one edge will be taken at a time as the program executes.

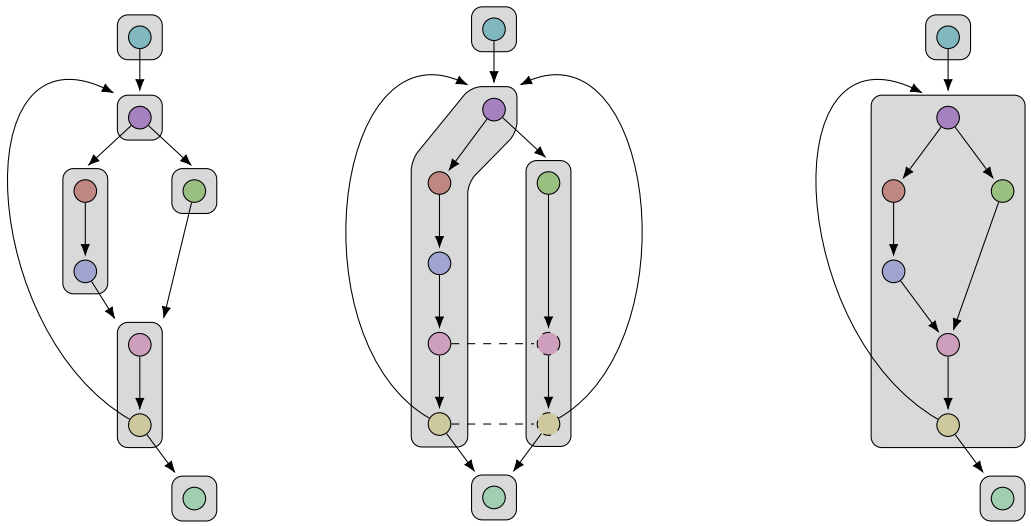
Data-flow graphs

Alternatively, instead of reasoning about the control flow of the program, one might be interested in the data flow of the program. Many compiler optimisations, such as dead-code elimination or constant propagation, need to perform data-flow analyses on the CFG [Kam and Ullman 1976; Kildall 1973]. In addition to that, hardware circuits are naturally expressed using pure data flow as netlists can be viewed as data flow graphs, and even HDLs can be modelled by synchronous data flow programming languages [Halbwachs et al. 1991]. One could therefore represent the code as a pure data-flow graph (DFG) to help with data-flow optimisations as well as the translation to the final hardware. An example of a DFG that is be compatible with the CFG shown in figure 2.3b is represented in figure 2.3c, where arrows now represent data dependencies between nodes instead of control-flow dependencies.

As can be observed in the diagram, the edges between the nodes are very different to the edges in the CFG. Nodes in the CFG may not have *needed* to be in that particular order, because when two instructions are independent, one still needs to define an order between them. This is necessary due to the sequential nature of the CFG, instead, the DFG only represents the necessary edges between the nodes. Additionally, we have added two additional nodes to handle the back edges of the loop, which are also data dependencies. These nodes are represented by a trapezium in the DFG and act as loop headers to control the loop iterations when interpreting the graph as pure data flow. These additional nodes are needed because otherwise the data dependencies due to the back edges of the loop would conflict with the data dependency that actually enters the loop. We therefore need an additional node to turn the data dependencies into the correct control-dependencies, allowing data into the loop when the loop is not executing, but while it is executing the node should only allow data through from the back edge of the loop. The example shown in figure 2.3c is similar to the concepts of μ-nodes in gated static single assignment form and the program dependence web [Campbell et al. 1993; Havlak 1994; Ottenstein et al. 1990; Tu and Padua 1995], which are intermediate languages that can be interpreted in a data flow oriented context.

Control- and data-flow graphs

As mentioned in the previous section, loops are especially problematic for pure DFGs. Instead, one can try to get the best of both CFGs and DFGs by creating a control- and data-flow graph (CDFG). In a CDFG, one instead has a CFG where each node is a DFG



(a) Basic blocks (b) Superblocks (c) Hyperblocks

Figure 2.4: Comparison of basic blocks, superblocks and hyperblocks.

instead of just an instruction. The example is shown in figure 2.3d, where the dashed arrows are control dependencies between DFGs, and the solid lines are the data dependencies within the DFG. In this way, loops can be supported without having to introduce special nodes, leaving the loop back edge as a control dependency instead. DFGs can represent the sections of code without arbitrary external incoming control-flow edges using data dependencies, thereby being more flexible in how instructions are rearranged without the downside of having to introduce additional nodes.

2.3.2 Grouping instructions into blocks

This section describes the representation of intermediate languages that are often used within an HLS tool. In particular, we will base our intermediate language on a CFG, but will describe various ways to group blocks of instructions together and describe the advantages and drawbacks that they provide. Especially when considering compiler targets that allow for instruction-level parallelism, such as very large instruction word (VLIW) processors or custom hardware, it is important to schedule instructions so that the parallelism can be exploited. Various ways in which instructions can be grouped affect the size of the regions of instructions that can be scheduled together, and therefore affect the performance that can be achieved [Faraboschi et al. 2001], due to how much reordering the scheduler can perform.

Basic blocks

To build a bit more structure in the CFG, it is useful to group non-branching instructions without any external incoming edges together forming basic blocks. An example of the CFG segmented into basic blocks is shown in figure 2.4a. Instructions within a basic block can then be represented as a list of instructions, as there cannot be any branches, and these lists of instructions can then be safely manipulated because there is a guarantee that there is no external incoming control flow in the middle of a basic block. For example, as long as two instructions in a basic block are independent, they can be safely reordered. This would have otherwise not been possible, because external incoming control flow may mean that reordering the instructions now leads to the unintended execution of an instruction. Basic blocks are used by IRs in real compilers such as Gimple in GCC or LLVM IR in clang as the default grouping of sequential instructions. They are a useful representation for various optimisations and transformations, because the CFG can be viewed as being coarser grained when manipulating control flow, because one does not have to inspect the control flow within basic blocks, and provides useful assumptions that can be made when performing intra-block transformations.

Superblocks

As figure 2.4a shows, one issue with basic blocks is that they are often quite small, therefore limiting the benefits that they can provide. One extension to basic blocks is superblocks [Hwu et al. 1993], shown in figure 2.4b. Superblocks extend the notion of basic blocks to contiguous regions without arbitrary incoming control-flow edges, but with multiple control-flow edges out of the superblock. The main benefit of this is that due to the extra flexibility of the multiple exits, superblocks can contain arbitrary linear traces through regions of non-looping code. The 'hot path' of the loop body can be grouped into a single superblock for example, which can then be heavily optimised. Various other important paths through the program can then also be grouped and optimised together. If different paths through the program reuse nodes from another path, as is shown with the pink and yellow nodes in figure 2.4b, then these nodes will have to be duplicated so that they can be included in a different path. As long as any optimisation takes into account the arbitrary exits that are possible in superblocks, the nodes within a superblock can be optimised independently from the rest of the code.

Unlike basic blocks, superblocks are not generally used to represent the IR itself, and they are mainly used to perform superblock scheduling. GCC, for example, implements a

superblock scheduling optimisations, but does not use the superblock representation for other transformations passes.

Hyperblocks

One downside of superblocks is that they cannot represent branching and joining control flow within a single block. For example, the block present in the CDFG shown in figure 2.3d could not be represented by a superblock. Hyperblocks are defined as basic blocks of predicated instructions with arbitrary outgoing edges leading to the next hyperblock [Mahlke et al. 1992]. They can represent arbitrary branching and joining control flow without arbitrary external incoming control-flow edges, and are therefore an extension to superblocks. This means that they could represent the CDFG shown in figure 2.3d, and an example of a hyperblock is shown in figure 2.4c, where in practice code within the hyperblock has been linearised using predicated instructions, instead of being represented as a graph. This leads to possibly more complex control flow than in both of the previous cases, however, it can be reasoned with using a SAT or SMT solver as the path information is part of the predicates.

Superblocks may require many blocks to be duplicated to group all the wanted instructions together. However, with predicated instructions, hyperblocks can represent joining control flow without having to duplicate any blocks because predicates can be selected so that the block is executed when either incoming edge is taken. Hyperblocks are not common in regular compilers, because predicated execution is rare in modern instruction set architectures. For example, Arm recently removed most conditionally executed instructions from their A64 ARMv8 ISA, because ‘predicated execution of instructions does not offer sufficient benefit to justify its [...] implementation cost in advanced implementations’ (Arm [2011, sec 3.2, p. 10]). However, hyperblock scheduling has been a popular intermediate language for HLS [Budiu and Goldstein 2002; Callahan and Wawrzynek 1998], as the flexibility of targeting hardware directly means that predication can be implemented efficiently.

On the other hand, GPUs often have sophisticated support for predication so that the same sequence of instructions (also called warps) can be executed on multiple different threads on the GPU, and so that diverging control flow can still be handled by a single, large warp. This can be achieved by designing predicates based on the thread identifier [Narasiman et al. 2011] and instructions operating on such predicates can be found in the SPIR-V specification [Kessenich et al. 2018, p. 172]. Trace scheduling methods that are aware of

warps produced by the GPU have therefore been developed, however, if-conversion is only performed after scheduling instead of using a scheduling algorithm that is aware of predication. 'Warp-Aware Trace Scheduling could benefit from trace-enlarging techniques, like Superblock scheduling's tail duplication and Hyperblock scheduling's predication' (Jablin et al. [2014]).

2.4 Scheduling

Instruction scheduling is the main transformation and optimisation performed by traditional HLS tools. The scheduling transformation introduces time into the untimed input representation by placing each instruction into a clock cycle in which it should execute. The scheduler must take into account the resources that were selected during the resource allocation step and schedule the instructions so that they meet certain constraints. In this section I will discuss scheduling techniques used by HLS tools, first discussing static scheduling in section 2.4.1, which is the scheduling algorithm used by Vericert in this dissertation, followed by describing dynamic scheduling in section 2.4.2, which is an alternative scheduling technique with different trade-offs to static scheduling.

2.4.1 Static scheduling

Static scheduling is used by the majority of synthesis tools [AMD 2023b; Canis et al. 2013; Intel 2020b; Roane 2023; Siemens 2021] and means that the time at which each operation will execute is known at compile time. The first step is to generate a CDFG, which is either already the structure of the code, or can be generated from a hyperblock CFG representation, for example, by using static analysis on each hyperblock to gather all data dependencies and convert them into DFGs.

Scheduling can have different optimisation goals for the design in terms of latency and resource usage. We will assume that scheduling is performed independently on blocks of code without back edges, i.e. on the DFG blocks in a CDFG like the one shown in figure 2.3d. Within these blocks, each operation in the DFG can be scheduled as soon as its predecessor in the graph has been scheduled, resulting in an as soon as possible (ASAP) schedule, where the start time of each node corresponds to the longest path from the start of the DFG to that node. If instead the start time of a node is taken to be the longest path from the node to the end of the DFG, then this would result in an as late as possible (ALAP) schedule. The difference between these two types of schedules shows the

slack of a particular operation, which can be exploited by schedulers to minimise resources while also minimising latency.

A more advanced scheduler can either optimise for resource usage based on a maximum latency or for latency based on a maximum amount of resources. In general this is an NP-hard problem, however, *list scheduling* [Baker 2019, p. 257] provides a heuristic for this problem by ordering nodes that can be scheduled according to a priority function and picking a subset of these nodes to actually be scheduled as long as the resources used by the subset is not greater than the available resources. Another alternative scheduling algorithm initially used to perform synthesis of behavioural hardware description languages is called force-directed scheduling [Paulin and Knight 1989], which is tasked to minimise the number of functional units, memory and buses by balancing operations associated with these resources. The intuition is that this balancing creates a schedule with high utilisation and therefore low resource usage, without sacrificing on overall execution time of the design. It works by iteratively calculating forces associated with each node and scheduling the node with the lowest force.

System of difference constraints scheduling

The main static scheduling algorithm used by HLS tools is the system of difference constraints (SDC) scheduling algorithm [Cong and Zhang 2006]. It generates an SDC that is a subset of a linear programming (LP) problem that can be incrementally modified and checked for feasibility as new constraints are added. Then, to solve the SDC it can be converted into an LP problem guaranteeing integer solutions.

The scheduling algorithm is built on the notion of constraining scheduling variables for an operation v ($\mathrm{sv}_i(v)$). Each operation is associated with a set of scheduling variables, but at a minimum it must have an initial and a final scheduling variable ($\mathrm{sv}_{\mathrm{init}}(v)$ and $\mathrm{sv}_{\mathrm{fin}}(v)$). The main advantage of this scheduling algorithm is that one can define different concepts as constraints in terms of scheduling variables. For example, if one has a data dependency from operation a to operation b, the following constraint is added to the SDC.

$$\mathrm{sv}_{\mathrm{fin}}(a) - \mathrm{sv}_{\mathrm{init}}(b) \leq 0 \tag{2.1}$$

Other types of constraints that are normally added to get a valid schedule are:

Control dependency constraint Constraint between blocks of instructions that are connected by a control-flow dependency. This ensures that a block is not scheduled

before another block it depends on. In particular, loop back edges therefore need to be removed during the SDC construction, as otherwise the system would become unsatisfiable.

Relative timing constraint This constraint ensures that two operations are separated by at least or at most a fixed number of cycles. This can be used to satisfy I/O timings.

Latency constraint This constraint specifies a maximum latency for a set of blocks.

Cycle time constraint This constraint is used to target a specific final clock period, and split up long combinational paths through the DFG into separate cycles. This allows for *operation chaining*, whereby multiple operations that are estimated to have a latency less than a clock cycle can be chained within the same clock cycle.

Resource constraint To handle limited resources, constraints can be used to make it infeasible to have operations use the same resource within the same clock cycle. Constructing these constraints relies on a linear ordering of the DFG, so some combination of ALAP or ASAP scheduling can be done to form the linear ordering before the actual scheduling step.

Solving the SDC with different optimisation functions produces different kinds of schedules. For example, optimising the following function will produce an ALAP schedule, as the start time of every operation is maximised.

$$\max \sum_{v \in V_{\text{op}}} \text{sv}_{\text{init}}(v) \tag{2.2}$$

Modern HLS tools often use a variation of SDC scheduling. For example, Vitis HLS and LegUp use an extension of SDC scheduling [Canis et al. 2014; Zhang and Liu 2013] with support for modulo scheduling (loop pipelining) [Rau 1996] and Bambu HLS uses an extension of SDC scheduling with support for speculative execution and loop code motion [Lattuada and Ferrandi 2015]. In this dissertation, SDC scheduling refers to the initial implementation of the scheduling without any extensions.

2.4.2 Dynamic scheduling

On the other hand, dynamic scheduling [Josipović et al. 2018] does not require the schedule to be known at compile time and instead it generates latency insensitive circuits [Carloni

et al. 2001], that use tokens to schedule operations at runtime. Whenever the data for an operation is available, it sends a token to the next operation, signalling that the data is ready to be read. The next operation does not start until all the required inputs to the operation are available, and once that is the case, it computes the result and then sends a token declaring that the result of that operation is also ready. The benefit of this approach is that only basic data-flow analysis is needed to connect the tokens correctly, however, the scheduling is done dynamically at run time, depending on how long each primitive takes to finish and when the tokens activate the next operations.

The benefit of this approach over static scheduling is that the latency of these circuits is normally significantly lower than the latency of static scheduled circuits, because they can take advantage of runtime information of the circuit. However, because of the signalling required to perform the runtime scheduling, the area of these circuits is usually much larger than the area of static scheduled circuits. In addition to that, much more analysis is needed to properly parallelise loads and stores to prevent bugs, which requires the addition of buffers in certain locations.

An example of a dynamically scheduled synthesis tool is Dynamatic [Josipović et al. 2018], which uses a load-store queue (LSQ) [Josipović et al. 2017] to order memory operations correctly even when loops are pipelined and there are dependencies between iterations. In addition to that, performance of the dynamically scheduled code is improved by careful buffer placement [Josipović et al. 2021], which allows for better parallelisation and pipelining of loops. Handel-C and the translation of occam into circuits are also examples of dynamic scheduling, however, less optimisations like automatic buffer placement have been explored.

2.5 Verification

Theorem provers can be categorised into two main types: automatic theorem provers described in section 2.5.1 and interactive theorem provers described in section 2.5.2. This section will give a brief overview of the characteristics of these different verification tools.

2.5.1 Automatic theorem provers

Automatic theorem provers are tools that can reason about logic automatically, and answer whether a theorem is true or whether there exists a counter example. Most automatic theorem provers are implemented around SAT or SMT solvers, which can be characterised

as deciding if a formula is satisfiable (or not) by finding an assignment that satisfies the formula. These solvers can also be used to used to prove that a theorem holds by showing that the negation of the theorem is unsatisfiable, which implies that the theorem must hold for all assignments. In SAT, the formula is purely boolean, but in SMT the boolean formula may include arbitrary theories that extend the logic, for example by including linear integer arithmetic or a theory of arrays that may have specialised solvers for the theory. This widens the space of problem that can be efficiently encoded in the logic by making use of specialised solvers for different domains, as opposed to encoding the problem using boolean logic, which quickly becomes infeasible due to the exponential growth of the formulas. Furthermore, SMT solvers such as Z3 [Moura and Bjørner 2008], cvc5 [Barbosa et al. 2022], Boolector [Brummayer and Biere 2009] or veriT [Bouton et al. 2009] can be used by higher level automatic verification tools such as bounded model checkers like CBMC [Kroening and Tautschnig 2014] or verification aware programming languages like Dafny [Leino 2010].

Automatic theorem provers are also the basis for formal hardware verification and equivalence checking, because the formal properties that need to be proven, as well as the hardware design itself, can be encoded in SMT. Symbiyosys [YosysHQ 2023], an open source formal verification for hardware, does exactly this by synthesising the hardware design including its assertions to SMT-LIB2 [Barrett et al. 2017], a standard input language to SMT solvers. If the SMT solver can show that the the assertions are unsatisfiable, this implies that the properties in the hardware will always hold. Commercial verification tools like Cadence Conformal [Cadence 2023a] and Synopsys VC Formal [Synopsys 2023] use similar techniques, but provide more complex solver orchestration to enable larger scale automatic verification of formal properties [Koelbl et al. 2009]. More details about current hardware verification methodologies are given in section 2.6.1, especially as they apply to HLS.

How can one trust the answer of such automatic theorem provers? From a high-level, they are running an opaque, highly optimised algorithm and return an answer to the problem, which may be incorrect. If the formula is satisfiable, the solution is simple: the solver can provide a model that satisfies the formula. Checking if the solver gave the right answer is just a matter of checking that the model satisfies the formula. If the formula is unsatisfiable, the solver will return ‘unsat’ as the answer as there is no model. Checking that a formula is actually unsatisfiable without having to trust the SMT solver can be done if the SMT solver can generate a proof witness, which can be checked by an independent,

trusted checker. Proof witnesses are normally a series of rewrites of axioms or basic lemmas that translate the original formula into *false*. If one trusts the checker, one does not have to trust the SMT solver, because if it can generate a valid proof witness, then one knows that the formula has to be equivalent to *false*. Some examples of SMT solvers that can generate proof witnesses are veriT [Bouton et al. 2009] or cvc5 [Barbosa et al. 2022], and the proof witnesses generated by these tools can actually be checked by a formally verified proof checker called SMTCoq [Armand et al. 2011], so that these proofs of SMT formulas can be used in a verified context.

Unfortunately, equivalence checkers used to verify designs or formal verification tools to prove assertions about hardware normally do not produce a proof witness that can be independently checked. This means that especially with the commercial, closed source formal verification tools, one has to trust that the result they produce is correct. Additionally, when automatic verification tools cannot prove the equivalence between two designs, for example, it is often unclear what additional information the checker needs to complete the proof. Instead one often has to guess where the solver is getting stuck, and provide assertions to guide it in the right direction.

2.5.2 Interactive theorem provers

Interactive theorem provers like Coq [Bertot and Castéran 2004], Isabelle [Paulson 1994] or Lean [Moura et al. 2015], on the other hand, focus on checking proofs that are provided to them, instead of automatically trying to prove theorems. These proofs can either be written manually by the user in a tactic language provided by the theorem prover, or automatically generated by external decision procedures, such as an automatic theorem prover that produces a witness. One benefit of using an interactive theorem prover is that the proof is checked by a small, trusted kernel.

Interactive theorem provers can help with verifying complex theorems, such as proving the correctness of a C Compiler, as is the case with CompCert, or prove the correctness of an operating system kernel like sel4 [Klein et al. 2009] or the four colour theorem proof [Gonthier 2008]. Interactive theorem provers are more suited to large scale verification projects compared to pure automatic theorem provers because they often combine the benefits of automation provide by automatic theorem provers, and the more granular manipulation of proofs provided by the interactive theorem prover itself as the tactic language, while having a small trusted computing base. Proofs can be developed by witnesses produced using automated solvers when the problem is small enough to be tractable, and

they can then be stitched together to prove more difficult theorems that could not have been proven automatically.

In practice, formalising projects in an interactive theorem prover leads to the formalisation of a lot of technical detail which may not have been present in the paper formalisation. One has to be precise when describing the behaviour of a system, and these details will have to be reasoned about in all proofs. It is therefore important to design abstractions carefully so that proofs can be reused, and so that proofs can be modified if definitions are tweaked.

More details on how an interactive theorem prover is used to develop verified programs is described in sections 2.6.2 and 2.7.

2.6 Verification of High-Level Synthesis

This section describes current ways in which designs generated by HLS tools are validated. First, I will describe unmechanised verification of HLS tools, meaning testing and verification methodologies that have not been formalised in a theorem prover. Next, I will describe mechanised verification around HLS.

A summary of the related works can be found in figure 2.5, which is represented as an Euler diagram. The categories chosen for the Euler diagram are: whether the tool is usable, whether it takes a high-level software language as input, whether it has a correctness proof, and finally whether that proof is mechanised. The goal of Vericert is to cover all of these categories.

Most practical HLS tools [AMD 2023b; Canis et al. 2013; Intel 2020b; Nigam et al. 2020] fit into the category of usable tools that take high-level inputs. On the other end of the spectrum, there are tools such as BEDROC [Chapman et al. 1992] for which there is no practical tool, and even though it is described as high-level synthesis, it more closely resembles today's logic synthesis tools.

Ongoing work in translation validation [Pnueli et al. 1998] seeks to prove equivalence between the hardware generated by an HLS tool and the original behavioural description in C. An example of a tool that implements this is Siemens's Catapult [Siemens 2021], which tries to match the states in the hardware description to states in the original C code after an unverified translation. Using translation validation is quite effective for verifying complex optimisations such as scheduling [Chouksey and Karfa 2020; Karfa et al. 2006; Youngsik Kim et al. 2004] or code motion [Banerjee et al. 2014; Chouksey et al. 2019], but

the validation has to be run every time the HLS is performed. In addition to that, the proofs are often not mechanised or directly related to the actual implementation, meaning the verifying algorithm might be wrong and hence could give false positives or false negatives.

Finally, there are a few relevant mechanically verified tools. First, Kôika is a formally verified translator from a core fragment of Bluespec into a circuit representation which can then be printed as a Verilog design. This is a translation from a high-level hardware description language into an equivalent circuit representation, so is a different approach to HLS. Lööw and Myreen [2019] used a proof-producing translator from HOL4 code describing state transitions into Verilog to design a verified processor, which is described further by Lööw et al. [2019]. Lööw [2021] has also worked on formally verifying a logic synthesis tool that can transform hardware descriptions into low-level netlists. This synthesis back end can seamlessly integrate with the proof-producing HOL4 to Verilog translator as it is based on the same Verilog semantics, and therefore creates verified translation from HOL4 circuit descriptions to synthesised Verilog netlists. Perna and Woodcock designed a formally verified translator from a deep embedding of Handel-C into a deep embedding of a circuit [Perna and Woodcock 2012; Perna et al. 2011]. Finally, Ellis [2008] used Isabelle to implement and reason about intermediate languages for software/hardware compilation, where parts could be implemented in hardware and the correctness could still be shown.

2.6.1 Unmechanised verification of HLS

This section describes how HLS designs are typically verified and describes the current state-of-the-art in verification of HLS transformations. The standard way to test designs in HLS is either using hardware test benches, testing the final hardware design like any other hardware design, or by using C/Verilog co-simulation, which is a feature some HLS tools support, whereby the same test code that drove the C code can also be used to drive the hardware design. Metrics such as code coverage or functional coverage can be gathered to check which parts or behaviours of the design were exercised, however, these coverage metrics generally do not imply full correctness of the design. Functional coverage is especially interesting, as the verification engineer defines how much of the design should be tested. If the behaviour that needs to be covered is too large, testing will not be able to achieve full coverage of the entire design. On the other hand, if the specified behaviour to be tested is quickly saturated, it is likely that many behaviours will remain untested.

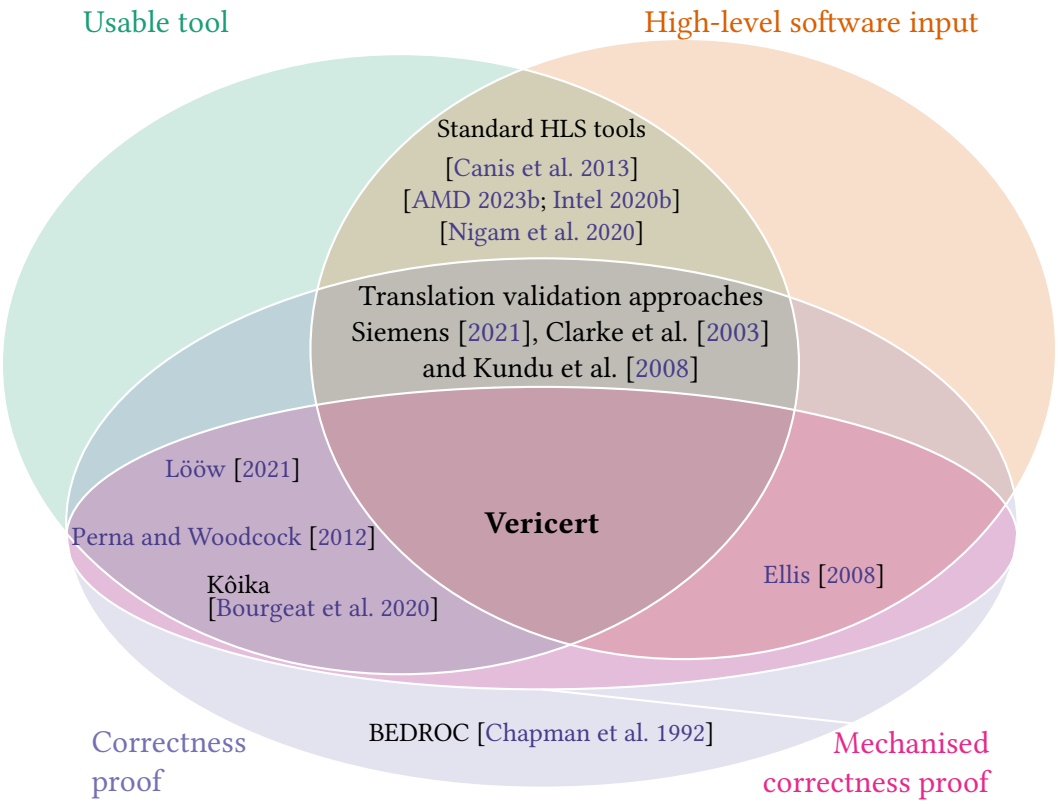


Figure 2.5: Summary of related work.

Translation Validation

Instead, the equivalence between the higher-level input code and the hardware design can sometimes be proven automatically. There are commercial tools that claim to support such equivalence checks, such as Siemens's SLEC [Chauhan 2020], Cadence's C2RTL [Cadence 2023b] or Synopsys's HECTOR [Koelbl et al. 2021], which try to match the states in the register-transfer level description to states in the original C code after an unverified translation. However, in general this does not scale to large designs because of the explosion in the size of the state and the semantic gap between the software model and the hardware design, leading to various ways in which the equivalence checking problems needs to be partitioned to be tractable, which may not always succeed automatically. This leads to manual proof engineering work that is required to reduce the size of the equivalence checking problem, leading to the same issues that were discussed in section 2.5.1, such as having to guess where the solver is getting stuck.

This technique is called translation validation [Pnueli et al. 1998], whereby the translation that the HLS tool performed is proven to have been correct for that specific input by implementing a validator that decides if the input and output of the transformation are equivalent. In addition to the commercial tools mentioned previously, there has been research on verifying specific HLS optimisations correct using translation validation as well, such as scheduling [Chouksey and Karfa 2020; Karfa et al. 2006; Youngsik Kim et al. 2004] or code motion [Banerjee et al. 2014; Chouksey et al. 2019]. Even though many of these papers are accompanied by proofs of correctness, these proofs are often not mechanised or directly related to the actual implementation, meaning the verifying algorithm might result in false positives.

More examples of translation validation for proofs about HLS algorithms [Chouksey and Karfa 2020; Chouksey et al. 2019; Karfa et al. 2012, 2006, 2010, 2008, 2007; Kundu et al. 2007, 2008] are performed using an HLS tool called SPARK [Gupta et al. 2003]. These translation validation algorithms can check the correctness of complicated optimisations such as code motion or loop inversions. These transformations are all verified as transformations on the SPARK IR. In general, these validation algorithms work by constructing a finite-state machine with data path (FSMD) [Hwang et al. 1999] representation, then partition the states and match them up using cut points. Finally, symbolic expressions are constructed for paths between cut points and are checked to be equivalent between the original and transformed design.

Even though the correctness of the verifier is proven correct in the papers, the algorithm

implementing the verifier might still include bugs. It is therefore possible that output is accepted even though it is not equivalent to the input. In the case of SPARK, there are no validation checks for the translation from C to the internal IR, as well as no checks for the final translation from the IR to Verilog. In addition to that, using translation validation to prove the correctness of the complete HLS translation either means that one needs an algorithm that verifies all transformations at once, or, like it is the case in SPARK, describes separate translation validation algorithms for each individual translation. The former means that the proof of correctness about the validator will be the proof of correctness of the entire translation, however, in practice, writing such an algorithm and verifying it is not possible, as the difference between the structure of the input and output will be too great and unpredictable. Instead, translation validation passes are composed with each other, however, this requires that one trusts that the composition of different validation algorithms, with different formulations of their respective soundness criteria, is itself sound.

Direct Unmechanised Proofs

There has also been work proving HLS correct without using translation validation, but by showing that the translation itself is correct. The first instance of this is proving the BEDROC [Chapman et al. 1992] HLS tool is correct. This HLS tool converts a high-level description of an algorithm, supporting loops and conditional statements, to a netlist and proves that the output is correct. It works in two stages, first generating a DFG from the input language, HardwarePal, then optimising the DFG to improve the routing of the design and the schedule of the operations. Finally, the netlist is generated from the DFG by placing all the operations that do not depend on each other into the same clock cycle. Data path and register allocation is performed by an unverified clique partitioning algorithm. The equivalence proof between the DFG and HardwarePal is done by a proof by simulation, where it is proven that, given a valid input configuration, applying a translation or optimisation rule will result in a valid DFG with the same behaviour as the input.

There has also been work on proving the translation from occam to gates [Page and Luk 1991] correctly using algebraic proofs [Jifeng et al. 1993]. This translation resembles dynamic scheduling as tokens are used to start the next operations. A mechanised translation from Handel-C into a netlist is described further in section 2.6.2.

2.6.2 Mechanised compiler proofs in high-level hardware design

Even though a proof for the correctness of an algorithm might exist, this does not guarantee that the algorithm itself behaves in the same way as the assumed algorithm in the proof. The specification of the algorithm is separate from the actual implementation, meaning there could be various implementation bugs in the algorithm that cause it to behave incorrectly. C compilers are a good example of this, where a few optimisations performed by the compilers have been proven correct, however these proofs are not linked directly to the actual implementations of these algorithms in GCC or Clang. Additionally, with the exception of a few proofs such as the proof of correctness of lazy code motion [Knoop et al. 1994], most proofs reference simpler versions of optimisations than the ones that are eventually implemented in the compiler. Yang et al. [2011] found more than 300 bugs in GCC and Clang, many of them appearing in the optimisation phases of the compiler. One way to link the proofs to the actual implementations in these compilers is to write the compiler in a language which allows for a theorem prover to check properties about the algorithms. Yang et al. found that CompCert only had five bugs in all the unverified parts of the compiler, meaning this method of proving algorithms correct provides great confidence that the compiler is correct.

This section explores formalisation of hardware design in general, focusing specifically on higher level hardware design, by describing formalisations of higher level hardware description languages that are synthesised to a lower level netlist representation of the hardware design.

The first mechanisation of a scheduling algorithm for the translation of a higher level specification to a hardware design is presented by Anderson and Ainscough [1994]. This formalisation is written in HOL, and describes a scheduler that is proven to obey control and data dependencies, and implements an ASAP schedule.

Perna and Woodcock [2012] developed a mechanically verified Handel-C to netlist translation written in HOL. The translation is based on previous work describing translation from occam to gates by Page and Luk [1991], which was proven correct by Jifeng et al. [1993] using algebraic proofs. As Handel-C is an extension of occam with C-like operations, the translation from Handel-C to gates can proceed in a similar way.

Perna and Woodcock mechanise the compilation of a subset of Handel-C to gates, which does not include memory, arrays or function calls. In addition to the constructs presented by Page and Luk, the prioritised choice construct is also added to the Handel-C subset that is supported. The verification proceeds by first defining the algorithm to perform

the compilation, chaining operations together with start and end signals that determine the next construct which will be executed. The circuits themselves are treated as black boxes by the compilation algorithm and are chosen based on the current statement in Handel-C which is being translated. One interesting property about these circuits is that the control signal is propagated correctly through the circuit. This is the property that is mechanised in the work for the translation of Handel-C to netlists. However, the final semantic equivalence of the translation was not mechanised.

Next, Fe-Si [Braibant and Chlipala 2013], Kami [Choi et al. 2017] and Kôika [Bourgeat et al. 2020] are all formalisations of derivatives of Bluespec, which popularised rule-based hardware design whereby rules can be written and reasoned about independently, but can be scheduled in parallel. Fe-Si focuses on formalising the translation of a Bluespec derivative to Verilog. Kami, on the other hand, focuses on the modular verification of hardware designs written in the rule-based hardware design language. Finally, Kôika provides one-rule-at-a-time reasoning, modularising proofs over rules as well. Kôika also provides a mechanised compilation from the collection of rules with a schedule into an equivalent circuit. This circuit can then be pretty-printed as a Verilog design. These are fundamentally different approaches to increasing the abstraction level of hardware design compared to translating software into hardware. This mainly comes down to Kôika giving fine-grained control over the intra-cycle scheduling of rules and both Kôika and Bluespec give control of the inter-block scheduling which is up to the hardware designer, whereas in traditional HLS the hardware design is automatically scheduled from the sequential software definition.

Finally, another related formalisation is Lutsig, a formally verified Verilog synthesis tool [Lööw 2021]. This is concerned with translating register-transfer level Verilog into netlist level Verilog, instead of translating from a higher level specification into hardware.

2.6.3 HLS formalised in Isabelle

Martin Ellis' work on the specification of hardware/software co-design [Ellis 2008] is the first attempt towards mechanically verified HLS using Isabelle. The main goal of the thesis is to provide a framework to prove hardware/software co-design compilers correct, where part of the design is specified in software and other parts of the design are translated to equivalent hardware to be accelerated. The dissertation describes the semantics of a static single assignment (SSA) based software IR which supports partitioning of code into hardware and software parts, as well as a custom netlist format which is used to

describe the hardware parts and is the final target of the hardware/software compiler. The dissertation then describes what the correctness would look like between the software and hardware semantics. The framework used to prove the correctness of the compilation from the IR to the netlist format is formalised in Isabelle.

As both the input IR and output netlist format have been designed from scratch, there is neither a description of a translation from a higher level languages into the SSA form, nor a description on how to map the netlist language onto an FPGA. The main goal of the thesis is to describe how correctness between the two languages could be stated and explores how the software language semantics can interact with the hardware semantics.

Finally, it is unclear whether or not a translation algorithm from the IR to the netlist format was implemented, as the only example in the thesis seems to be compiled by hand to explain the correctness theorem with respect to that example. There are also no benchmarks on real input programs showing the efficiency of the translation algorithm, and it is therefore unclear whether the framework would be able to prove more complicated optimisations that a compiler might perform on the source code. The dissertation is likely assuming that high-level programs and their netlist implementations are verified according to the correctness property manually for each design, as there is no implementation or proof of an automatic translation from the software code into the mixed software/hardware design.

2.7 CompCert

CompCert [Leroy 2006, 2009b; Leroy et al. 2016] is a formally verified C compiler written in Coq [Bertot and Castéran 2004]. The verified compiler in Coq is extracted to OCaml code and can then be used. CompCert comprises eleven intermediate languages, which are used to gradually translate C code into assembly preserving the behaviour. Proving the translation directly without going through the intermediate languages would be infeasible, especially with the many optimisations that are performed during the translation, as there is a large semantic gap between the semantics of Asm and the semantics of Clight. The first three intermediate languages (C#minor, Cminor, CminorSel) are used to transform Clight, a deterministic subset of C, into a more assembly-like language called register transfer language (Rtl). This language consists of a CFG of instructions, and is therefore well suited for various compiler optimisations such as constant propagation, dead-code elimination or function inlining. After Rtl, each intermediate language is used to get

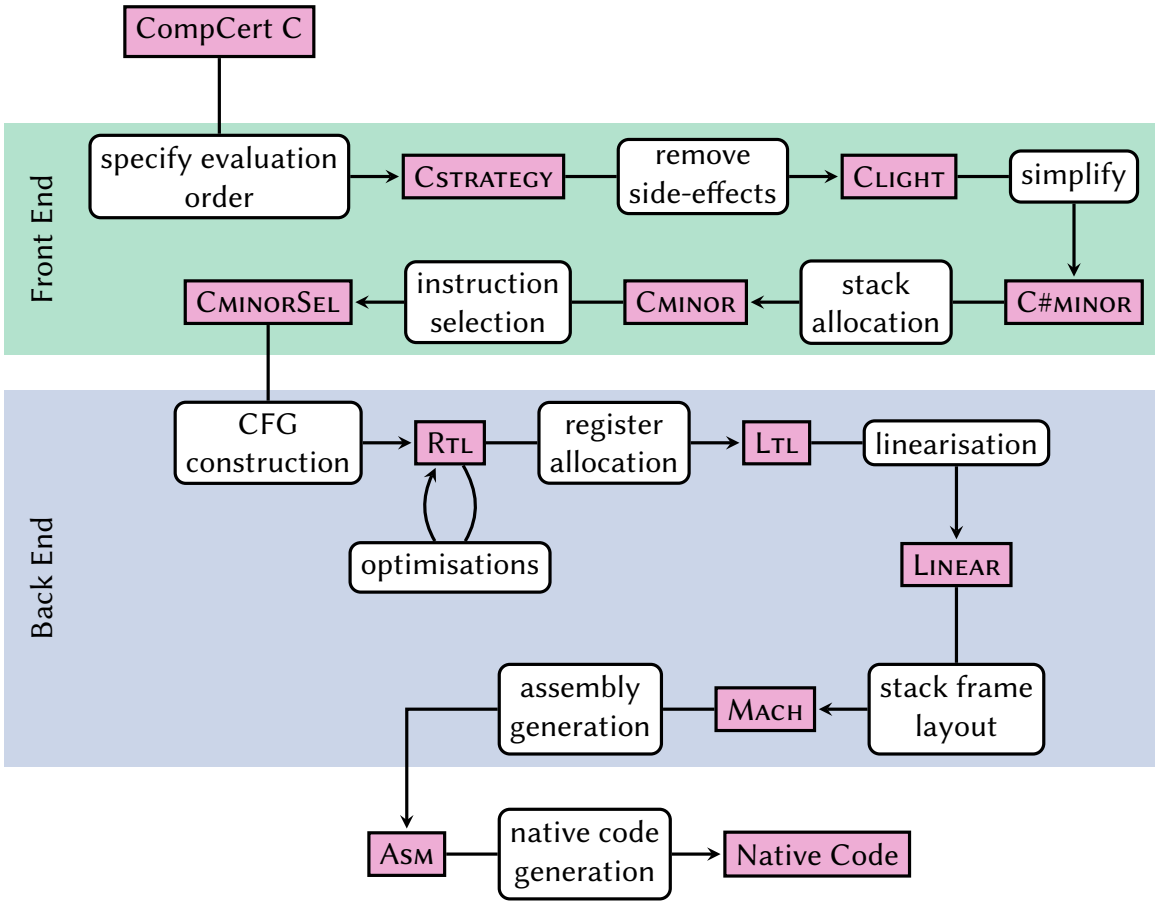


Figure 2.6: CompCert intermediate languages in the front end and back end of the compiler. The parts outside of the coloured boxes are trusted, whereas all the languages transformations within the coloured boxes are verified and untrusted.

closer to the assembly language of the architecture, performing operations such as register allocation. Figure 2.6 gives a summary of each transformation that takes place in CompCert, from the input C language to the final native code that is run on the CPU.

CompCert proofs follow two main patterns. The first type of correctness proof is a direct proof that the algorithm implemented in Coq is correct. These types of proofs do not carry any run time cost, because they reason purely about the algorithm that was implemented and they can be erased when the C compiler is extracted to OCaml code. The translation algorithm itself is proven correct, so no additional code that is associated with the proof is needed when one is using the compiler. However, some transformations are easier to build a validator for that checks the result of the transformation instead of proving that the transformation algorithm always produces correct results. CompCert uses translation validation to prove the correctness of the register allocation transformation, for

example. Register allocation reduces to the graph colouring problem, where all registers with overlapping liveness properties are connected by edges. The algorithm is then tasked to colour the graph with the fewest colours so that connected nodes have different colours. It is therefore clear that checking the colouring of such a graph is simple, as one can check that nodes connected by an edge have different colours. However, proving that such an algorithm will always produce such a graph is more difficult because many heuristics can be used during the graph colouring to try and achieve the fewest colours efficiently. The algorithm that checks the colours of the graph is called the validator. CompCert implements register allocation in this way, performing the register allocation in unverified OCaml code, and implementing a verified validator for register allocation in Coq. This still produces a verified translation, as CompCert is allowed to fail on inputs, in which case the correctness theorem holds trivially. For translation validation, that means that as long as compilation only proceeds when the validator succeeds, it is equivalent to having verified the transformation correct. However, using a validator means that it needs to check the correctness criterion every time the compiler is run, meaning the efficiency of the validator is also important to take into account.

2.7.1 CompCert correctness theorem

The correctness theorem of CompCert should ensure that the compiler does not introduce bugs into the program as it is translated from C into assembly. To do so, one must first define the execution of programs written in C, as well as programs written in assembly.

Small-step semantics Small-step semantics in CompCert are defined as a labelled transition system. A transition between states s and s', emitting events t, is denoted as a single step $s \xrightarrow{t} s'$. In addition to that, one also needs to designate an initial state and a final state, the latter returning the final result after executing the program. Finally, each program is also associated with a global environment. CompCert defines a small-step semantics framework that can be reused by most languages, providing additional useful constructs such as the reflexive, transitive closure of $\longrightarrow$ denoted as $\longrightarrow^*$ or the transitive 'plus' closure denoted as $\longrightarrow^+$. A semantics for a CompCert language is created by defining the type for the state, for a step in the semantics, and the initial and final states and the global environment of the program.

Program behaviour Once the semantics have been defined, the global behaviour of the program can be defined, which will be used to express the correctness theorem. A C program can behave in three main ways, it can either *terminate successfully*, *diverge*, or finally *go wrong*. Successful termination happens when the final state is reached from the initial state, in which case the final state will contain the return-value of the main function in addition to a finite stream of events. If the program diverges, then the program is executing indefinitely, in which case it will emit a possibly infinite stream of events but no return value. Finally, if the program goes wrong, for example when undefined behaviour is encountered, it means that there is no more valid step defined in the semantics from the current state.

Correctness theorem

The correctness theorem can be stated as a simulation between the semantics of the source program and the semantics of the compiled program. The final correctness theorem in CompCert is a backward simulation between the source program semantics and compiled program semantics, stated as follows, where we write $\lfloor x \rfloor$ for the presence of value x in an option type and we write $A \Downarrow \mathcal{B}$ for an execution of A leading to an observable behaviour $\mathcal{B}$.

Theorem 2.1 (Semantic preservation)**.** *If a successfully compiled assembly program C has a behaviour $\mathcal{B}$ which does not go wrong, then there must exist behaviour $\mathcal{B}'$ of source program S which may be less defined than behaviour $\mathcal{B}$, which is expressed by $\mathcal{B}' \precsim \mathcal{B}$.*

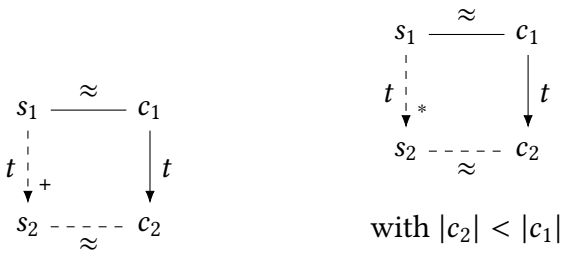
$$\texttt{compile\_c}\ S = \lfloor C \rfloor \implies (\forall \mathcal{B}.\, C \Downarrow \mathcal{B} \implies (\exists \mathcal{B}'.\, S \Downarrow \mathcal{B}' \wedge \mathcal{B}' \precsim \mathcal{B})) \tag{2.3}$$

A direct corollary of this is the following semantic preservation, where one proves that the source program is *safe*, i.e. that it is free of behaviour that goes wrong. In that case, one cannot define any more behaviour, so $\mathcal{B}$ must be a valid behaviour for the source program.

Corollary 2.1.1 (Refinement of compilation)**.** *If a successfully compiled assembly program C has a behaviour $\mathcal{B}$ and the source program is* safe*, meaning it is free of undefined behaviour, then $\mathcal{B}$ must be a behaviour of source program S.*

$$\texttt{compile\_c}\ S = \lfloor C \rfloor \wedge \mathit{safe}(S) \implies (\forall \mathcal{B}.\, C \Downarrow \mathcal{B} \implies S \Downarrow \mathcal{B}) \tag{2.4}$$

This theorem correctly conveys the property that the compilation of the source program should not introduce any bugs, which is especially clear in the corollary. As long as



(a) Backward simulation diagrams with a well-founded order on the compiled program state.

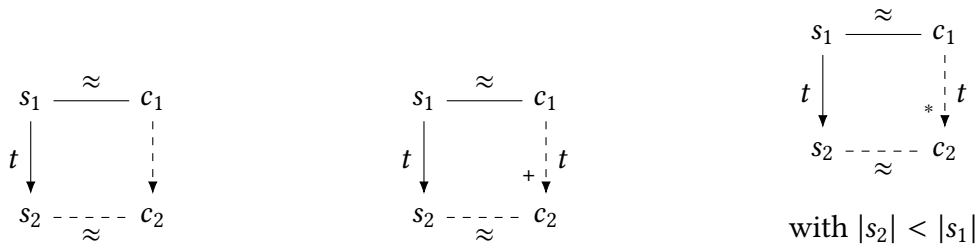
(b) Progress property needed for the backward simulation.

Figure 2.7: Examples of simulation diagrams that make up the backward simulation.

one proves that the source program does not have any undefined behaviour, then every behaviour of the compiled program needs to be a behaviour of the source program.

These theorems are proven by showing simulations between the semantics of the source program and the semantics of the compiled program. In particular, theorem 2.1 can be proven to hold if one can show a backward simulation between the source program and the compiled program, which is how the theorem is proven in CompCert. A backward simulation is a number of relations between states and transitions of the semantics of both programs. The heart of the backward simulation can be represented as a *simulation diagram* and shown in figure 2.7. Figure 2.7a shows the main two types of simulation diagrams that may hold for a specific transition, where the solid lines represent what can be assumed, and the dashed lines have to be shown to hold. First, for a step in the compiled program from state c_1 to state c_2 that emits trace t and a state s_1 which matches with the state c_1 using the $\approx$ relation, there must exist one or more steps in the source program that also emit the same trace t and produce a state s_2 that matches with state c_2. If the state transition in the compiled program does not emit a trace, but some measure on the state has a well-founded order that decreases, then the source semantics may stall as long as s_1 matches with both c_1 and c_2. In addition to that, to be able to show semantic preservation from the backward simulation, one also needs to prove that the compiled program makes progress if there exist transitions in the source program from two matching states s_1 and c_1. This is shown in figure 2.7b.

Backward simulations are hard to prove by induction on the semantics of the compiled program, because the matching relation requires that one define a decompilation of the compiled program into the source program to match individual instructions with individual sections of the code. This may not be possible for many constructs, as statements generally



(a) Lock-step (b) 'plus' (c) 'star'

Figure 2.8: Forward simulation diagrams. The forward simulation is defined in terms of the more general 'plus' or 'star' simulation diagrams, however, the lock-step simulation diagram is a useful special case of the 'plus' simulation diagram which can be used for many of the translation passes.

compile to a group of instructions. Instead, a forward simulation, shown in figure 2.8, is much more natural to prove correct, because one can induct over the semantics of the source program and define the matching relation in terms of the compilation. However, without additional properties, the forward simulation is not sufficient to imply semantic preservation, because a forward simulation would allow for additional behaviours in the compiled program that were not behaviours of the source program.

However, if one can show that the semantics of the compiled program is deterministic, then the forward simulation implies the backward simulation. In addition, forward simulations as well as backward simulations can be composed. This means that one can prove a forward simulation for each transformation shown in figure 2.6 from Clight to Asm which implies a forward simulation from Clight to Asm. After showing that the semantics of Asm is deterministic, one can then prove a backward simulation from Asm to Clight.

2.7.2 Instruction scheduling in CompCert

There are a few implementations of scheduling algorithms in CompCert or in derivatives of CompCert that are relevant to this dissertation, as the scheduling algorithm is central to the HLS transformation. First, I will describe implementations of list scheduling in CompCert, followed by an implementation of superblock scheduling and finally an implementation of trace scheduling. These scheduling methods all implement validators based on symbolic execution of the code.

Symbolic expressions are formulated in terms of initial values of registers (denoted by $\cdot^0$) at the start of the block. For example, figure 2.9 shows an example of symbolic execution,

```
z := x + y;
t := z * y;
M[12] := x;
y := M[x];
```

$$
\begin{aligned}
\texttt{z} &\mapsto \texttt{x}^0 + \texttt{y}^0 \\
\texttt{t} &\mapsto (\texttt{x}^0 + \texttt{y}^0) * \texttt{y}^0 \\
\texttt{M} &\mapsto \texttt{store}(\texttt{M}^0, \texttt{x}^0, 12) \\
\texttt{y} &\mapsto \texttt{load}(\texttt{store}(\texttt{M}^0, \texttt{x}^0, 12), \texttt{x}^0) \\
r &\mapsto r^0 \quad \text{for all other registers } r
\end{aligned}
$$

Figure 2.9: Example of symbolic execution adapted from Tristan and Leroy [2008].

where the result is a map from *resources*, which might be memory or registers, to symbolic expressions, which are expressions in terms of initial values of registers. The symbolic execution proceeds sequentially through the linear block and updates the symbolic state. If a register is read from, its current symbolic expression is looked up in the symbolic state and replaces the register to build a new symbolic expression.

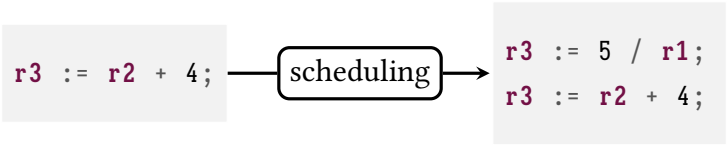
The symbolic state can then be used to validate code transformations. If the code transformation only reorders instructions, then the correctness of the transformation only relies on data dependencies, which is naturally encoded in the symbolic state. As long as one can show that the symbolic expressions of two registers are syntactically equal, then this means that the reordering of instructions did not violate any data dependencies, and this should imply that the behaviours of the original block and the reordered block are identical as well.

List scheduling

Tristan and Leroy [2008] were the first to propose adding scheduling to a verified compiler. Six et al. [2020] then optimise the validator and extend it so that it works with the Kalray KVX VLIW processor as a post-pass scheduling step in their translation and is integrated into a fork of CompCert called CompCert KVX [Six et al. 2023]. In both these cases, the list scheduling algorithm works on the Mach intermediate language in CompCert, but in the case of CompCert KVX the list scheduling pass is also used to transform Mach into Asm. List scheduling is performed on basic blocks. These are implicitly represented in Tristan and Leroy [2008], but are explicitly constructed in Six et al. [2020] as an AsmBlock language, making it easier to reason about transformations in individual blocks.

The schedule is validated after the fact by running symbolic execution before and after scheduling. The symbolic execution of both blocks starts with the same symbolic state, sequentially symbolically executing each instruction while updating the symbolic state. The final states reached for both the block before and after scheduling can then be compared. If the scheduler only *reorders* instructions, syntactic equality suffices for comparing symbolic

expressions, as an instruction is only reordered if it does not break any data-dependencies. In that case, the symbolic expressions for the reordered instruction would be identical as they were independent. However, the scheduler is unverified, and it could therefore introduce new instructions. As a result, the comparison of the symbolic state is not enough to show equivalent behaviour. Consider the following example:

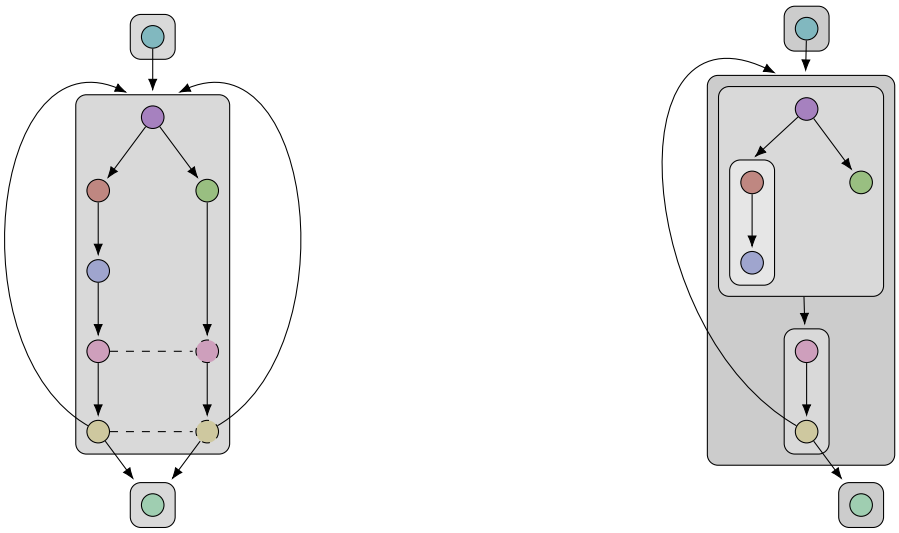


In this case, the scheduler introduces an additional instruction that is later overwritten. The symbolic states will therefore be equivalent, however, the transformed block may encounter undefined behaviour when `r1` is 0. To guard against the scheduler introducing additional instructions that may have undefined behaviour, Tristan and Leroy introduce the concept of *constraints*, which is the set of all previously encountered operations. In the case of the example, the constraint set of the original block would be empty, whereas the constraints of the second block would be { `5 / r1` }. For correctness, the scheduled basic block should therefore not introduce new constraints, like in the previous example. Instead, the validation has to ensure that the constraints of the scheduled block are a subset of the constraints of the original block.

Together, this forms a validation algorithm for list scheduling by comparing the symbolic states of the basic blocks for equivalence, and ensuring that the constraints form a subset. The post-pass scheduler by Six et al. optimises the validation algorithm by using hash-consing to replace the expensive expression comparison by pointer comparisons.

2.7.3 Trace scheduling

Tristan and Leroy [2008] also formalise a version of trace scheduling, where instructions can be moved along traces of the program that are free of back edges, and can therefore be moved across basic block boundaries. This gives more freedom to the scheduler, leading to more optimal code in most cases. The first part of the trace scheduling algorithm is to transform the CFG into a graph of trees representation, shown in figure 2.10a. This representation is similar to the superblock representation shown in figure 2.4b where tail duplication is needed to represent any internal joining control flow. However, this tail duplication then means that the whole block can be represented as a single tree. Instructions



(a) Graph of trees

(b) Block transfer language (BTL)

Figure 2.10: Comparison of the graph of trees structure and the block transfer language (BTL) structure, extending the comparisons shown in figure 2.4.

can be reordered by the scheduler within a tree. In general, tail duplication should be avoided as it can result in an exponential increase in the number of nodes. The boundaries of the trees are chosen by calculating possible cut points in the CFG, which correspond to labels that are the target of back edges, or function calls.

Validation is performed by running the symbolic execution over the tree and comparing symbolic states in the same way as was done for list scheduling when one reaches the leaves of the tree (i.e. the exit nodes). While traversing the tree, one must also ensure that the same conditions are encountered and evaluated, and these have to remain in the same order in the tree. Instructions that are moved after a control flow statement need to be duplicated by the scheduler to remain correct.

In general, this validation technique for trace scheduling is effective at verifying complex schedules, but it can quickly become infeasible to check the equivalence as the size of the blocks grows. This is mainly due to two sources of inherent inefficiencies in the validation algorithm. First, the size of the symbolic expressions can grow exponentially as they do not share any subexpressions, which means that comparing two expressions can take exponentially longer. This affects both the list scheduling and the trace scheduling validator, and is addressed by Six et al. through hash-consing. Secondly, comparing two trees symbolically can also lead to an exponential number of comparisons of symbolic

states, due to duplicated expressions in different branches of the tree. This means that the choice of cut points becomes important to try and minimise the size of the trees and still achieve a good schedule.

Superblock and extended block scheduling

Six et al. [2022] then formalise superblock scheduling as a pre-pass optimisation at the RTL level, which is a restricted form of trace scheduling that was specifically developed to target VLIW processors [Hwu et al. 1993]. The idea is to translate RTL code into RTLPATH code, which is a superblock representation of the original code containing multiple traces through the extended non-branching block. The superblock construction needs to take into account which traces through the program will be executed more frequently and must group those together. Six et al.'s scheduler then reorders instructions within a superblock trace, and also combines them where it is advantageous to do so, making it necessary to allow for rewrites in the symbolic expressions as well.

The validator works similarly to the post-pass list scheduling algorithm in CompCert KVX, but it is extended to support symbolic expression normalisation to verify rewrites and expansion of instructions correct. It therefore also includes all the optimisations from the list scheduling post-pass scheduler in CompCert KVX such as hash-consing. In addition to that, liveness is added to the correctness theorem so that equivalence only needs to be checked for registers that are live at this particular superblock exit. This makes it possible to introduce new registers, and allows for instructions to be expanded into multiple instructions, without having to reason about any intermediate registers that were introduced.

In contrast to the list scheduling transformation, the superblock transformation requires validation of symbolic states at each exit of the superblock with the corresponding exit in the scheduled superblock. This is similar to the validation of graph of trees, except that traces are still represented as lists instead of trees. Gourdin et al. [2023] later replaced RTLPATH with block transfer language (BTL), a general intermediate block language shown in figure 2.10b, where blocks are either: (1) a sequence of blocks, shown in the diagram as being connected by an arrow inside of a grey box, (2) a conditional instruction containing two blocks, shown as a node with two split arrows pointing to a block, (3) a standard instruction, shown as a simple node in the graph, or finally (4) a control-flow instruction, shown as a node in the graph with an edge exiting the block and pointing to a new external block. BTL is defined as a nested structure of blocks with conditionals, and is therefore

general enough to represent hyperblocks. The main difference between the graph of trees representation and Btl is that graph of trees cannot represent a sequence of two trees, which is representable in Btl. As the language is not restricted to superblocks anymore, the scheduler was modified to work with *extended blocks*, which are essentially equivalent to the graph of trees representation, as they are trees without any internal joining control flow. This makes the scheduler more flexible, making it possible to apply light loop pipelining optimisations by scheduling an unrolled loop. However, the symbolic execution is still performed on each trace through the program, and no sharing between tracing takes place, meaning that the validation is exponential in the number of internal joins. Only extended blocks are scheduled so this is avoided.

2.8 Summary

This chapter described the process of high-level synthesis, as well as how verification is normally performed when using high-level synthesis to produce hardware designs. The main conclusion is that when using high-level synthesis one is mainly relying on testing to check the equivalence between high-level designs and the low-level hardware design. Further testing in then purely performed on the low-level hardware design at the register-transfer level, where it can be difficult to verify behaviour compared to using the high-level design.

I gave an overview of CompCert, describing the correctness theorem and describing existing attempts at formally verifying instruction scheduling, which is the main optimisation and transformation that HLS tools perform to generate hardware. The current formalised scheduling passes are mainly targeting CPUs, where it is more important to schedule instructions over traces. HLS benefits from using general hyperblocks, especially combining blocks where control flow is joined internally, meaning it benefits from a different intermediate representation that is more tailored to hardware.

3 Introduction to Vericert

This chapter describes the main architecture of the HLS tool, and the way in which the Verilog back end was added to CompCert. This chapter also covers an example of converting a simple C program into hardware, expressed in the Verilog language. Section 3.1 is based on a short paper coauthored with Zewei Du, Nadesh Ramanathan, and John Wickerson [Herklotz et al. 2021a]. The rest of the chapter, together with chapter 4, is based on a paper coauthored with James D. Pollard, Nadesh Ramanathan, and John Wickerson [Herklotz et al. 2021b].

3.1 Unreliability of High-Level Synthesis

Are HLS tools reliable? Questions have been raised about the reliability of HLS before; for example, Andrew Canis, co-creator of the LegUp HLS tool, wrote that 'high-level synthesis research and development is inherently prone to introducing bugs or regressions in the final circuit functionality' (Canis [2015, Section 3.4.6]). In this section, I investigate whether there is substance to this concern by conducting an empirical evaluation of the reliability of several widely used HLS tools.

The approach used is called *fuzzing*. This is an automated testing method in which randomly generated programs are given to compilers to test their robustness [Chen et al. 2013; Liang et al. 2018; Lidbury et al. 2015; Sun et al. 2016; Yang et al. 2011; Zhang et al. 2019]. The generated programs are typically large and rather complex, and they often combine language features in ways that are legal but counter-intuitive; hence they can be effective at exercising corner cases missed by human-designed test suites. Fuzzing has been used extensively to test conventional compilers; for example, Yang et al. [2011] used it to reveal more than three hundred bugs in GCC and LLVM, and tested CompCert as well, only finding bugs in the verified parts of the code.

```
unsigned int x = 0x1194D7FF;
int arr[6] = {1, 1, 1, 1, 1, 1};

int main() {
  for (int i = 0; i < 2; i++)
    x = x >> arr[i];
  return x;
}
```

Figure 3.1: Miscompilation bug in Xilinx Vivado HLS. The generated hardware returns `0x006535FF` but the correct result is `0x046535FF`.

Example 3.1 (A miscompilation bug in Vivado HLS)**.** Figure 3.1 shows a program that produces the wrong result during hardware simulation in Xilinx Vivado HLS v2018.3, v2019.1 and v2019.2.[1] The program repeatedly shifts a large integer value `x` right by the values stored in array `arr`. Vivado HLS returns `0x006535FF`, but the result returned by GCC (and subsequently confirmed manually to be the correct one) is `0x046535FF`. The bug was initially revealed by a randomly generated program of around 113 lines, which was reduced to the minimal example shown in the figure. I reported this issue to Xilinx, who confirmed it to be a bug.[2]

The fuzzer used Csmith to randomly generate C code and modified it to make it suitable for HLS tools. For example, pointers had to be removed because some cases, like pointers of other pointers, are explicitly not supported by Vivado HLS. I tested the following HLS tools: Intel i++ 18.1, LegUp 4.0, Bambu 0.9.7 and Xilinx Vivado HLS v2018.3, v2019.1 and v2019.2. Figure 3.2 shows an Euler diagram of our results. We see that 918 (13.7%), 167 (2.5%), 83 (1.2%) and 26 (0.4%) test-cases fail in Bambu, LegUp, Vivado HLS and Intel i++ respectively. The bugs I reported to the Bambu developers were fixed during our testing campaign, so I also tested the development branch of Bambu (0.9.7-dev) with the bug fixes, and found only 17 (0.25%) failing test-cases remained. Although i++ has a low failure rate, it has the highest time-out rate (540 test-cases) due to its remarkably long compilation time. No other tool had more than 20 time-outs. Note that the absolute numbers here do not necessarily correspond to the number of bugs in the tools, because a single bug in a language feature that appears frequently in our test suite could cause many failures.

[1] This program, like all the others in this paper, includes a `main` function, which means that it compiles straightforwardly with GCC. To compile it with an HLS tool, I rename `main` to `result`, synthesise that function, and then add a new `main` function as a testbench that calls `result`.

[2] https://web.archive.org/web/20210419185153/https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Issue-with-shift-in-for-loop/m-p/1170197

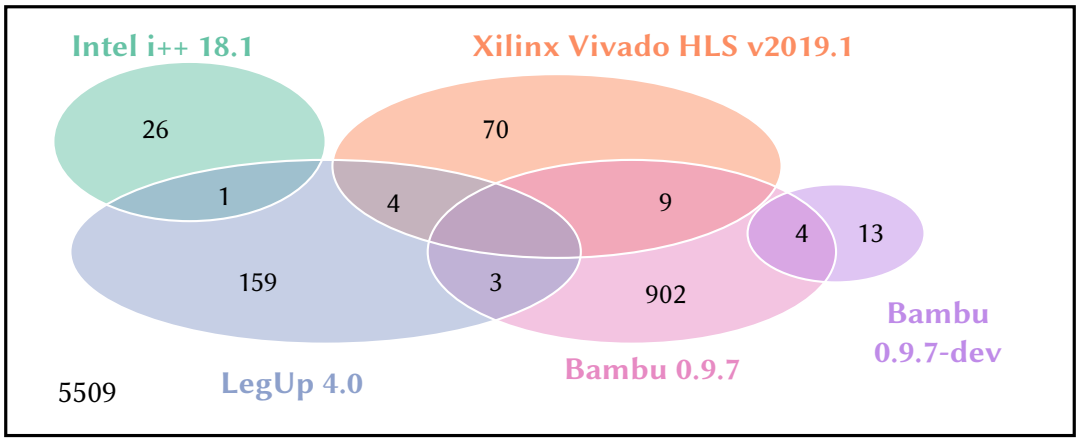


Figure 3.2: The number of failures per tool out of 6700 test-cases. Overlapping regions mean that the same test-cases failed in multiple tools.

Moreover, I am reluctant to draw conclusions about the relative reliability of each tool by comparing the number of failures, because these numbers are so sensitive to the parameters of the randomly generated test suite used. In other words, I can confirm the *presence* of bugs, but cannot deduce the *number* of them (nor their importance).

In total, I found at least 8 unique bugs across all the tools, including both crashes and miscompilations. Conventional compilers have become quite resilient to fuzzing over the last decade, so recent work on fuzzing compilers has had to employ increasingly imaginative techniques to keep finding new bugs [Even-Mendoza et al. 2021]. In contrast, I have found that HLS tools can be made to exhibit bugs even using the relatively basic fuzzing techniques. This motivates the need for a verified HLS tool.

3.2 Main Design Decisions of Vericert

Our solution to the reliability problem in HLS is Vericert, a formally verified HLS tool. First, I will describe the main design decisions behind Vericert in section 3.2 and I will then give an example of a translation from C code into Verilog through Vericert in section 3.3.

Choice of source language C was chosen as the source language as it remains the most common source language amongst production-quality HLS tools [AMD 2023b; Canis et al. 2013; Intel 2020a; Pilato and Ferrandi 2013]. This, in turn, may be because it is '[t]he starting point for the vast majority of algorithms to be implemented in hardware' (Gajski et al. [2010]), lending a degree of practicality. The availability of CompCert [Leroy 2009b]

also provides a solid basis for formally verified C compilation. Since a lot of existing code for HLS is written in C, supporting C as an input language, rather than a custom domain-specific language, means that Vericert is more practical. I considered Bluespec [Nikhil 2004], but decided that although it 'can be classed as a high-level language' (Greaves [2019]), it is too hardware-oriented to be suitable for traditional HLS. I also considered using a language with built-in parallel constructs that map well to parallel hardware, such as occam [Page and Luk 1991], Spatial [Koeplinger et al. 2018] or Scala [Bachrach et al. 2012].

Choice of target language Verilog was chosen as the output language for Vericert because it is one of the most popular HDLs and there already exist a few formal semantics for it that could be used as a target [Lööw et al. 2019; Meredith et al. 2010]. Bluespec, previously ruled out as a source language, is another possible target and there exists a formally verified translation to circuits of variants of Bluespec using Kôika [Bourgeat et al. 2020] or Fe-Si [Braibant and Chlipala 2013]. However, Bluespec is mainly targeted at being a source language for hardware design and would present similar challenges to targeting a language like Verilog from a software language.

Choice of implementation language I chose Coq as the implementation language because of its mature support for code extraction; that is, its ability to generate OCaml programs directly from the definitions used in the theorems. CompCert [Leroy 2009b] was chosen as the front end because it has a well established framework for simulation proofs about intermediate languages, and it already provides a validated C parser [Jourdan et al. 2012]. The Vellvm framework [Zhao et al. 2012] was also considered because several existing HLS tools are already LLVM-based, but additional work would be required to support a high-level language like C as input. The .NET framework has been used as a basis for other HLS tools, such as Kiwi [Greaves and Singh 2008], and LLHD [Schuiki et al. 2020] has been recently proposed as an intermediate language for hardware design, but neither are suitable for us because they lack formal semantics.

Architecture of Vericert An overview of Vericert's workflow is given in figure 3.3, which shows that Vericert branches off from CompCert at the RTL stage, followed by a number of transformations related to the scheduling instructions, and finally transformations that generate the final hardware.

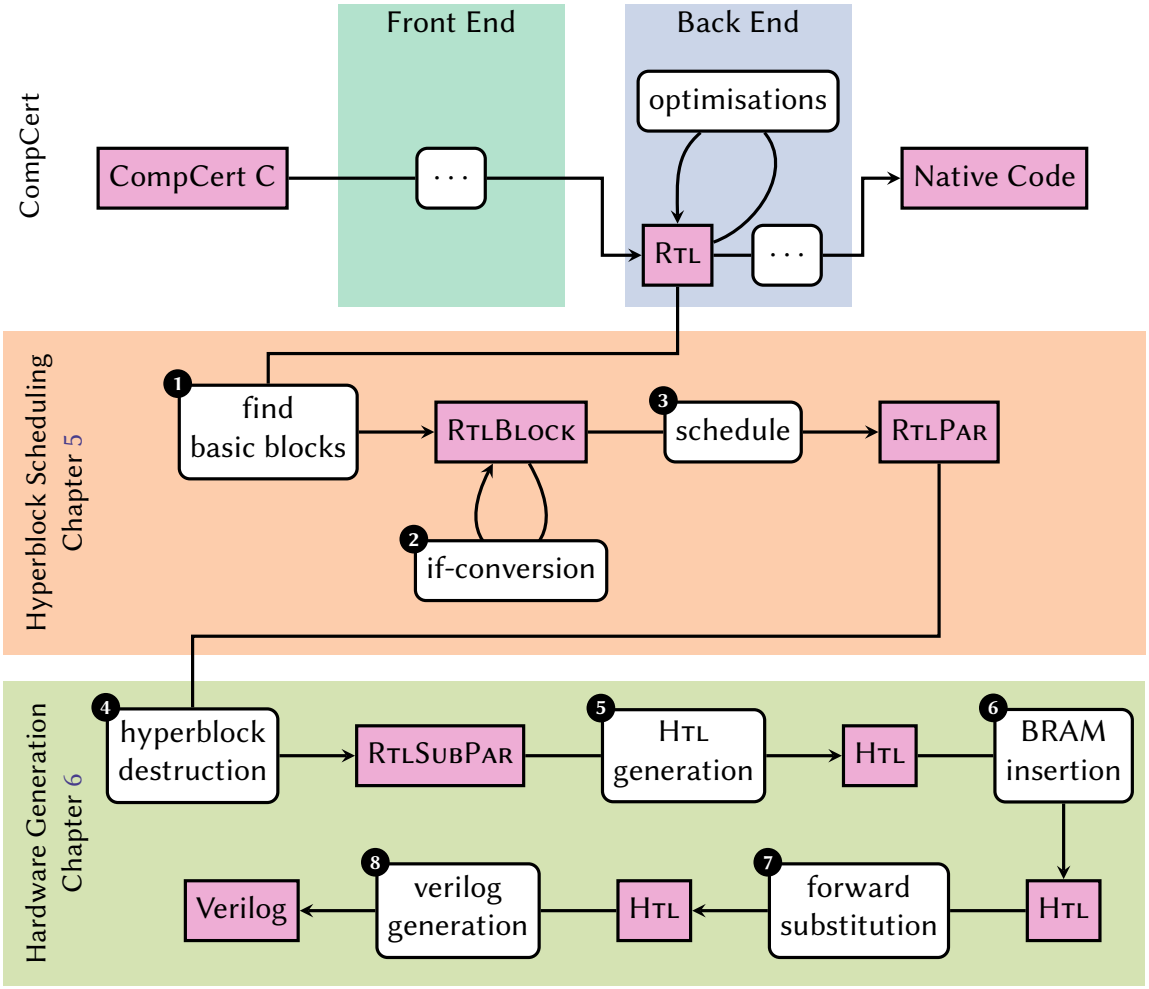


Figure 3.3: Vericert as a Verilog back end to CompCert.

CompCert's RTL was selected as the starting point. Branching off *before* this point (at CMINORSEL or earlier) denies CompCert the opportunity to perform optimisations such as constant propagation and dead-code elimination, which, despite being designed for software compilers, have been found useful in HLS tools as well [Cong et al. 2011]. And if we branch off *after* this point (at LTL or later) then CompCert has already performed register allocation to reduce the number of registers and spill some variables to the stack; this transformation is not required in HLS because there are many more registers available, and these should be used instead of BRAM whenever possible.

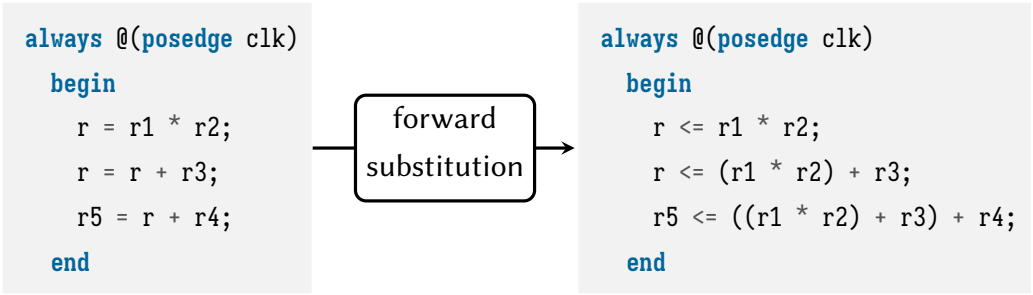
RTL is also attractive because it is the closest intermediate language to LLVM IR, which is used by several existing HLS compilers. It has an unlimited number of pseudo-registers, and is represented as a CFG where each instruction is a node with links to the instructions that can follow it. RTL does not have the SSA property, however, this is not required for the translation to hardware and mainly assists the static analysis passes. One difference between LLVM IR and RTL is that RTL includes operations that are specific to the chosen target architecture; I chose to target the x86_32 back end because it generally produces relatively dense RTL thanks to the availability of complex addressing modes. The translation from RTL is then performed as follows:

1. Basic block generation creates basic blocks from the pure RTL CFG.
2. If-conversion combines basic blocks into a single hyperblock based on heuristics on the CFG layout. The hyperblock is represented as a basic block of predicated instructions with an exit instruction to leave the basic block prematurely.
3. The scheduler itself is written in unverified OCaml and works similarly to those in existing HLS tools [Canis et al. 2013]: it takes a set of scheduling constraints that capture the clock period, available hardware resources, and dependencies between operations, encodes them as a system of difference constraints (SDC) [Cong and Zhang 2006], and then hands them off to a linear program solver. The result is checked by a verified validator that ensures the correctness of the schedule that was produced by the scheduler.
4. Next, the hyperblocks are destroyed to explicitly place instructions into individual states.
5. HTL is generated, which is a language that models an FSMD, which is used as an intermediate language that is simpler to manipulate than Verilog but is still close

enough to a hardware description so that hardware-specific optimisations can be performed on it.

❻ BRAM insertion generates a proper memory interface for any interaction with the stack frame.

❼ To ensure that these assignments are actually performed in parallel, a final pass performs *forward substitution* [Hopwood 1978, p. 109] to turn the sequence of Verilog blocking assignments into a sequence of nonblocking assignments. For example:



The two versions are semantically equivalent, but we find that the second, in which both right-hand sides must be evaluated before either assignment is performed, makes the downstream logic synthesis tools more likely to produce the hardware we intend (which, in this particular example, involves exploiting a fused multiply–accumulator unit if available).[3]

❽ Finally, syntactic Verilog is generated from HTL, which consists of translating the FSMD into a case statement and implementing the memory interface in Verilog.

3.3 Translating C to Verilog by Example

Figure 3.4 illustrates the translation of a simple program that stores and retrieves a value from an array. This section describes the stages of the Vericert translation, referring to this program as an example.

[3] As the example shows, forward substitution does not remove duplicate writes to a variable because Verilog semantics explicitly state that the order of nonblocking assignments to the same variable will be preserved [IEEE 2024, p. 254].

```c
int main(int i) {
    int x[2] = {3, 6};
    return x[i];
}
```

(a) Example C code passed to Vericert.

```
main(r1) {
  7: r5 := 3
  6: M[&Stack[0]] := r5
  5: r4 := 6
  4: M[&Stack[4]] := r4
  3: r3 := &Stack[0]
  2: r2 := M[r3 + r1 * 4 + 0]
  1: return r2
}
```

(b) RTL produced by the CompCert front end without any optimisations.

Figure 3.4: Translating a simple program from C to RTL.

3.3.1 Translating C to RTL

The first stage of the translation uses unmodified CompCert to transform the C input, shown in figure 3.4a, into an RTL intermediate representation, shown in figure 3.4b. As part of this translation, function inlining is performed on all functions, which allows us to support function calls without having to support the `Icall` RTL instruction. Although the duplication of the function bodies caused by inlining can increase the area of the hardware, it can have a positive effect on latency and is therefore a common HLS optimisation [Noronha et al. 2017]. Scheduling in particular benefits from inlining of function calls so that the instructions can be scheduled together and larger hyperblocks can be formed. Inlining precludes support for recursive function calls, but this feature is not supported in most HLS tools anyway [Thomas 2016].

3.3.2 Scheduling RTL instructions

The first step in the translation performed by Vericert is to schedule the instructions according to the resource constraints imposed by the hardware target. An example of such a constraint is that two memory operations cannot be performed in the same cycle. The goal is to schedule as many instructions as possible together to give the scheduler the most flexibility. Vericert therefore constructs hyperblocks from the RTL CFG, by building basic blocks first and then performing if-conversion on blocks that should be merged. This is shown in figure 3.5a, where all the instructions can fit into a single basic block. Each hyperblock can then be scheduled individually using SDC scheduling by specifying any necessary constraints, the result of which is shown in figure 3.5b, where one can see that

```
main(r1) {
  7: {
    r5 := 3
    M[&Stack[0]] := r5
    r4 := 6
    M[&Stack[4]] := r4
    r3 := &Stack[0]
    r2 := M[r3 + r1 * 4 + 0]
    return r2
  }
}
```

(a) Code in RtlBlock after basic blocks have been generated.

```
main(r1) {
  7: {
    r5 := 3
    r4 := 6
    r3 := &Stack[0]
    goto 13
  }
  13: {
    M[&Stack[0]] := r5
    goto 14
  }
  14: {
    M[&Stack[4]] := r4
    goto 15
  }
  15: {
    r2 := M[r3 + r1 * 4 + 0]
    goto 16
  }
  16: { return r2 }
}
```

(b) RtlSubPar code produced after scheduling and hyperblock destruction.

Figure 3.5: Scheduling a simple program from RtlBlock to RtlPar.

instructions with dependencies have been separated into different states and instructions that can execute in parallel are placed into the same state. Each memory operation is placed into its own state. The instructions are placed into additional bundles, designated by parentheses, which could contain additional operations that are chained sequentially within one clock cycle.

The SDC scheduling algorithm itself is unverified but a verified validator checks the resulting schedule against the unscheduled program. This makes it possible to change heuristics in the scheduler without it affecting the proof. The scheduled program language is called RtlPar and specifies in which cycle each instruction will be executed. RtlPar however still contains the coarse-grained structure of hyperblocks, so this structure is destroyed to produce RtlSubPar, shown in figure 3.5b. The representation is still hardware agnostic even if the current implementation of the scheduler is specific to an FPGA target, and therefore supports all Rtl instructions. Details about the scheduler and proof of correctness are given in chapter 5. This representation is then ready to be translated into hardware.

3.3.3 Translating RTLPAR to HTL

The next translation is from RTLPAR, the program formed of scheduled hyperblocks, to an intermediate hardware translation language (HTL). This translation involves going from a CFG representation of the computation to an FSMD representation [Hwang et al. 1999]. An FSMD is a generalised state machine where the state is supplemented by registers and memory that can store results of computations. Figure 3.6 shows the resulting FSMD architecture for the running example. The right-hand block is the control logic that computes the next state, while the left-hand block updates all the registers and BRAM based on the current program state. However, in general the state machine cannot be separated from the data path completely, because state updates may depend on computations in the data path and on the value of registers. Hence, an HTL program consists of a map from states to Verilog statements, which describe updates done to both the state register as well as other registers and interact with memory.

The HTL language was mainly introduced to simplify the proof of translation from RTL to Verilog, as these languages have very different semantics. It serves as an intermediate language with similar semantics to RTL at the top level, using maps to represent what to execute at every state, and similar semantics to Verilog at the lower level because it already uses Verilog statements instead of more abstract instructions to perform computations. The next state is also computed explicitly in each state by modifying the state register. Compared to plain Verilog and due to using maps to represent the Verilog statement that should execute at every state, HTL is simpler to manipulate and analyse, thereby making it easier to prove optimisations like BRAM insertion.

Translating memory Typically, HLS-generated hardware consists of a sea of registers and RAMs. This memory view is very different from the C memory model, so I perform the following translation from CompCert's abstract memory model to a concrete BRAM. Variables that do not have their address taken are kept in registers, which correspond to the registers in RTL. All address-taken variables, arrays, and structs are kept in BRAM. The stack frame of the main function becomes an unpacked array of 32-bit integers representing the BRAM block. Any loads and stores are temporarily translated to direct accesses to this array, where each address has its offset removed and is divided by four. In a separate HTL-to-HTL conversion, these direct accesses are then translated to proper loads and stores that use an interface to communicate with the BRAM, shown on lines 32–33, 36–37 and 40–41 of figure 3.7 in the final Verilog design. This pass inserts a BRAM block with an

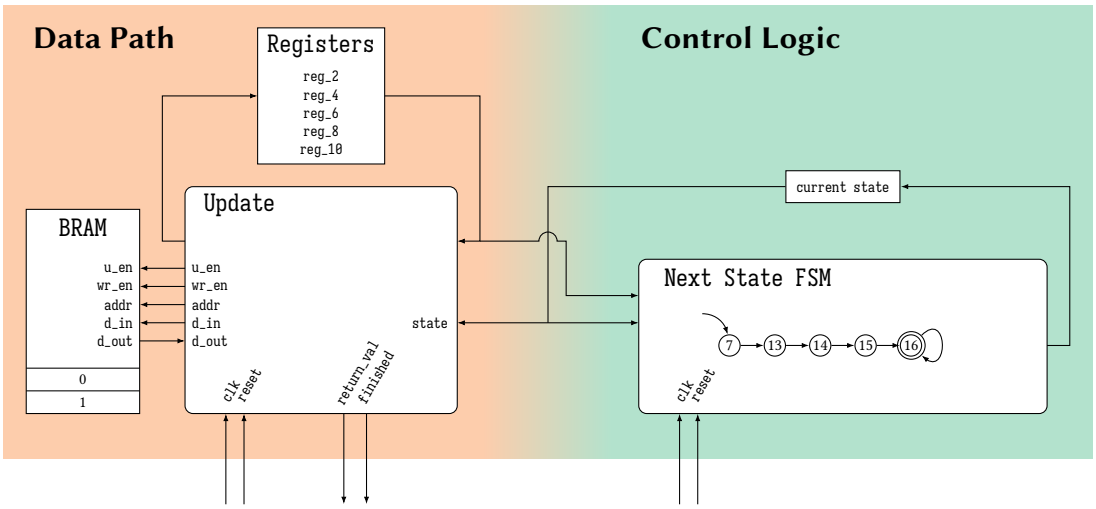


Figure 3.6: The FSMD for the example shown in figure 3.4, split into a data path and control logic for the next state calculation. The update block takes the current state, current values of all registers and at most one value stored in the BRAM, and calculates new values that can either be stored back in the BRAM, in the registers, or can be the next state for the state machine.

interface defined around the unpacked array. Without this interface and without the BRAM block, the synthesis tool processing the Verilog hardware description would not identify the array as a BRAM, and would instead implement it using many registers. A high-level overview of the architecture and of the BRAM interface can be seen in figure 3.6, where loads are specified to take one cycle and stores take half a cycle.

Translating instructions Most RTL instructions correspond to hardware constructs. For example, line 2 which in figure 3.4b shows a 32-bit register `r5` being initialised to 3, after which the control flow moves execution to line 3. This initialisation is also encoded in the Verilog generated from HTL at state 8 in the always-block implementing the FSMD, shown at line 44 in figure 3.7 with the assignment `reg_10 <= 32'd3`. Simple operator instructions are translated in a similar way. For example, the add instruction is just translated to the built-in add operator, similarly for the multiply operator. All 32-bit instructions can be translated in this way, but some special instructions require extra care. One such instruction is the `Oshrximm` instruction, which has no Verilog equivalent and is discussed further in section 6.2.2. Another is the `Oshldimm` instruction, which is a left rotate instruction that has no Verilog equivalent and therefore has to be implemented in terms of other operations and proven to be equivalent. The only 32-bit instructions that are not translated are case-

```
module main(reset, clk, reg_2, finish, return_val);
  output logic [31:0] return_val = 0;
  output logic finish = 0;
  input [31:0] reg_2;
  input clk, reset;

  logic [31:0] reg_8 = 0, reg_4 = 0, d_in = 0;
  logic [31:0] addr = 0, reg_10 = 0, reg_6 = 0, wr_en = 0;
  logic [31:0] d_out = 0, en = 0, u_en = 0;
  logic [31:0] state = 0;

  // BRAM interface
  (* ram_style = "block" *)
  logic [31:0] stack [1:0];
  always @(negedge clk)
    if ({u_en != en}) begin
      if (wr_en) stack[addr] <= d_in;
      else d_out <= stack[addr];
      en <= u_en;
    end

  // Finite-state machine with data path implementation
  always @(posedge clk)
    if ({reset == 32'd1}) state <= 32'd7;
    else case (state)
        32'd21: begin state <= 32'd16; reg_4 <= d_out; end
        32'd16: begin
          finish <= 32'd1; return_val <= reg_4;
          state <= 32'd16; state <= (32'd0 ? 32'd17 : 32'd16);
        end
        32'd15: begin
          state <= 32'd21; u_en <= ( ~ u_en); wr_en <= 32'd0;
          addr <= {{{reg_6 + 32'd0} + {reg_2 * 32'd4}} / 32'd4};
        end
        32'd14: begin
          state <= 32'd15; u_en <= ( ~ u_en); wr_en <= 32'd1;
          d_in <= reg_8; addr <= 32'd1;
        end
        32'd13: begin
          state <= 32'd14; u_en <= ( ~ u_en); wr_en <= 32'd1;
          d_in <= reg_10; addr <= 32'd0;
        end
        32'd7: begin
          reg_10 <= 32'd3; reg_8 <= 32'd6; reg_6 <= 32'd0;
          state <= 32'd13;
        end
        default:;
      endcase
endmodule
```

Figure 3.7: Verilog implementation of RTL code produced by CompCert produced by scheduling the code, instantiating a BRAM and translating to Verilog.

statements (`Ijumptable`) and those instructions related to function calls (`Icall`, `Ibuiltin`, and `Itailcall`), because inlining is enabled by default.

3.3.4 Translating HTL to Verilog

Finally, we have to translate the HTL code into proper Verilog. The challenge here is to translate our FSMD representation into a Verilog AST. However, as all the instructions in HTL are already expressed as Verilog statements, only the top-level data path map needs to be translated to valid Verilog case-statements. We also require declarations for all the variables in the program, as well as declarations of the inputs and outputs to the module, so that the module can be used inside a larger hardware design. In addition to translating the maps of Verilog statements, an always-block implementation the BRAM interface also has to be created, which is only modelled abstractly at the HTL level. Figure 3.7 shows the final Verilog output that is generated for our example. The BRAM interface implementation is shown on lines 12–20 in the Verilog code.

Although this translation seems quite straightforward, proving that this translation is correct is complex. All the implicit assumptions that were made in HTL need to be translated explicitly to Verilog statements and it needs to be shown that these explicit behaviours are equivalent to the assumptions made in the HTL semantics. One main example of this is proving that the specification of the BRAM in HTL does indeed behave in the same way as its Verilog implementation. I discuss these proofs in upcoming sections.

Correctness Theorem and Verilog Semantics

4

This chapter describes the trusted computing base of Vericert, which includes the final correctness theorem and the formalised Verilog semantics.

The trusted computing base of verified software such as CompCert is the code that is not verified, and therefore needs to be trusted by the user when using the tool or the correctness theorem. This is in contrast to the untrusted code that has been verified, and therefore cannot introduce bugs. It is important to understand the trusted computing base of verified software such as CompCert and Vericert to recognise where bugs could still be introduced. When Yang et al. [2011] fuzz tested CompCert and all bugs that were found were in the unverified, trusted parts of the compiler. For example, there were bugs in the unverified front end of CompCert, which was then addressed by reducing the trusted computing base by verifying the correctness of the C parser [Jourdan et al. 2012]. Another bug was found because of a missing constraint in the semantics of the PowerPC.

The trusted computing base of CompCert comprises the Coq theorem prover itself, the OCaml extraction from Coq which is unverified, the front end semantics of C which may not model the C standard faithfully, and finally the semantics for each assembly language [Monniaux and Boulmé 2022]. In general, the trusted computing base of Vericert is the same as that of CompCert, the main two differences are described in this section, and they are the correctness theorem and the target Verilog semantics.

4.1 Formulating the Correctness Theorem

The main correctness theorem is analogous to that stated in CompCert and described in section 2.7.1: for all Clight source programs C, if the translation to the target Verilog code

succeeds, $safe(C)$ holds and the target Verilog has behaviour $\mathcal{B}$ when simulated, then C will have the same behaviour $\mathcal{B}$. The predicate $safe(C)$ means all observable behaviours of C are safe, which can be defined as $\forall \mathcal{B},\ C \Downarrow \mathcal{B} \implies \mathcal{B} \notin \texttt{Wrong}$. A behaviour is in `Wrong` if it can 'go wrong', meaning it contains undefined behaviour. In CompCert, a behaviour is also associated with a trace of I/O events, but since external function calls are not supported in Vericert, this trace will always be empty.

Theorem 4.1. *Whenever the translation from C succeeds and produces Verilog V, and all observable behaviours of C are safe, then V has behaviour $\mathcal{B}$ only if C has behaviour $\mathcal{B}$.*

$$\forall C\ V\ \mathcal{B}.\quad \texttt{HLS}(C) = \lfloor V \rfloor \wedge safe(C) \implies (V \Downarrow \mathcal{B} \implies C \Downarrow \mathcal{B}). \tag{4.1}$$

Why is this correctness theorem also the right one for HLS? It could be argued that hardware inherently runs forever and therefore does not produce a definitive final result. This would mean that the CompCert correctness theorem would probably be unhelpful with proving hardware correctness, as the behaviour would always be divergent. However, in practice, HLS does not normally produce the top-level of the design that connects all the components of the design together, therefore needing to run forever. Rather, HLS often produces smaller components that take an input, execute, and then terminate with an answer. To start the execution of the hardware and to signal to the HLS component that the inputs are ready, the *rst* signal is set and unset. Then, once the result is ready, the *fin* signal is set and the result value is placed in *ret*. This is encoded in the Verilog semantics. The correctness theorem therefore also uses these signals, and the proof shows that once the *fin* flag is set, the value in *ret* is correct according to the semantics of Verilog and CompCert C. Note that the compiler is allowed to fail and not produce any output; the correctness theorem only applies when the translation succeeds.

How can we prove this theorem? Note that like in CompCert, this theorem can be proven using a backward simulation. The same proof technique can be used to verify the correctness of Vericert, extending the forward simulation from RTL to Verilog. Then, by proving that the Verilog semantics are *deterministic*, the composition of forward simulation in CompCert, as well as the simulations in Vericert imply the backward simulation needed to prove the theorem. For each transformation this dissertation will therefore describe how the forward simulations are proven.

All the corollaries proven using the top-level CompCert correctness theorem still hold, and do not have to be proven again or modified, because the correctness theorem is unchanged except for the target language semantics. Note however that Vericert does

not support separate compilation of translation units, and needs a main function. Any corollaries and top-level theorems about the linking of translation units therefore are not needed and hold vacuously, as compilation would fail on these translation units.

4.2 A Formal Semantics for Verilog

This section describes the Verilog semantics that was chosen for the target language, including the changes that were made to the semantics to make it a suitable HLS target. The Verilog standard is quite large [IEEE 2006, 2005], and is split into two standards, one for synthesis and one for simulation. The simulation semantics are similar to semantics one would expect, assigning detailed behaviour to each part of the Verilog language. However, in contrast to programming languages, 'HDLs are better understood as a shorthand for describing digital hardware' (Weste and Harris [2010, p. 699]), as synthesis tools try to understand what kind of hardware the user wanted to describe, which may sometimes behave differently to simulating the original Verilog design. It is therefore important to take that into account in the Verilog semantics and ensure that after synthesis the design still behaves according to the semantics. Luckily, the syntax and semantics can be reduced to a small subset that Vericert can target. This section also describes how Vericert's representation of memory differs from CompCert's memory model.

The Verilog semantics I use is ported to Coq from a semantics written in HOL4 by Lööw and Myreen [2019] which was used to prove the translation from a HOL4 description of the hardware to Verilog [Lööw et al. 2019]. This semantics is quite practical as it is restricted to a small subset of Verilog, which can nonetheless be used to model the hardware constructs required for HLS. The main features that are excluded are continuous assignment and combinational always-blocks; these are modelled in other semantics such as that by Meredith et al. [2010].

The semantics of Verilog is inherently parallel as it needs to describe hardware. The main construct in Verilog is the always-block. A module can contain multiple always-blocks, all of which run in parallel. These always-blocks further contain statements such as if-statements or assignments to variables. Only *synchronous* logic is supported, which means that the always-block is triggered on (and only on) the positive or negative edge of a clock signal. However, combinational expressions can still be expressed within synchronous assignments, so in practice these features are not needed if the Verilog is being generated.

The semantics combines the big-step and small-step styles. The overall execution of the

hardware is described using a small-step semantics, with one small step per clock cycle; this is appropriate because hardware is routinely designed to run for an unlimited number of clock cycles and the big-step style is ill-suited to describing infinite executions. Then, within each clock cycle, a big-step semantics is used to execute all the statements. An example of a rule for executing an always-block that is triggered at the positive edge of the clock is shown below, where Σ is the state of the registers in the module and s is the statement inside the always-block:

$$\textsc{Always}\ \frac{(\Sigma, s) \downarrow_{\text{stmnt}} \Sigma'}{(\Sigma, \texttt{always @(posedge clk)}\ s) \downarrow_{\text{always}^+} \Sigma'} \tag{4.2}$$

This rule says that assuming the statement s in the always-block runs with state Σ and produces the new state Σ', the always-block will result in the same final state. The rule is triggered once for every clock cycle, on the positive edge of the clock (denoted by the $\cdot^+$).

Two types of assignments are supported in always-blocks: nonblocking and blocking assignment. Nonblocking assignments all take effect simultaneously at the end of the clock cycle, while blocking assignments happen instantly so that later assignments in the clock cycle can pick them up. This means that sequential logic is usually expressed using nonblocking assignments, which will often write the assignment to a register that can be read in the next cycle, whereas blocking assignment is used for combinational expressions within the cycle and could therefore be thought of as a wire instead. When writing Verilog, this is only a guideline, and in practice the tools may create registers for blocking assignments if the synthesis tool sees that they are used to store state. To mitigate this, the formal semantics specify that communication between always blocks can only happen through nonblocking assignment, and blocking assignment can only be used for local variables, so that these likely will result in wires in the final hardware. These two restrictions do not only help produce more predictable hardware that behaves like the simulation, but it also simplifies the overall semantics of Verilog. As only clocked always-blocks are supported, communication between these always-blocks can only be performed through nonblocking assignment, which means that the order in which the always blocks are executed does not matter and will always result in the same final state.

To model both of these assignments, the state Σ has to be split into two association maps: Γ, which contains the current values of all variables and arrays, and Δ, which contains the values that will be assigned at the end of the clock cycle. Σ can therefore be defined as

follows: $\Sigma = (\Gamma, \Delta)$. Blocking and Nonblocking assignment can therefore be expressed as follows:

$$\textsc{Blocking Reg}\ \frac{\texttt{name}\ d = \lfloor n \rfloor \qquad (\Gamma, e) \downarrow_{\text{expr}} v}{((\Gamma, \Delta), d\ \texttt{=}\ e\texttt{;}) \downarrow_{\text{stmnt}} (\Gamma[n \mapsto v], \Delta)} \qquad \textsc{Nonblocking Reg}\ \frac{\texttt{name}\ d = \lfloor n \rfloor \qquad (\Gamma, e) \downarrow_{\text{expr}} v}{((\Gamma, \Delta), d\ \texttt{<=}\ e\texttt{;}) \downarrow_{\text{stmnt}} (\Gamma, \Delta[n \mapsto v])} \tag{4.3}$$

where assuming that $\downarrow_{\text{expr}}$ evaluates an expression e to a value v, the nonblocking assignment d `<=` e updates the future state of the variable d with value v.

Finally, the following rules dictate how the whole module runs in one clock cycle:

$$\textsc{Module}_1^+\ \frac{(\Gamma, \Delta, a) \downarrow_{\text{always}^+} (\Gamma', \Delta') \qquad (\Gamma', \Delta', \vec{m}) \downarrow_{\text{module}^+} (\Gamma'', \Delta'')}{(\Gamma, \Delta, a :: \vec{m}) \downarrow_{\text{module}^+} (\Gamma'', \Delta'')} \qquad \textsc{Module}_2^+\ \frac{}{(\Gamma, \Delta, \varnothing) \downarrow_{\text{module}^+} (\Gamma, \Delta)} \tag{4.4}$$

$$\textsc{Program}\ \frac{(\Gamma, \epsilon, \vec{m}) \downarrow_{\text{module}^+} (\Gamma', \Delta')}{(\Gamma, \texttt{module main(...);}\ \vec{m}\ \texttt{endmodule}) \downarrow_{\text{program}} (\Gamma'\ //\ \Delta')} \tag{4.5}$$

where Γ is the initial state of all the variables, ϵ is the empty map because the Δ map is assumed to be empty at the start of the clock cycle, and $\vec{m}$ is a list of variable declarations and always-blocks that $\downarrow_{\text{module}^+}$ evaluates sequentially to obtain (Γ', Δ'). The final state is obtained by merging these maps using the // operator, which gives priority to the right-hand operand in a conflict. This rule ensures that the nonblocking assignments overwrite at the end of the clock cycle any blocking assignments made during the cycle.

4.2.1 Changes to the semantics

Five changes were made to the semantics proposed by Lööw and Myreen [2019] to make it suitable as an HLS target.

Adding array support The main change is the addition of support for arrays, which are needed to model BRAM in Verilog. BRAM is needed to model the stack in C efficiently, without having to declare a variable for each possible stack location. In the original semantics, RAMs (as well as inputs and outputs to the module) could be modelled using a function from variable names to values, which could be modified accordingly to model

inputs to the module. However, this representation does not support merging of individual array elements. This is quite an abstract description of memory and can also be expressed as an array instead, which is the path I took. This requires the addition of array operators to the semantics and correct reasoning of loads and stores to the array in different always-blocks simultaneously. Consider the following Verilog code:

```
1 logic [31:0] x[1:0];
2 always @(posedge clk) begin
3   x[0] = 1;
4   x[1] <= 1;
5 end
```

This always-block modifies one array element using blocking assignment and then a second using nonblocking assignment. If the existing semantics were used to update the array, then during the merge, the entire array `x` from the nonblocking association map would replace the entire array from the blocking association map. This would replace `x[0]` with its original value and therefore behave incorrectly. Accordingly, I modified the maps so they record updates on a per-element basis. Our state Γ is therefore further split up into Γ_r for instantaneous updates to variables, and Γ_a for instantaneous updates to arrays ($\Gamma = (\Gamma_r, \Gamma_a)$); Δ is split similarly ($\Delta = (\Delta_r, \Delta_a)$). The merge function then ensures that only the modified indices get updated when Γ_a is merged with the nonblocking map equivalent Δ_a.

The implementation of arrays is done using dependently typed arrays that keep track of their size, such that out-of-bounds accesses can be detected and handled appropriately. Section 4.2.3 describes the implementation of this memory model in more detail, comparing it to the CompCert memory model.

Adding negative edge support To reason about circuits that execute on the negative edge of the clock (such as our BRAM interface described briefly in section 3.3.3), support for negative-edge-triggered always-blocks was added to the semantics. This is shown in the modifications of the PROGRAM$^{\pm}$ rule shown below:

$$\frac{\textsc{Program}^{\pm} \quad (\Gamma, \epsilon, \vec{m}) \downarrow_{\text{module}^+} (\Gamma', \Delta') \qquad (\Gamma' \mathbin{//} \Delta', \epsilon, \vec{m}) \downarrow_{\text{module}^-} (\Gamma'', \Delta'')}{(\Gamma, \texttt{module main(...); } \vec{m} \texttt{ endmodule}) \downarrow_{\text{program}^{\pm}} (\Gamma'' \mathbin{//} \Delta'')} \tag{4.6}$$

The main execution of the module $\downarrow_{\text{module}}$ is split into $\downarrow_{\text{module}^+}$ and $\downarrow_{\text{module}^-}$, which are rules that only execute always-blocks triggered at the positive and at the negative edge

respectively. The positive-edge-triggered always-blocks are processed in the same way as in the original PROGRAM rule. The output maps Γ' and Δ' are then merged and passed as the blocking assignments map into the negative edge execution, so that all the blocking and nonblocking assignments are resolved. Finally, all the negative-edge-triggered always-blocks are processed and merged to give the final state.

Adding declarations Explicit support for declaring inputs, outputs and internal variables was added to the semantics to make sure that the generated Verilog also contains the correct declarations. This adds some guarantees to the generated Verilog and ensures that it synthesises and simulates correctly. Otherwise, if there are bugs in the insertion of declarations, the Verilog would fail to synthesise or simulate. This is done by adding a declaration entry for each register that is used.

Removing support for external inputs to modules Support for receiving external inputs was removed from the semantics. The main module in Verilog models the main function in C, and since the inputs to a C function should not change during its execution, there is no need for external inputs for Verilog modules. In addition to that, this external inputs function was also used to model memory in the original semantics, which is not needed as arrays are fully supported. This means that the BRAM can be faithfully modelled as Verilog and it can be reasoned about.

Simplifying representation of bitvectors Finally, I use 32-bit integers to represent bitvectors rather than arrays of Booleans. This is because Vericert (currently) only supports types represented by 32 bits. However, extending Vericert would likely not require the full flexibility of bitvectors as arrays of Booleans either, as long as fine-grained bit-size optimisations for registers are not introduced, because the Verilog subset could be restricted to bitvectors of sizes 4, 8, 16, 32 and 64, which are already supported by the CompCert integer type.

4.2.2 Integrating the Verilog semantics into CompCert's model

The CompCert computation model defines a set of states through which execution passes. In this subsection, I explain how I extend our Verilog semantics with four special-purpose registers in order to integrate it into CompCert.

$$\textsc{Step}\quad \frac{\Gamma_r[rst] = 0 \qquad \Gamma_r[fin] = 0 \qquad (m, (\Gamma_r, \Gamma_a)) \downarrow_{\text{program}^{\pm}} (\Gamma_r', \Gamma_a')}{\texttt{State}\ sf\ m\ \Gamma_r[st]\ \Gamma_r\ \Gamma_a \longrightarrow \texttt{State}\ sf\ m\ \Gamma_r'[st]\ \Gamma_r'\ \Gamma_a'}$$

$$\textsc{Finish}\quad \frac{\Gamma_r[fin] = 1}{\texttt{State}\ sf\ m\ n\ \Gamma_r\ \Gamma_a \longrightarrow \texttt{Returnstate}\ sf\ \Gamma_r[ret]}$$

$$\textsc{Call}\quad \frac{}{\texttt{Callstate}\ sf\ m\ \vec{r} \longrightarrow \texttt{State}\ sf\ m\ n\ ((\texttt{init\_params}\ \vec{r}\ a)[st \mapsto n, fin \mapsto 0, rst \mapsto 0])\ \epsilon}$$

$$\textsc{Return}\quad \frac{}{\texttt{Returnstate}\ (\texttt{Stackframe}\ r\ m\ pc\ \Gamma_r\ \Gamma_a :: sf)\ v \longrightarrow \texttt{State}\ sf\ m\ pc\ (\Gamma_r[st \mapsto pc, r \mapsto v])\ \Gamma_a}$$

Figure 4.1: Top-level small-step semantics for Verilog modules in CompCert's computational framework.

CompCert executions pass through three main states, which have been adapted to the Verilog semantics by using association maps for the registers and arrays. These three states otherwise match the states used by other languages, simplifying the integration of the Verilog semantics into CompCert. These states, together with the transitions defined on the states, act as a weak calling convention for the hardware produced by the HLS tool. These are therefore not part of the general Verilog semantics themselves, but part of the Verilog semantics for designs produced through HLS.

`State` $sf\ m\ v\ \Gamma_r\ \Gamma_a$ The main state when executing a function, with stack frames sf, the top-level module m, current state v and variable states Γ_r and Γ_a. The regular execution of the module will proceed from `State` to `State` until the *fin* signal is high.

`Callstate` $sf\ m\ \vec{r}$ The state that is reached when a function is called, with the stack frames sf, the top-level module m and arguments $\vec{r}$. In practice, as there are no function calls in the Verilog semantics, this state is only reached at the start of the execution. A better name for this state might therefore be `Resetstate`, as it corresponds to the behaviour of the module when the reset input was asserted.

`Returnstate` $sf\ v$ The state that is reached when a function returns back to the caller, with stack frames sf and return value v.

To support this computational model, I extend the Verilog module I generate with the following four registers and a BRAM block:

program counter The program counter is modelled using a register that keeps track of the state, named *st*. This register should always contain the current state, and will be checked on the next transition to pick the next state.

function entry point When a function is called, the entry point denotes the first instruction that will be executed. This can be modelled using a reset signal that sets the state accordingly, denoted as *rst*. In this translation, function calls are not implemented, so the reset signal resets the main module and sets the state parameter for this main module.

return value The return value can be modelled by setting a finished flag to 1 when the result is ready, and putting the result into a 32-bit output register. These are denoted as *fin* and *ret* respectively. Instead of a return instruction, checking that *fin* is 1 should signal that the result is stored in the return register.

stack The function's stack frame can be modelled as a BRAM block, which is implemented using an array in the module, and denoted as *stk*. When the main function is initially called, it allocates a stack frame, which should match the BRAM block that was implemented.

Figure 4.1 shows the inference rules for moving between the computational states. The first, Step, is the normal rule of execution. It defines one step in the `State` state, assuming that the module is not being reset, that the finish state has not been reached yet, that the current and next state are v and v', and that the module runs from state Γ to Γ' using the Step rule. This rule also ensures that the state register contains the value of the current state, by checking the value of $\Gamma_r[st]$. It is then ensured that the resulting state is also the value of the *st* register in the updated association map $\Gamma'_r[st]$.

The Finish rule returns the final value of running the module and is applied when the *fin* register is set; the return value is then taken from the *ret* register.

Note that there is no step from `State` to `Callstate`; this is because function calls are not supported, and it is therefore impossible in our semantics ever to reach a `Callstate` except for the initial call to main. So the Call rule is only used at the very beginning of execution; likewise, the Finish rule is only matched for the final return value from the main function. As a result, the final rule called Return is never reached in practice, as the

stack will always be empty. This rule could ideally therefore be removed, however it is still present to match the state transitions used in CompCert. To remove this rule, one would have to show that the stack remains empty by induction over the semantics.

In addition to these rules, a valid semantics also needs an initial state and a final state. These are therefore defined as follows:

$$\textsc{Initial}\ \frac{\texttt{is\_internal}\ P.\texttt{main}}{\texttt{initial\_state}\ (\texttt{Callstate}\ []\ P.\texttt{main}\ [])} \qquad \textsc{Final}\ \frac{}{\texttt{final\_state}\ (\texttt{Returnstate}\ []\ n)\ n}$$

where the initial state is the `Callstate` with an empty stack and no arguments for the `main` function of program P, where this `main` function needs to be in the current translation unit. The final state results in the program output of value n when reaching a `Returnstate` with an empty stack.

4.2.3 Memory model

The Verilog semantics do not define a memory model for Verilog, as this is not needed for a hardware description language. There is no preexisting architecture that Verilog will produce; it can describe any memory layout that is needed. Instead of having a specific semantics for memory, the Verilog semantics only needs to support the language features that can produce these different memory layouts, these being Verilog arrays. I therefore define semantics for updating Verilog arrays using blocking and nonblocking assignment. This makes it possible to describe many different memory layouts in Verilog, providing a lot of flexibility. I then have to develop a memory layout that will be targeted by the HLS tool and prove that the C memory model that CompCert uses matches the interpretation of arrays used in Verilog together with the memory layout that was chosen. The CompCert memory model is infinite, whereas our representation of arrays in Verilog is inherently finite. There have already been efforts to define a general finite memory model for all intermediate languages in CompCert, such as CompCertS [Besson et al. 2018] or CompCert-TSO [Ševčík et al. 2013], or keeping the intermediate languages intact and translating to a more concrete finite memory model in the back end, such as in CompCertELF [Wang et al. 2020]. I also define such a translation from CompCert's standard infinite memory model to finite arrays that can be represented in Verilog. There is therefore no more notion of an abstract memory model and all the interactions to memory are encoded in the hardware itself.

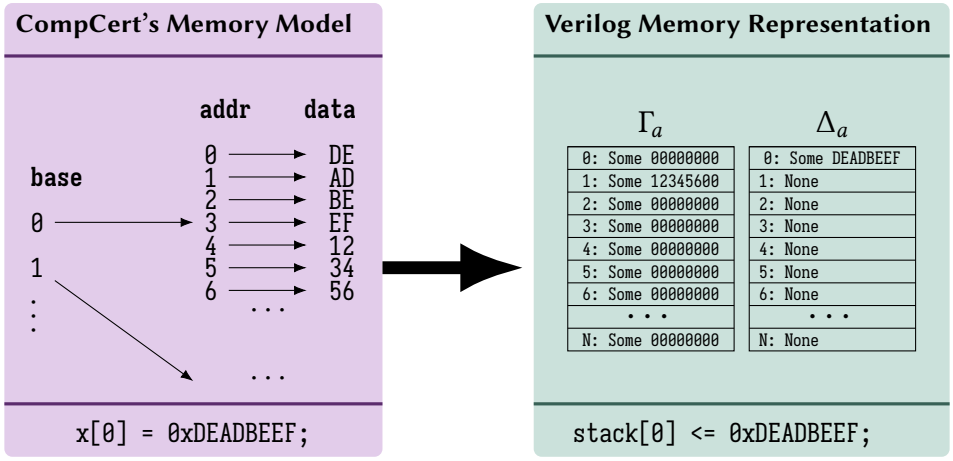


Figure 4.2: Change in the memory model during the translation of RTL into HTL. The state of the memories in each case is recorded straight after the execution of the store to memory.

This translation is represented in figure 4.2. CompCert's memory model is represented as a map from blocks to maps from memory addresses to memory contents. Each block represents an area in memory; for example, a block can represent a global variable or a stack for a function. CompCert also includes permissions that are associated with each block, which are lost in the translation to arrays. In contrast, the array has a static size, which has to be the same size as the writable addresses in CompCert's memory.

Because there are no global variables in the C code, only the stack of the main function will be allocated in the memory. Our Verilog semantics defines two finite arrays of optional values, one for the blocking assignments map Γ_a and one for the nonblocking assignments map Δ_a. The optional values are present to ensure correct merging of the two association maps at the end of the clock cycle, so that information about which cells of the array were modified is present. The invariant used in the proofs is that block associated with the function stack should be equivalent to the merged representation of the Γ_a and Δ_a maps.

In particular, note that the Verilog arrays are word-addressable instead of the CompCert memory model which is byte-addressable. The instruction set architectures of CPUs typically specify memory as being byte-addressable, however, the implementation of the memory is not that straightforward. CPUs usually implement multiple levels of caching as well as the conversion from virtual addresses to physical addresses and may operate on byte- or word-addressable memories behind the byte-addressable interface. Often, the performance of the caches will be tuned to provide the best performance for reading and writing words to memory. This makes it possible to have good general performance,

and makes it possible to interact with larger, but slower memories such as dynamic random-access memory (DRAM). However, Vericert assumes that it is targeting an FPGA without external DRAM, so only fast BRAM memory is used. This means that the memory architecture can be specialised to work well with words without using a complex memory architecture, by directly using a word-addressable BRAM. A single BRAM is enough to support this subset of C that is translated to Verilog.

4.2.4 Deterministic Verilog semantics

Finally, to integrate the Verilog semantics into CompCert's backward simulation framework, we need to show that the Verilog semantics is deterministic. This result allows us to replace the forward simulations I have proved with the desired backward simulations. This was proven for the small-step semantics shown in figure 4.1.

Lemma 4.2. *If a Verilog program V admits behaviours B_1 and B_2, then B_1 and B_2 must be the same.*

$$\forall V\ B_1\ B_2. \quad V \Downarrow B_1 \land V \Downarrow B_2 \implies B_1 = B_2. \tag{4.7}$$

Proof sketch. The Verilog semantics is deterministic because the order of operation of all the constructs is defined, so there is only one way to evaluate the module, and hence only one possible behaviour. In particular, non-determinism in the simulation of Verilog designs often comes from the fact that always-blocks are evaluated in an arbitrary order. However, as I ensure that always-blocks only communicate using nonblocking assignment, the order of evaluation does not change the final state, meaning a sequential evaluation of always-blocks can be chosen. □

4.3 Summary

In conclusion, the trusted computing base of Vericert is similar to that of CompCert. The correctness theorem can remain unchanged, except for the change of the target language semantics. The Verilog semantics can be formalised within CompCert's semantic framework. I reuse an existing semantics by Lööw and Myreen [2019] and modify it to be a suitable HLS target, in addition to integrating the semantics into CompCert's general state transition framework for its intermediate languages.

5 Verified Hyperblock Scheduling

This chapter describes the hyperblock scheduling pass in Vericert, which collects operations into groups that can be executed in parallel. This chapter is based on a paper coauthored with John Wickerson which has been submitted to PLDI 2024 [Herklotz and Wickerson 2024].

Many approaches to scheduling have been proposed over the years. Some, such as *list scheduling* [Baker 2019, p. 257] only reorder instructions within a basic block. This means they squander opportunities for performance improvements that could be obtained by reordering instructions across branches. A more powerful alternative is *trace scheduling* [Ellis 1985; Fisher 1981], which works by creating paths (or 'traces') through the code, across basic block boundaries, and then reorders the instructions within those paths. In its most general form, trace scheduling is considered infeasible on large programs, but two special cases called *superblock scheduling* [Hwu et al. 1993] and *hyperblock scheduling* [Mahlke et al. 1992] have been developed, both of which impose restrictions on the form of traces in order to obtain tractable algorithms. Superblocks generalise basic blocks by allowing early exits, while hyperblocks generalise superblocks by additionally allowing each instruction to be *predicated*. CPUs often lack support for predicated execution, so superblock scheduling is the natural choice in that setting. But in custom hardware, predicated execution can be implemented efficiently, making hyperblock scheduling a natural fit for HLS. Indeed, the use of hyperblock scheduling in HLS was first proposed over two decades ago [Budiu and Goldstein 2002; Callahan and Wawrzynek 1998], and is nowadays used by popular HLS tools such as AMD Vitis HLS [AMD 2023a], LegUp [Canis 2015, p. 60], Google XLS [Google 2023, line 112], and Bambu [Ferrandi 2014, line 304]. Hyperblock scheduling is therefore a natural choice for Vericert to close the gap between the verified and unverified HLS tools, and model what existing tools are already doing. Support for the scheduling of predicated instructions is also important for future HLS-specific optimisations, such as loop scheduling, because these often assume that basic blocks can be merged using

if-conversion.

In this chapter, I present:

- **the first verified implementation of hyperblock scheduling**, which is more general than Six et al.'s verified superblock scheduler [Six et al. 2022] and more computationally tractable than Tristan and Leroy's verified trace scheduler [Tristan and Leroy 2008],
- **the first verified implementation of general if-conversion** (a pre-scheduling pass that turns if-statements into hyperblocks), building on CompCert's naïve if-converter that only handles simple cases [AbsInt 2019], and
- **a novel use of a verified SAT solver during translation validation** in order to reason about predicates.

First, section 5.1 gives an overview of the scheduling optimisation, then section 5.2 introduces the new intermediate languages used for the scheduling transformation. Section 5.3 then describes hyperblock construction using if-conversion followed by sections 5.4 to 5.6 describing the implementation, validation and verification of the actual hyperblock transformation. Finally, section 5.8 describes the use of a validated SMT solver as part of the hyperblock scheduling correctness proof.

5.1 Overview

Making hardware designs parallel is actually not the challenging part of our work – Verilog synthesis tools do this implicitly. Indeed, it is easy to write Verilog so that an arbitrary sequence of C operations is mapped to a single piece of combinational logic that executes in just one clock cycle. But the problem with such unfettered parallelism is that these large pieces of combinational logic may have long critical paths, and thus lead to hardware that can only run at a low clock frequency. The actual challenge, then, is producing the *right amount* of parallelism.

This is a scheduling problem. Vericert must decide how to schedule each block of instructions across one or more clock cycles so that when the downstream Verilog synthesis tool performs parallelisation, the resulting combinational paths will be short enough to meet the target frequency.

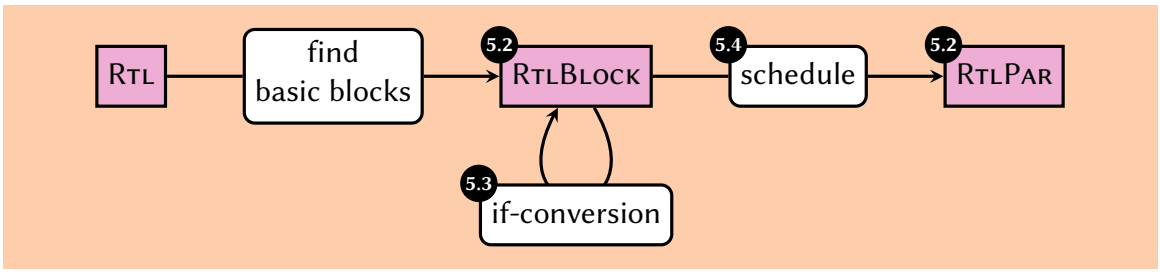


Figure 5.1: New passes and intermediate languages introduced in this work.

To clarify: the downstream synthesis tool is responsible for the parallelisation itself – Vericert does not perform it, nor does it guarantee its soundness. Rather, it predicts the parallelisation that the synthesis tool will perform, and schedules so that if parallelisation is performed as predicted, the target frequency should be met. What Vericert *does* guarantee is that if it reorders any instructions as part of the scheduling process, these reorderings preserve the program's (sequential) behaviour.

Figure 5.1 shows the main components of the hyperblock scheduler. First I find the basic blocks in RTL and form RTLBLOCK, which structures the basic blocks as lists of instructions.

Vericert then performs if-conversion [Allen et al. 1983]. This is a transformation that merges basic blocks from two sides of a fork in the CFG into a single, larger basic block (now called a hyperblock) that uses predication to control which instructions are executed. If-conversion is helpful because larger blocks can give the scheduler more opportunities to find parallelism. Figure 5.2 gives an example of multiple RTL basic blocks being merged into a single hyperblock, where the if-statement is turned into an assignment. Note that basic block was duplicated with two different predicate conditions, because it was if-converted twice, once per branch of the if-statement. CompCert already has an if-conversion pass [AbsInt 2019], but it can only handle simple cases such as replacing `if(c) {x = a;} else {x = b;}` with `x = c ? a : b`, whereas our implementation can handle arbitrary forks in the CFG. If-conversion can be applied selectively, using heuristics to judge which basic blocks should be combined. Having proved if-conversion correct wherever it is applied, these heuristics can be adjusted with no impact on the correctness proof.

Next the hyperblocks are scheduled in turn. The scheduler takes the list of predicated instructions and produces a list of groups of instructions such that all the instructions in the same group can be safely executed in parallel. Actually, our scheduler produces a list of groups of *lists of* instructions. The idea behind this three-level representation, which I call RTLPAR, is that each inner list is a sequence of instructions that can be chained

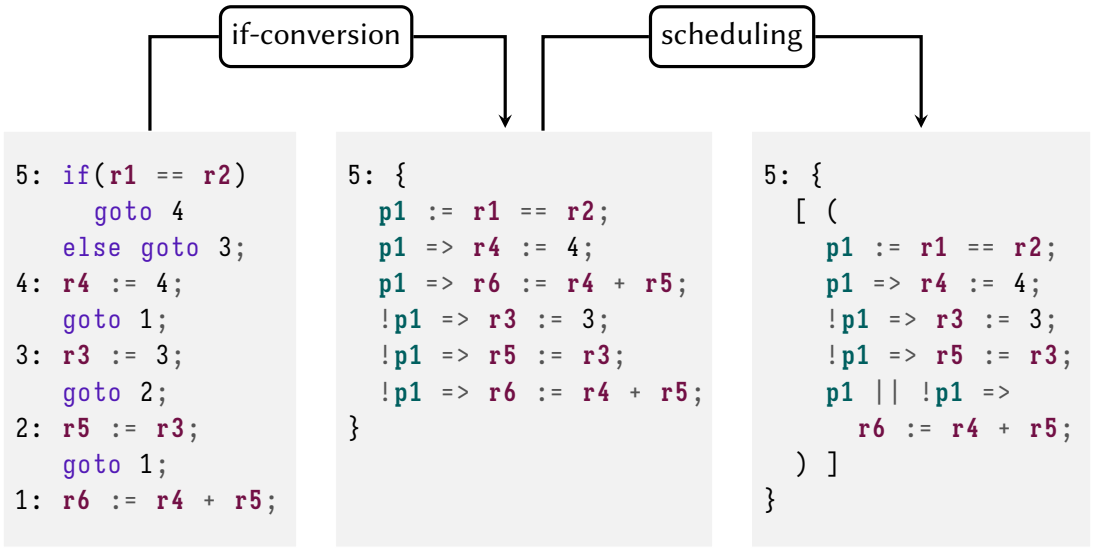


Figure 5.2: Example of an if-conversion transformation performed in Vericert, followed by a scheduling operation, after which the final node in the CFG can be shared.

together, then each group contains instruction chains that can be executed in parallel. In this way, Vericert supports *operation chaining*, a long-established optimisation in hardware design [Pangrle and Gajski 1987, p. 1101]. This is shown in the third hyperblock in figure 5.2, where all instructions are actually chained, so are placed in the inner list, because the overall estimated latency is less than half a clock period. Note here that the scheduler can share the duplicated instruction in the original basic block, and it could furthermore remove the predicate from the instruction completely as it is executed unconditionally. It can therefore rectify the duplication that is inserted by the if-conversion transformation.

What remains is to ensure the correctness of each schedule. Following previous work on verified scheduling by Tristan and Leroy [2008] and Six et al. [2022], I use translation validation, but dealing with hyperblocks brings additional complexity, as explained below.

Tristan and Leroy implement trace scheduling in its full generality. They use a tree to represent all the possible control-flow paths through a block. These trees can be 'exponentially larger than the original code' (Tristan and Leroy [2008, p. 25]), which makes it prohibitively expensive to construct and compare the trees before and after scheduling, and thus undermines the usefulness of their scheduler.

Six et al. restrict their scheduler to superblocks. Since a superblock has only a single control-flow path (with early exits), the need for trees is avoided. This allows their validator to be 'efficient even for large superblocks' (Six et al. [2022, p. 53]). However, superblocks are less general than hyperblocks, and there are code patterns where superblock scheduling can

lead to considerable code duplication that hyperblocks would avoid. Moreover, superblock scheduling is reliant on profiling and branch-prediction heuristics to pick a hot path through the program – should such a hot path even exist.

Hyperblocks can branch and merge control flow using predicated execution, and hence a single hyperblock can capture many control-flow paths without the exponential blow-up that Tristan and Leroy encountered. In particular, hyperblocks can handle well the case where two branches of a conditional statement are executed equally often, unlike the superblocks that Six et al. use. However, our task of validating the equivalence of two hyperblocks is complicated by having to reason about predicates. Where the prior works only needed to check that the scheduled block contains a dependency-respecting permutation of the original block's instructions, the validator must account for the fact that predicates may be modified during scheduling. For instance, the sequence `p => i; !p => i`, which executes `i` if `p` holds and then executes `i` if `p` does *not* hold, may be optimised to `i`.

The approach I take is to translate both the RTLBLOCK hyperblock and the RTLPAR hyperblock (i.e., before and after scheduling) into their strongest postconditions, starting from the same symbolic initial state, and then comparing these postconditions for equivalence with the help of a SAT solver that I have programmed and verified in Coq. By being able to solve queries like $(\texttt{p} \wedge \texttt{!p}) \leftrightarrow \mathit{false}$, the SAT solver enables reasoning about reordering of instructions in a predicate-aware fashion.

5.2 New Intermediate Languages

Our work introduces two new intermediate languages: RTLBLOCK and RTLPAR, which implement the sequential and scheduled semantics of hyperblocks respectively. They are based on CompCert's RTL, but instead of mapping from states to instructions, RTLBLOCK maps from states to hyperblocks, and RTLPAR maps from states to hyperblocks of instructions grouped into specific cycles.

Hyperblocks are made up of instructions as defined in figure 5.3, where $\vec{\cdot}$ denotes a list and $\cdot^?$ denotes an optional parameter. Most instructions are similar to their RTL counterparts, except each instruction is now guarded by an optional predicate. One additional instruction is for setting a predicate (p) equal to an evaluated condition ($r_1\ op_c\ r_2$). The other new instruction is `E`, which takes a control-flow instruction (I_{cf}) and allows for early exit from the hyperblock.

These instructions are used in both RTLBLOCK and RTLPAR. The main difference between

registers:	r, r1, r2, ... $\in \mathbb{r}$	
predicates:	p, p1, p2, ... $\in \mathbb{p}$	
CFG node labels:	$L \in \mathbb{L} ::= \mathbb{N}$	
guard expressions:	$G \in \mathbb{G} ::= \mathbb{p} \mid \neg\mathbb{p} \mid \mathit{true} \mid \mathit{false} \mid \mathbb{G} \wedge \mathbb{G} \mid \mathbb{G} \vee \mathbb{G}$	
arithmetic ops:	$op_{\text{a}} \in \mathbb{a} ::= \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \ldots$	
conditional ops:	$op_{\text{c}} \in \mathbb{c} ::= \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \ldots$	
addressing modes:	$d \in \mathbb{d} ::= \texttt{Stack} \mid \texttt{Global} \mid \texttt{M} \mid \ldots$	
instructions:	$I \in \mathbb{I} ::= \texttt{skip}$	(no-op)
	$\mid \mathbb{G} \texttt{ => } \mathbb{r} \texttt{ := } \mathbb{r}\ \mathbb{a}\ \mathbb{r}$	(arith/logical op)
	$\mid \mathbb{G} \texttt{ => } \mathbb{r} \texttt{ := } \mathbb{d}\texttt{[}\mathbb{r}\texttt{]}$	(memory load)
	$\mid \mathbb{G} \texttt{ => } \mathbb{d}\texttt{[}\mathbb{r}\texttt{]} \texttt{ := } \mathbb{r}$	(memory store)
	$\mid \mathbb{G} \texttt{ => } \mathbb{p} \texttt{ := } \mathbb{r}\ \mathbb{c}\ \mathbb{r}$	(assign predicate)
	$\mid \mathbb{G} \texttt{ => E}(\mathbb{I}_{\text{cf}})$	(block exit)
control-flow instructions:	$I_{\text{cf}} \in \mathbb{I}_{\text{cf}} ::= \texttt{if } (\mathbb{r}\ \mathbb{c}\ \mathbb{r})\ \mathbb{L}\ \mathbb{L}$	(conditional)
	$\mid \texttt{goto } \mathbb{L}$	(goto node)
	$\mid \texttt{call } \mathit{sig}\ f\ \vec{\mathbb{r}}\ \mathbb{r}\ \mathbb{L}$	(function call)
	$\mid \texttt{tailcall } \mathit{sig}\ f\ \vec{\mathbb{r}}$	(tailcall)
	$\mid \texttt{builtin } f_{ext}\ \vec{\mathbb{r}}\ \mathbb{r}\ \mathbb{L}$	(builtin function)
	$\mid \texttt{jumptable } \mathbb{r}\ \vec{\mathbb{L}}$	(jumptable)
	$\mid \texttt{return } \mathbb{r}^?$	(function return)
	$H \in \textsc{RtlBlock} ::= I\ \texttt{list}$	
	$H_{\text{par}} \in \textsc{RtlPar} ::= I\ \texttt{list list list}$	

Figure 5.3: Syntax of RtlBlock and RtlPar, with our hyperblock additions highlighted.

these two languages is how these instructions are arranged within the hyperblock and the execution semantics of the hyperblock. An RtlBlock hyperblock is a list of instructions, with a straightforward sequential semantics. An RtlPar hyperblock is a list of lists of lists of instructions, with nested blocks corresponding to where instructions should be placed in hardware. Each innermost list contains a chain of instructions that can be executed sequentially within a single clock cycle; each middle list contains a group of chains that can be executed in parallel; and the outermost list contains groups to be executed sequentially (in consecutive clock cycles).

The existing CompCert semantics for Rtl is a small-step operational semantics defined on a CFG. At each step, the instruction in the CFG at the current program counter is evaluated in a context Γ. This is a 3-tuple comprising an environment Γ_{Env} that has global information about the program, a mapping Γ_{R} from registers to values, and a mapping Γ_{M} from memory addresses to values.

In order to give a semantics for RtlBlock and RtlPar, we need to handle predicates, so we turn Γ into a 4-tuple $(\Gamma_{\text{Env}}, \Gamma_{\text{R}}, \Gamma_{\text{P}}, \Gamma_{\text{M}})$, where the additional component Γ_{P} maps predicates to Booleans. Moreover, we need to deal with CFGs where each node is not just a single instruction, but a hyperblock. The semantics, which we provide in figure 5.4, is denoted by $\Gamma \vdash a \Downarrow_A b$ and states that under the context Γ and using the definition A, a will be evaluated to b. It is a big-step semantics in the sense that it executes an entire hyperblock in a single step. The ExecInstr rule executes an arithmetic instruction if its guard evaluates to true (and there are similar rules for the other guarded instructions). Here we write $\Gamma \vdash a \downarrow b$ for an evaluation function for operations: given Γ and a it computes b. The ExecInstrFalse rule handles the case where the guard does not hold. ExecExit handles the execution of an exit instruction by recording the control-flow instruction I_{cf} for leaving the block. The next two rules are for executing an instruction list, with BlockContinue handling the case where the head instruction does not exit the block, and BlockExit handling the case where it does. Finally, ExecRtlBlock and ExecRtlPar provide the semantics of RtlBlock and RtlPar blocks respectively. Both languages use the list execution semantics defined by the above rules, but RtlPar blocks are first flattened into a single list (via `concat`).

Note that these rules define a sequential semantics for RtlPar. That is, although RtlPar blocks contain lists of instruction chains that have been identified by the scheduler as being suitable for parallel execution, our semantics nonetheless executes them sequentially. This is because although the scheduler identifies where parallelism can be profitably extracted,

$$\textsc{ExecInstr}\ \frac{\Gamma_P \vdash G \downarrow \mathit{true} \qquad \Gamma \vdash \texttt{r1}\ \mathit{op}_a\ \texttt{r2} \downarrow v}{\Gamma \vdash (G\ \texttt{=>}\ \texttt{rd}\ \texttt{:=}\ \texttt{r1}\ \mathit{op}_a\ \texttt{r2}) \Downarrow_{\mathbb{I}} ((\Gamma_R[\texttt{rd} \mapsto v], \Gamma_P, \Gamma_M), \texttt{None})}$$

$$\textsc{ExecInstrFalse}\ \frac{\Gamma_P \vdash G \downarrow \mathit{false}}{\Gamma \vdash (G\ \texttt{=>}\ \_) \Downarrow_{\mathbb{I}} (\Gamma, \texttt{None})}$$

$$\textsc{ExecExit}\ \frac{\Gamma_P \vdash G \downarrow \mathit{true}}{\Gamma \vdash (G\ \texttt{=>}\ \texttt{E}(I_{cf})) \Downarrow_{\mathbb{I}} (\Gamma, \lfloor I_{cf} \rfloor)}$$

$$\textsc{BlockContinue}\ \frac{\Gamma \vdash I \Downarrow_{\mathbb{I}} (\Gamma', \texttt{None}) \qquad \Gamma' \vdash l \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma'', I_{cf})}{\Gamma \vdash I :: l \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma'', I_{cf})}$$

$$\textsc{BlockExit}\ \frac{\Gamma \vdash I \Downarrow_{\mathbb{I}} (\Gamma', \lfloor I_{cf} \rfloor)}{\Gamma \vdash I :: l \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma', I_{cf})}$$

$$\textsc{ExecRtlBlock}\ \frac{\Gamma \vdash H \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma', I_{cf})}{\Gamma \vdash H \Downarrow_{\textsc{RtlBlock}} (\Gamma', I_{cf})}$$

$$\textsc{ExecRtlPar}\ \frac{\Gamma \vdash \texttt{concat}(\texttt{concat}\ H_{par}) \Downarrow_{\texttt{list}(\mathbb{I})} (\Gamma', I_{cf})}{\Gamma \vdash H_{par} \Downarrow_{\textsc{RtlPar}} (\Gamma', I_{cf})}$$

Figure 5.4: Semantics of RtlBlock and RtlPar hyperblocks, where Γ is a 4-tuple $(\Gamma_{Env}, \Gamma_R, \Gamma_P, \Gamma_M)$.

the program does not actually become parallel until the final 'forward substitution' pass, where Verilog blocking assignments become nonblocking assignments. Hence, to avoid unnecessary complexity, the semantics are kept sequential at the RtlPar stage. (A parallel RtlPar semantics may allow more optimisations to be validated, but we save that for future work.)

The overall behaviour $\mathcal{B}$ of an RtlBlock or RtlPar program is the same as that of existing CompCert intermediate languages. Either the program terminates with a return value and a finite trace of externally visible events, like external I/O events which can be produced by system calls (part of the I_{cf} instructions), or the program diverges with a possibly infinite trace of external events. Note that the top-level correctness theorem of Vericert does not include a trace of externally visible events, because the only external behaviour of the hardware is the final return value; nevertheless, until the hardware is generated, instructions emitting external behaviours are kept and reasoned about.

5.3 Verified If-Conversion

If-conversion introduces the predicated instructions that form hyperblocks. CompCert does have an if-converter already, but it applies only in a few special cases. We need a more

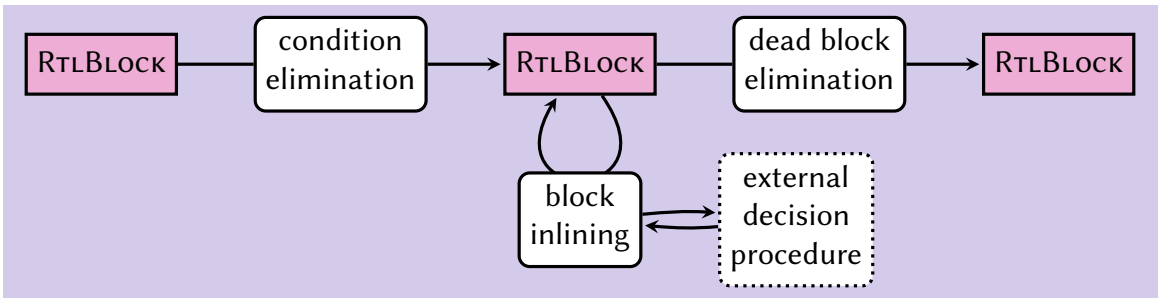


Figure 5.5: Details of the if-conversion pass, showing the three different stages of the transformation.

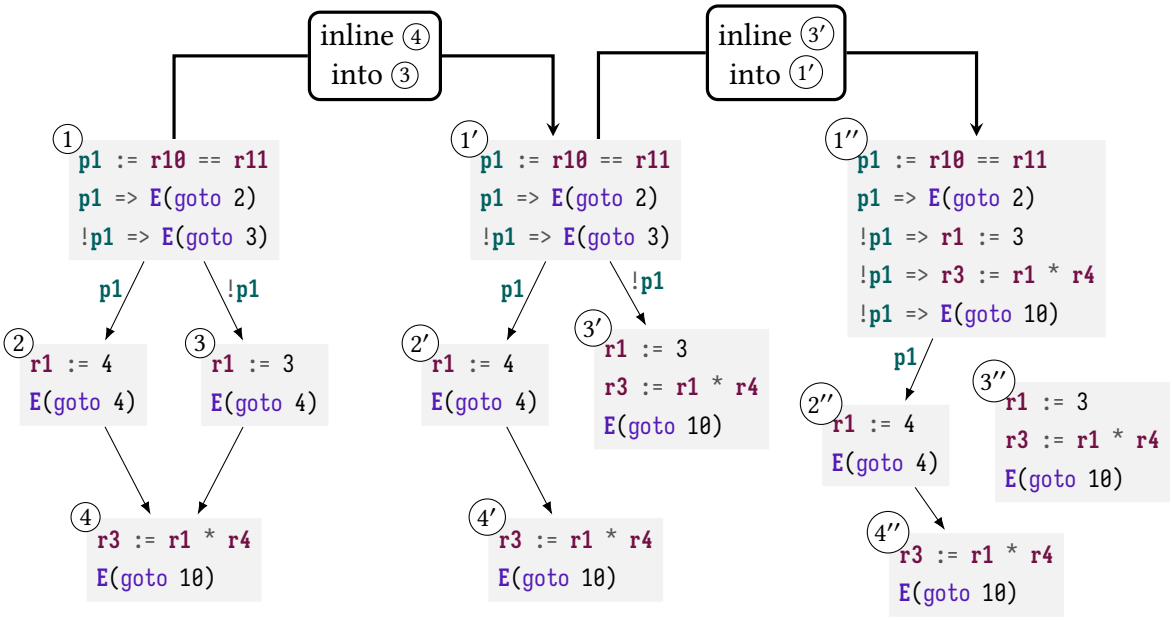
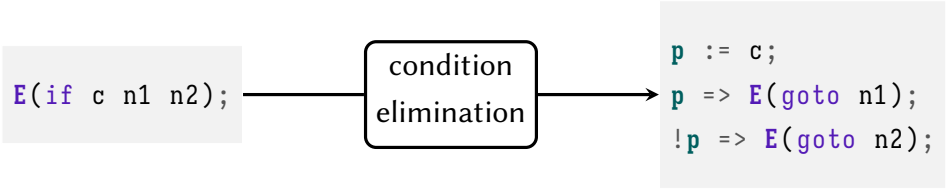


Figure 5.6: An example showing two iterations of the block-inlining pass.

general algorithm that can handle arbitrary branching code. We therefore implement the first formally verified implementation of a general if-conversion algorithm, with support for heuristics for branch prediction.

To simplify both the implementation of the if-converter and its correctness proof, it is split up into three distinct transformations, which are also shown in figure 5.5:

1. **Condition Elimination** First, every conditional instruction in the block is replaced by two predicated `goto` instructions. For example:



2. **Block Inlining** This is performed by replacing a predicated `goto` instruction by the list of instructions in the block that it points to, adding the predicate to each of those instructions. This transformation only converts one level of blocks at a time, but it can be repeatedly invoked to create larger blocks, as shown in figure 5.6. Currently, the block inlining pass is called a fixed number of times, however, it should be possible to execute it a variable number of times depending on the if-conversions that are selected by the decision procedure. The pointed-to block is left unchanged in case it is still pointed to by other blocks; as such, this transformation performs *tail duplication* [Chang et al. 1991].

3. **Dead Block Elimination** Finally, any blocks that are now unreachable from the function's entry-point (such as ③″ in figure 5.6) are removed, to reduce code size.

The decision about which `goto` instructions should be inlined is offloaded to an external procedure. This separation of concerns means that the correctness of the transformation can be proven once-and-for-all for a single, general if-conversion algorithm, which can then be extended with various heuristics to change the performance of the generated code. In our implementation, we use simple static heuristics to pick these paths, following Ball and Larus [1993], such as avoiding inlining loop back-edges, or blocks with an instruction count that exceeds a threshold (currently 50).

To prove the top-level end-to-end correctness theorem of Vericert, forward simulations proven for each transformation are composed together. The following theorem is a forward simulation and states the correctness of if-conversion.

Theorem 5.1 (Forward simulation of if-conversion)**.** *If program S is* `safe` *(free from undefined behaviour) and has behaviour $\mathcal{B}$, then* `ifconvert`(S) *should have the same behaviour. That is:* $\texttt{safe}(S) \land S \Downarrow \mathcal{B} \implies \texttt{ifconvert}(S) \Downarrow \mathcal{B}$.

Proof sketch. Each of the three transformations is verified using the simulation-diagram approach [Leroy 2009a, p. 379]. These simulations are then composed into an overall simulation for if-conversion. Condition elimination is straightforward because it is a purely local replacement. Dead block elimination is also straightforward, being similar

to a CFG-pruning transformation from CompCertSSA [Barthe et al. 2014]. Both of these transformations can be verified using a lock-step simulation diagram.

The block inlining pass is a bit more involved. To see why, consider how to prove a forward simulation for the first transformation in figure 5.6. The edges ① → ①′, ② → ②′, and ④ → ④′, are straightforward, but ③ is challenging, because ③′ does not straightforwardly simulate ③ (there is no edge from ③′ that can mimic the edge from ③ to ④). To resolve this, our simulation relation has to be more fine-grained, so that ③ can be mapped to the first 'part' of ③′ and ④ can be mapped to the second, meaning the simulation cannot proceed in lock-step. □

5.4 Implementing Hyperblock Scheduling

This section discusses the implementation of hyperblock scheduling in Vericert. The scheduler takes each hyperblock of an if-converted RtlBlock program in turn, and schedules it to form an RtlPar hyperblock. The scheduler is unverified, but it uses a verified translation validation algorithm to prove each output correct, which we will describe in section 5.5.

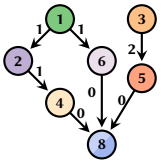
The scheduler is written in OCaml, and follows the SDC scheduling approach [Cong and Zhang 2006]. The SDC scheduler generates a function that should be minimised plus a set of constraints that must be respected while doing so. In our case, the function we minimise is the overall latency of the block (i.e. the end time of the last operation). The constraints come from three sources. First, the cumulative latency of all the operations in each chain must not exceed a predefined limit; this ensures that operation chaining does not reduce the maximum clock frequency of the resultant hardware. Second, whenever operation I_1 has a data dependency on I_2, I_2's end time must precede I_1's start time. Third, since our hardware has only a single BRAM controller, no two memory operations (loads or stores) may be scheduled for the same cycle.

These are all passed to an LP; we use `lp_solve` [Berkelaar 2010]. The solver outputs a mapping from instructions to states (clock cycles). We reconstruct from this mapping an RtlPar block. Data-dependent instructions mapped to the same state are placed into the same chain, at the innermost level of the RtlPar block; independent instructions mapped to the same state are placed in different chains in the same parallel group; and instructions mapped to different states are placed in different groups (the outermost list of the RtlPar block).

Figure 5.7 shows an example of our scheduler in action. In figure 5.7a, we see the

```
1 [        r2 := r1 + r4;
6  p1 =>   p3 := r4 == r2;
2  p1 =>   r1 := r2 + r4;
3  !p1 & !p2 =>
           r3 := r1 * r1;
4  p2 =>   r3 := r1 * r4;
7  p2 =>   E(goto 10);
5  !p2 =>  r3 := r3 * r3;
8          E(goto 10);    ]
```

(a) RTLBLOCK hyperblock to be scheduled.



(b) Data dependencies in the RTLBLOCK hyperblock (with transitive dependencies removed).

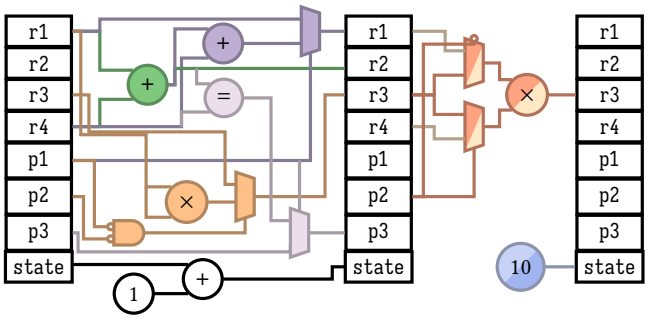
```
3 [[[ !p2 & !p1 =>
              r3 := r1 * r1 ];
1   [         r2 := r1 + r4;
2       p1 => r1 := r2 + r4;
6       p1 => p3 := r4 == r2 ]];
5  [[ !p2 =>  r3 := r3 * r3 ];
4   [  p2 =>  r3 := r1 * r4 ];
8   [         E(goto 10); ]]]
```

(c) Scheduled RTLPAR hyperblock.

```
state 1:
3 r3 = !p2 & !p1 ? r1 * r1 : r3;
1 r2 = r1 + r4;
2 r1 = p1 ? r2 + r4 : r1;
6 p3 = p1 ? r4 == r2;
  state = 32'd2;

state 2:
5 r3 = !p2 ? r3 * r3 : r3;
4 r3 = p2 ? r1 * r4 : r3;
8 state = 32'd10;
```

(d) HTL translation of the hyperblock, where instructions are placed into states and are then flattened from the original RTLPAR block and translated into a sequence of Verilog blocking assignments.



(e) Diagram showing how the operations in the RTLPAR hyperblock are scheduled. The circuit is drawn unrolled, so the duplicated registers are actually the same physical register. The colours represent which operations the resources belong to, showing that operations can be chained and shared.

Figure 5.7: Example of scheduling a hyperblock.

RtlBlock hyperblock to be scheduled. It contains six predicated operations: two additions, three multiplications, and a predicate assignment. The scheduler analyses the hyperblock and constructs a dependency graph (figure 5.7b). Each edge of the graph is annotated with the combinational delay of the operation at its head and an estimate for the delay incurred by the path produced by the edge. For example, every edge that leads to operation (8) `E(goto 10)` is annotated with a delay of 0 because the assignment to the 'state' variable (`state`) is performed immediately and the path delay is assumed to be negligible. These two delays are combined so that the graph can be reasoned about as a flow-graph to find the longest combinational path between two nodes.

The scheduler exploits predicates to eliminate dependencies. For example, (3) and (4) appear dependent due to a write-after-write (WAW) conflict on `r3`, but because their predicates are mutually exclusive, the conflict can be removed from the dependency graph. The scheduler would also remove operations whose predicates are false. These transformations are performed before the LP is generated.

The scheduler transforms the RtlBlock hyperblock into a RtlPar hyperblock (figure 5.7c). Even though the addition in (2) and the comparison in (6) both depend on (1), they can still be placed into the same state because the addition has a short enough combinational delay that two additions can be performed in a single clock cycle. The multiplication in (3) can also be placed into the same state as it does not have any data dependencies with any of the other instructions. The next state has two independent multiplications, (4) and (5), that can be scheduled for the same cycle. Finally, the hyperblock is terminated by a control-flow instruction that jumps to state `10`. This operation needs to be scheduled after all the other operations, but because it is performed by simply setting the next state of the state machine, this can be done in parallel with the last operation.

Figure 5.7d shows the corresponding Htl blocks. Htl maps each state to a sequence of Verilog blocking assignments. The translation from RtlPar to Htl first translates each element of the outer list into a separate Htl state. Then, the sequence of blocking assignments is produced by flattening the remaining inner nested lists. Verilog's blocking assignment has a sequential semantics, meaning it behaves like the RtlPar block.

Figure 5.7e shows how the resultant hardware executes. In particular, the two multiplications (4) and (5) could be allocated to the same resources since they never execute at the same time.

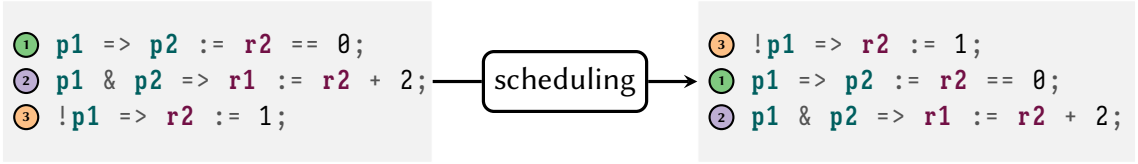


Figure 5.8: An example schedule. It is valid to move 3 before 1 and 2 because despite the appearance of data dependencies on r2, 3 is in fact independent because its guard is mutually exclusive with the other two.

5.5 Validation of Hyperblock Scheduling

Although the scheduling algorithm itself is complex with many heuristics, it is quite simple to check each specific schedule. To do so, we follow Tristan and Leroy [2008] and symbolically execute each block before and after scheduling, then compare the two obtained symbolic states for equivalence. The main difference from Tristan and Leroy's approach is that our paths are represented by predicates instead of by explicit branches. As a result, several non-obvious design decisions need to be made so that the validation process is tractable in the presence of hyperblocks. In what follows, we explain these decisions informally with the aid of the example shown in figure 5.8. The validator is presented more precisely in section 5.5.5.

5.5.1 First attempt: basic symbolic execution

The most natural way to extend Tristan and Leroy's approach is to treat predicates in the same way as registers. Symbolic execution then yields a symbolic state that assigns to each register and predicate an expression that is in terms of the initial values of the registers and predicates. Applying this approach to the example in figure 5.8 produces the two symbolic states shown in table 5.1. Note that we write $\texttt{r2}^0$ for the initial value of r2, and so on.

The pre- and post-scheduling expressions for r2 are syntactically equal, but reasoning about the equivalence of the two expressions for p2 is more involved: our validator needs to understand that $\texttt{if}\ \neg\texttt{p1}^0\ \texttt{then}\ 1\ \texttt{else}\ \texttt{r2}^0$ is equivalent to $\texttt{r2}^0$ in any context where $\texttt{p1}^0$ is true. Such reasoning could be performed by an SMT solver, encoding each arithmetic operator as an uninterpreted function, but formalising an SMT solver involves a lot of additional proof, and would be slow at run time.

Table 5.1: First attempt: basic symbolic execution

	Pre-scheduling symbolic state	Post-scheduling symbolic state
r1	$\texttt{if } (\texttt{p1}^0 \wedge (\texttt{if } \texttt{p1}^0 \texttt{ then } \texttt{r2}^0 == 0 \texttt{ else } \texttt{p2}^0))$ $\texttt{then } \texttt{r2}^0 + 2$ $\texttt{else } \texttt{r1}^0$	$\texttt{if } \left(\texttt{p1}^0 \wedge \left(\begin{array}{l}\texttt{if } \texttt{p1}^0 \\ \texttt{then } \left(\begin{array}{l}\texttt{if } \neg\texttt{p1}^0 \\ \texttt{then } 1 \\ \texttt{else } \texttt{r2}^0\end{array}\right) == 0 \\ \texttt{else } \texttt{p2}^0\end{array}\right)\right)$ $\texttt{then } (\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else } \texttt{r2}^0) + 2$ $\texttt{else } \texttt{r1}^0$
r2	$\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else } \texttt{r2}^0$	$\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else } \texttt{r2}^0$
p2	$\texttt{if } \texttt{p1}^0 \texttt{ then } \texttt{r2}^0 == 0 \texttt{ else } \texttt{p2}^0$	$\texttt{if } \texttt{p1}^0$ $\texttt{then } (\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else } \texttt{r2}^0) == 0$ $\texttt{else } \texttt{p2}^0$

5.5.2 Second attempt: using value summaries

Instead we would prefer to rely on a SAT solver, as it is easier to formalise and verify. A SAT solver can handle Boolean reasoning nicely, but cannot reason about arithmetic. So, to allow the use of a SAT solver, we rewrite each expression into a normal form where all the if-expressions are pulled to the top level, e.g. replacing $(\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 \texttt{ else } \texttt{r2}^0) == 0$ with $\texttt{if } \neg\texttt{p1}^0 \texttt{ then } 1 == 0 \texttt{ else } \texttt{r2}^0 == 0$. On our worked example (figure 5.8), this results in the symbolic states shown in table 5.2.

Note that we treat register expressions and predicate expressions slightly differently. For register expressions, we combine all the if-expressions into a single multi-way conditional, which we write using 'cases' notation. We call these expressions *value summaries* after Sen et al. [2015], who used the same data structure for a different purpose (namely, making symbolic execution more efficient).

For predicate expressions, we do not need value summaries, because all of the $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$ operations that appear in the predicate expressions become purely Boolean (rather than a mix of integers and Booleans), and hence can be expanded to $(e_1 \longrightarrow e_2) \wedge (\neg e_1 \longrightarrow e_3)$, as we have done in table 5.2. These predicate expressions can then be straightforwardly translated into propositional formulas that can be reasoned about using a SAT solver. For example, the generated query for checking the equivalence between the expressions assigned to p2 looks like the following:

Table 5.2: Second attempt: using value summaries.

	Pre-scheduling symbolic state	**Post-scheduling symbolic state**
r1	$\begin{cases} \mathtt{r2}^0 + 2, & \text{if } \mathtt{p1}^0 \wedge \left(\begin{array}{l}(\mathtt{p1}^0 \longrightarrow \mathtt{r2}^0 == 0)\\ \wedge\ (\neg\mathtt{p1}^0 \longrightarrow \mathtt{p2}^0)\end{array}\right) \\ \mathtt{r1}^0, & \text{if } \neg\left(\mathtt{p1}^0 \wedge \left(\begin{array}{l}(\mathtt{p1}^0 \longrightarrow \mathtt{r2}^0 == 0)\\ \wedge\ (\neg\mathtt{p1}^0 \longrightarrow \mathtt{p2}^0)\end{array}\right)\right) \end{cases}$	$\begin{cases} 1 + 2, & \text{if } \mathtt{p1}^0 \wedge \psi \wedge \neg\mathtt{p1}^0 \\ \mathtt{r2}^0 + 2, & \text{if } \mathtt{p1}^0 \wedge \psi \wedge \mathtt{p1}^0 \\ \mathtt{r1}^0, & \text{if } \neg(\mathtt{p1}^0 \wedge \psi) \end{cases}$
r2	$\begin{cases} 1, & \text{if } \neg\mathtt{p1}^0 \\ \mathtt{r2}^0, & \text{if } \mathtt{p1}^0 \end{cases}$	$\begin{cases} 1, & \text{if } \neg\mathtt{p1}^0 \\ \mathtt{r2}^0, & \text{if } \mathtt{p1}^0 \end{cases}$
p2	$(\mathtt{p1}^0 \longrightarrow \mathtt{r2}^0 == 0) \wedge (\neg\mathtt{p1}^0 \longrightarrow \mathtt{p2}^0)$	ψ

where ψ abbreviates $(\mathtt{p1}^0 \longrightarrow ((\neg\mathtt{p1}^0 \longrightarrow 1 == 0) \wedge (\neg\neg\mathtt{p1}^0 \longrightarrow \mathtt{r2}^0 == 0))) \wedge (\neg\mathtt{p1}^0 \longrightarrow \mathtt{p2}^0)$

$$\left((x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}) \wedge (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0})\right) \leftrightarrow S_\psi \tag{5.1}$$

where S_ψ abbreviates $\left((x_{\mathtt{p1}^0} \longrightarrow ((\neg x_{\mathtt{p1}^0} \longrightarrow x_{1==0}) \wedge (\neg\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}))) \wedge (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0})\right)$. In the encoding, each SAT variable x_e encodes the truth value of the expression e in the formula.

We can also generate SAT queries to check the equivalence of the register expressions. This involves issuing multiple queries to the SAT solver: if an expression appears in both the pre- and post-scheduling value summaries, then we generate a query to check that their guards are equivalent, and if an expression only appears in one of the value summaries, then its guard should be equivalent to *false*. For instance, to check r1 we generate the following three SAT queries:

$$\begin{aligned} \mathit{false} &\leftrightarrow x_{\mathtt{p1}^0} \wedge S_\psi \wedge \neg x_{\mathtt{p1}^0} \\ \left(x_{\mathtt{p1}^0} \wedge \left((x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}) \wedge (\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0})\right)\right) &\leftrightarrow \left(x_{\mathtt{p1}^0} \wedge S_\psi \wedge x_{\mathtt{p1}^0}\right) \\ \neg\left(x_{\mathtt{p1}^0} \wedge \left(\left(x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{r2}^0==0}\right) \wedge \left(\neg x_{\mathtt{p1}^0} \longrightarrow x_{\mathtt{p2}^0}\right)\right)\right) &\leftrightarrow \neg\left(x_{\mathtt{p1}^0} \wedge S_\psi\right) \end{aligned} \tag{5.2}$$

However, in constructing all these SAT queries, we have assumed that a Boolean value can be assigned to each of the atoms in a formula. This might not actually be the case – for instance, r1/r2 is not evaluable if r2 is zero, and r1 == r2 is not evaluable if either r1 or r2 is an invalid pointer. So, to compare the two expressions for p2, we actually need to use *three-valued logic*. That means all the SAT variables in (5.1) and (5.2) actually need to be *pairs* of binary variables, and the $\wedge$ and $\vee$ operations need to be three-valued analogues of

conjunction and disjunction. This is a problem because we have seen already that these formulas, particularly those for comparing register expressions, become quite large even for toy examples. Indeed, when we tried this three-valued approach on the test cases in our evaluation (chapter 7), most ran out of memory during validation.

Hence, in the next subsection we describe how we manage to avoid three-valued logic where possible.

5.5.3 Third attempt: using value summaries and final-state guards

Although it was not the case in our worked example, it turns out that when comparing the predicate expressions that arise in realistic examples, syntactic equality or near-equality usually suffices. This means that we only need to resort to solving three-valued SAT queries as an occasional fallback, so its performance impact is limited in practice.

Where syntactic methods usually do *not* suffice is for comparing the guards of register expressions. However, here we can actually avoid the need for three-valued logic altogether. We observe that the guards in the `r1` expressions are simply copied from the expressions for `p2` (which we abbreviated as ψ in table 5.2), so rather than writing out the full expressions in the guards, we can write $\texttt{p2}^{\text{f}}$ as a shorthand (the 'f' clarifies that it is referring to the *final* value of `p2`).

To make it possible to refer to final values, we need to ensure that once `p2` has been assigned or used, it is never overwritten. We can achieve this by enforcing SSA form for predicate assignments. That does not impose any restrictions on the Vericert user because predicates are only introduced by internal compiler transformations. SSA form is not needed for register assignments.

The resultant symbolic states are shown in table 5.3. It can immediately be seen that the expressions have become much shorter. Indeed, the three queries for validating `r1` become:

$$\begin{aligned} \mathit{false} &\leftrightarrow x_{\texttt{p1}^{\text{f}}} \wedge x_{\texttt{p2}^{\text{f}}} \wedge \neg x_{\texttt{p1}^{\text{f}}} \\ x_{\texttt{p1}^{\text{f}}} \wedge x_{\texttt{p2}^{\text{f}}} &\leftrightarrow x_{\texttt{p1}^{\text{f}}} \wedge x_{\texttt{p2}^{\text{f}}} \wedge x_{\texttt{p1}^{\text{f}}} \\ \neg(x_{\texttt{p1}^{\text{f}}} \wedge x_{\texttt{p2}^{\text{f}}}) &\leftrightarrow \neg(x_{\texttt{p1}^{\text{f}}} \wedge x_{\texttt{p2}^{\text{f}}}) \end{aligned} \tag{5.3}$$

What is less obvious is why we no longer need three-valued logic. This is because:

- We can assume that all predicate expressions in the pre-scheduling symbolic state are evaluable, because if any were not, the input program would fail at run time and we do not need to prove anything about our scheduler.

Table 5.3: Third attempt: using value summaries and final values in guards.

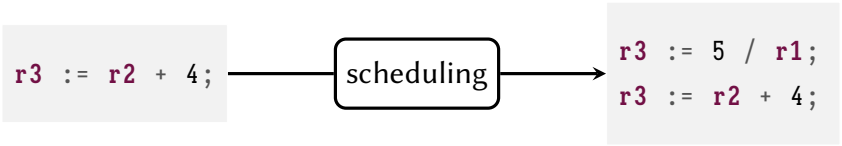
	Pre-scheduling symbolic state	**Post-scheduling symbolic state**
r1	$\begin{cases} \mathtt{r2}^0 + 2, & \text{if } \mathtt{p1}^\mathrm{f} \wedge \mathtt{p2}^\mathrm{f} \\ \mathtt{r1}^0, & \text{if } \neg(\mathtt{p1}^\mathrm{f} \wedge \mathtt{p2}^\mathrm{f}) \end{cases}$	$\begin{cases} 1 + 2, & \text{if } \mathtt{p1}^\mathrm{f} \wedge \mathtt{p2}^\mathrm{f} \wedge \neg\mathtt{p1}^\mathrm{f} \\ \mathtt{r2}^0 + 2, & \text{if } \mathtt{p1}^\mathrm{f} \wedge \mathtt{p2}^\mathrm{f} \wedge \mathtt{p1}^\mathrm{f} \\ \mathtt{r1}^0, & \text{if } \neg(\mathtt{p1}^\mathrm{f} \wedge \mathtt{p2}^\mathrm{f}) \end{cases}$
r2	$\begin{cases} 1, & \text{if } \neg\mathtt{p1}^\mathrm{f} \\ \mathtt{r2}^0, & \text{if } \mathtt{p1}^\mathrm{f} \end{cases}$	$\begin{cases} 1, & \text{if } \neg\mathtt{p1}^\mathrm{f} \\ \mathtt{r2}^0, & \text{if } \mathtt{p1}^\mathrm{f} \end{cases}$
p2	$(\mathtt{p1}^0 \longrightarrow \mathtt{r2}^0 == 0) \wedge (\neg\mathtt{p1}^0 \longrightarrow \mathtt{p2}^0)$	ψ

where ψ abbreviates $(\mathtt{p1}^0 \longrightarrow ((\neg\mathtt{p1}^0 \longrightarrow 1 == 0) \wedge (\neg\neg\mathtt{p1}^0 \longrightarrow \mathtt{r2}^0 == 0))) \wedge (\neg\mathtt{p1}^0 \longrightarrow \mathtt{p2}^0)$

- We have already proven, either using syntactic comparison or three-valued logic, that the predicate expressions in the post-scheduling symbolic state are equivalent to those in the pre-scheduling state, which means that they too must be evaluable.
- The guards in the register expressions only refer to these expressions – they cannot include unsafe expressions like division or pointer comparison – and so they must also be evaluable.
- It therefore suffices to use 2-valued logic to compare the guards.

5.5.4 Handling overwritten expressions

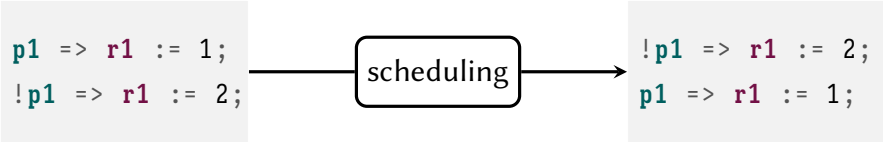
There is an additional subtlety that needs to be handled: the possibility that the scheduler introduces undefined behaviour. Consider the following example, due to Tristan and Leroy [2008].



Symbolic execution yields identical pre- and post-scheduling results, namely $\mathtt{r3} \mapsto \mathtt{r2}^0 + 4$. Despite this, the schedule is invalid because the post-scheduling block only executes correctly when r1 is nonzero. To detect and forbid such cases, we follow Tristan and Leroy and keep track of all the expressions that are evaluated into a register or memory location.

We shall call this the *encountered expression set.*[1] For example, the encountered expressions set of the pre-scheduling block above includes only $\texttt{r2}^0 + 4$, but the post-scheduling block's set also includes $5/\texttt{r1}^0$. Because the encountered set has grown, we deem the schedule invalid.

In order to use Tristan and Leroy's approach with hyperblocks, it needs extending to handle predicated instructions. The obvious way to do this is to generate the encountered expressions for each predicated instruction in the same way that we perform symbolic execution, which is essentially to treat `p => r := e` as if it is the non-predicated instruction `r := p ? e : r`. However, this approach leads to too many unnecessary constraints being imposed on the scheduler, leaving it unable to reorder some instructions that have only benign WAW dependencies. To see this, consider the following example.



When performing symbolic execution on the pre-scheduling block, we encounter the pair of expressions $\texttt{if p1}^\texttt{f}\ \texttt{then 1 else r1}^0$ and $\texttt{if}\ \neg\texttt{p1}^\texttt{f}\ \texttt{then 2 else if p1}^\texttt{f}\ \texttt{then 1 else r1}^0$, but on the post-scheduling block we encounter $\texttt{if}\ \neg\texttt{p1}^\texttt{f}\ \texttt{then 2 else r1}^0$ and $\texttt{if p1}^\texttt{f}\ \texttt{then 1 else if}\ \neg\texttt{p1}^\texttt{f}\ \texttt{then 2 else r1}^0$; these two pairs are not equivalent, so this (correct) schedule cannot be validated.

Instead, for the purposes of calculating encountered expressions, our approach is to treat `p => r := e` as the instruction `r := p ? e : •`, where • is a dummy expression representing the absence of an assignment. Now the set of encountered expressions is $\{\texttt{if p1}^\texttt{f}\ \texttt{then 1 else •}, \texttt{if}\ \neg\texttt{p1}^\texttt{f}\ \texttt{then 2 else •}\}$ for both pre- and post-scheduling, so validation can be completed.

5.5.5 Formalising the symbolic state and symbolic execution

The previous sections gave an informal overview of the structure of the symbolic state and the validation algorithm. This section will give formal definitions of these concepts.

Symbolic states Figure 5.9 defines the symbolic states that symbolic execution produces. Several components make use of value summaries (as explained in section 5.5.2), so we define the value summary $\mathcal{S}(t)$ as a set of terms of type t, each paired with a Boolean

[1] Tristan and Leroy simply called them 'constraints'.

arithmetic expressions:	$\mathbb{A} ::= \mathtt{r}^0$	(initial value of register)
	$\mid \mathbb{M}[\mathbb{A}]$	(load from memory)
	$\mid \mathbb{A}\ \mathtt{a}\ \mathbb{A}$	(binary arithmetic operation)
memory expressions:	$\mathbb{M} ::= \mathtt{Mem}^0$	(initial contents of memory)
	$\mid \mathbb{M}[\mathbb{A}\ \mathtt{->}\ \mathbb{A}]$	(updated memory)
predicate expressions:	$\mathbb{B} ::= \mathtt{p}^0 \mid \neg\mathtt{p}^0$	(initial value of predicate)
	$\mid \mathbb{A}\ \mathtt{c}\ \mathbb{A} \mid \neg(\mathbb{A}\ \mathtt{c}\ \mathbb{A})$	(binary conditional operation)
	$\mid \mathit{true} \mid \mathit{false}$	(true, false)
	$\mid \mathbb{B} \wedge \mathbb{B} \mid \mathbb{B} \vee \mathbb{B}$	(and, or)
value summaries:	$\mathcal{S}(t) = \mathcal{P}(\mathbb{G} \times t)$	(select an element of t according to which predicate holds)
symbolic states:	$\sigma_\mathrm{R} \in \Sigma_\mathrm{R} = \mathtt{r} \to \mathcal{S}(\mathbb{A})$	(expressions for registers)
	$\sigma_\mathrm{P} \in \Sigma_\mathrm{P} = \mathtt{p} \to \mathbb{B}$	(expressions for predicates)
	$\sigma_\mathrm{M} \in \Sigma_\mathrm{M} = \mathcal{S}(\mathbb{M})$	(contents of memory)
	$\sigma_\mathrm{E} \in \Sigma_\mathrm{E} = \mathcal{S}(\mathbb{I}^?_\mathrm{cf})$	(instruction to exit block)
	$\sigma_\mathrm{C} \in \Sigma_\mathrm{C} = \mathcal{P}(\mathcal{S}(\mathbb{A} + \mathbb{M}))$	(set of encountered expressions)

Figure 5.9: Syntax of symbolic states.

guard of type $\mathbb{G}$. Henceforth, we shall sometimes write value summaries explicitly as a set of (guard, value) pairs.

A symbolic state σ is made up of five components, the main three being: a register map σ_R that assigns an arithmetic expression (as a value summary) to each register, a predicate map σ_P that assigns a Boolean expression to each predicate, and an expression σ_M for the contents of memory (again as a value summary). We also need symbolic execution to track how control exits the block (to make sure that it does so in the same way after scheduling), so σ_E stores a value summary that evaluates to the instruction that is executed to exit the block (or to 'None' if the block has not finished yet). Finally, σ_C tracks the set of encountered expressions, as motivated in section 5.5.4.

Constructing symbolic states The expressions are constructed using a function which updates the symbolic expressions assigned for each resource. A core function used to update value summaries is the coalescing union operator $\uplus_q$ [Sen et al. 2015], which

conjoins $\neg q$ to each guard in its left operand and q to each guard in its right operand:

$$\begin{aligned} &\uplus_q \in \mathcal{S}(\mathbb{X}) \to \mathcal{S}(\mathbb{X}) \to \mathcal{S}(\mathbb{X}) \\ &X_1 \uplus_q X_2 \triangleq \{(\neg q \wedge G, v) \mid (G, v) \in X_1\} \cup \{(q \wedge G, v) \mid (G, v) \in X_2\} \end{aligned} \tag{5.4}$$

To turn a value summary back into a Boolean formula, we use the following operation, where g expands gates into predicate expressions:

$$\begin{aligned} &\wedge \in \mathcal{S}(\mathbb{B}) \to \mathbb{B} \\ &\wedge\{\, (G_1, B_1), \ldots, (G_n, B_n) \,\} \triangleq (g(G_1) \to B_1) \wedge \cdots \wedge (g(G_n) \to B_n) \end{aligned} \tag{5.5}$$

It is also useful to equip value summaries with the operators of an applicative functor [McBride and Paterson 2008], so that when we have value summaries of functions and of inputs, we can obtain a value summary of outputs:

$$\begin{aligned} &<\!*\!> \in \mathcal{S}(\mathbb{X} \to \mathbb{Y}) \to \mathcal{S}(\mathbb{X}) \to \mathcal{S}(\mathbb{Y}) \\ &F <\!*\!> X \triangleq \{\, (G \wedge G', f(x)) \mid (G, f) \in F, (G', x) \in X \,\} \end{aligned} \tag{5.6}$$

Following Sen et al. [2015], we simplify value summaries as they are built up, so as to keep their size from exploding: coalescing two elements (G, v) and (G', v') where $v = v'$ into a single element $(G \vee G', v)$, and removing elements (G, v) whenever $G \leftrightarrow \mathit{false}$.

Symbolic execution The symbolic execution of instruction I is performed by the α function. It takes the current symbolic state σ and produces an updated one. It also takes an 'enabled' predicate q, which is conjoined with the current instruction's guard; it ensures that after an exit instruction is taken, any subsequent instructions are nullified. So, whenever an exit instruction is encountered, the enabled predicate is conjoined with the negation of the exit instruction's guard.

In figure 5.10, we show three important cases of α: symbolically executing an arithmetic operation, an exit instruction, and a predicate assignment. To symbolically execute a whole hyperblock (denoted $\cdot^{\#}$), we run α on each instruction in turn, threading the symbolic state through, starting from the empty symbolic state (denoted $\varnothing$):

$$[i_1; i_2; \ldots; i_n]^{\#} \triangleq \alpha\ i_n\ (\ldots(\alpha\ i_2\ (\alpha\ i_1\ (\mathit{true}, \varnothing)))\ldots) \tag{5.7}$$

$$
\begin{aligned}
&\alpha\ (G \texttt{ => r := r1 + r2})\ (q, (\sigma_R, \sigma_P, \sigma_M, \sigma_E, \sigma_C)) && \text{(arithmetic operation)}\\
&\quad \triangleq \texttt{let } \phi = \{\ (\mathit{true}, (+))\ \} <*> \sigma_R[\texttt{r1}] <*> \sigma_R[\texttt{r2}] \texttt{ in}\\
&\qquad \texttt{let } \sigma'_R = \sigma_R\left[\texttt{r} \mapsto (\sigma_R[\texttt{r}] \uplus_{G \wedge q} \phi)\right] \texttt{ in}\\
&\qquad \texttt{let } \sigma'_C = \sigma_C \cup \{\ \{\ (\mathit{true}, \bullet)\ \} \uplus_{G \wedge q} \phi\ \} \texttt{ in}\\
&\qquad (q, (\sigma'_R, \sigma_P, \sigma_M, \sigma_E, \sigma'_C))\\
&\alpha\ (G \texttt{ => E}(I_{cf}))\ (q, (\sigma_R, \sigma_P, \sigma_M, \sigma_E, \sigma_C)) && \text{(exit instruction)}\\
&\quad \triangleq \texttt{let } \sigma'_E = \sigma_E \uplus_{G \wedge q} \{\ (\mathit{true}, \lfloor I_{cf} \rfloor)\ \} \texttt{ in}\\
&\qquad \left(q \wedge \neg G, \left(\sigma_R, \sigma_P, \sigma_M, \sigma'_E, \sigma_C\right)\right)\\
&\alpha\ (G \texttt{ => p := r1 == r2})\ (q, (\sigma_R, \sigma_P, \sigma_M, \sigma_E, \sigma_C)) && \text{(predicate assignment)}\\
&\quad \triangleq \texttt{let } \phi = \wedge(\{\ (G \wedge q, (==))\ \} <*> \sigma_R[\texttt{r1}] <*> \sigma_R[\texttt{r2}]) \texttt{ in}\\
&\qquad \texttt{let } \sigma'_P = \sigma_P[\texttt{p} \mapsto (\phi \wedge (\neg(G \wedge q) \rightarrow \sigma_P[\texttt{p}]))] \texttt{ in}\\
&\qquad (q, (\sigma_R, \sigma'_P, \sigma_M, \sigma_E, \sigma_C))
\end{aligned}
$$

Figure 5.10: Symbolic execution of selected instructions.

Comparing symbolic states After symbolically executing the RtlBlock and RtlPar blocks, we obtain two symbolic states, σ and σ'. We wish to show that σ' is a *symbolic refinement* of σ (written $\sigma \gtrsim \sigma'$), and we do so by component-wise comparison, as shown in equation (5.8) and explained below.

$$
\frac{\sigma_R \approx \sigma'_R \qquad \sigma_P = \sigma'_P \vee \sigma_P \approx_{3v} \sigma'_P \qquad \sigma_M \approx \sigma'_M \qquad \sigma_E \approx \sigma'_E \qquad \sigma_C \gtrsim \sigma'_C}{\sigma \gtrsim \sigma'} \tag{5.8}
$$

The core comparison operation that we rely upon is between two value summaries, written $\approx$. Whenever a value appears in both value summaries, we check that its guards are equivalent (via a SAT query), and whenever a value appears in just one value summary, we check that its guard is equivalent to *false* (again via SAT query). This approach suffices for the register maps ($\sigma_R \approx \sigma'_R$), the memory maps ($\sigma_M \approx \sigma'_M$), and the exit expressions ($\sigma_E \approx \sigma'_E$). For the predicate maps, we first attempt to show syntactic equality ($\sigma_P = \sigma'_P$). If this fails, we fall back to using a slow but reliable equivalence check with a three-valued solver ($\sigma_P \approx_{3v} \sigma'_P$). Finally, for the encountered expressions sets, we write $\sigma_C \gtrsim \sigma'_C$ to mean that every expression in σ'_C has an equivalent in σ_C.

5.5.6 Defining a Verified Scheduler

We can now define a verified scheduler using the standard translation validation approach:

$$\begin{aligned}\texttt{scheduleAndVerify}\ H \triangleq\ &\texttt{let}\ H_{\text{par}} = \texttt{schedule}\ H\ \texttt{in}\\ &\texttt{if}\ H^{\#} \gtrsim H^{\#}_{\text{par}}\ \texttt{then}\ \lfloor H_{\text{par}} \rfloor\ \texttt{else}\ \texttt{Error}\end{aligned} \tag{5.9}$$

5.6 Proving the Validator Correct

In order to prove our validator correct, we need to prove that whenever our validator deems $H^{\#}_{\text{par}}$ to be a symbolic refinement of $H^{\#}$, there is indeed a forward simulation from H to H_{par}; that is:

$$H^{\#} \gtrsim H^{\#}_{\text{par}} \implies H \rightsquigarrow H_{\text{par}}. \tag{5.10}$$

The natural way to prove $H \rightsquigarrow H_{\text{par}}$ is to follow Tristan and Leroy [2008] by constructing the chain $H \rightsquigarrow H^{\#} \rightsquigarrow H^{\#}_{\text{par}} \rightsquigarrow H_{\text{par}}$. In order to do this, we need to be able to talk about forward simulations that involve not just programs (H and H_{par}) but also symbolic states ($H^{\#}$ and $H^{\#}_{\text{par}}$). Thus we need a semantics not just for blocks (cf. figure 5.4) but also for symbolic states.

5.6.1 A semantics for symbolic states

We need a function that takes a symbolic state σ and applies it to an initial concrete state Γ. The output is the concrete state Γ', together with the control-flow instruction I_{cf} that is executed to exit the block. The function is written as $\Gamma \vdash \sigma \Downarrow (\Gamma', I_{\text{cf}})$, and is defined in figure 5.11. It works as follows:

- The entry point is the SemState rule. This rule has six antecedents. The first constructs a Boolean value for each predicate in the final state. The second constructs a value for each register in the final state, consulting Γ'_{p} to get the final values of predicates when evaluating value summaries (cf. section 5.5.3). The third constructs the final contents of memory. The fourth determines the control-flow instruction for exiting the block and the fifth expands Γ. The sixth does not calculate a component of the final state; instead, its purpose is to prevent the final state being calculated at all if any of the encountered expressions (cf. section 5.5.4) are unevaluable.

- The $\Downarrow_{\mathbb{A}}$ rules (RegBase, Load, and Op) map register expressions to concrete values (integer, float, pointer, or 'undefined'). In the Op rule, we write $\downarrow$ to indicate the existing CompCert evaluation semantics for the arithmetic operation op_{a}, which

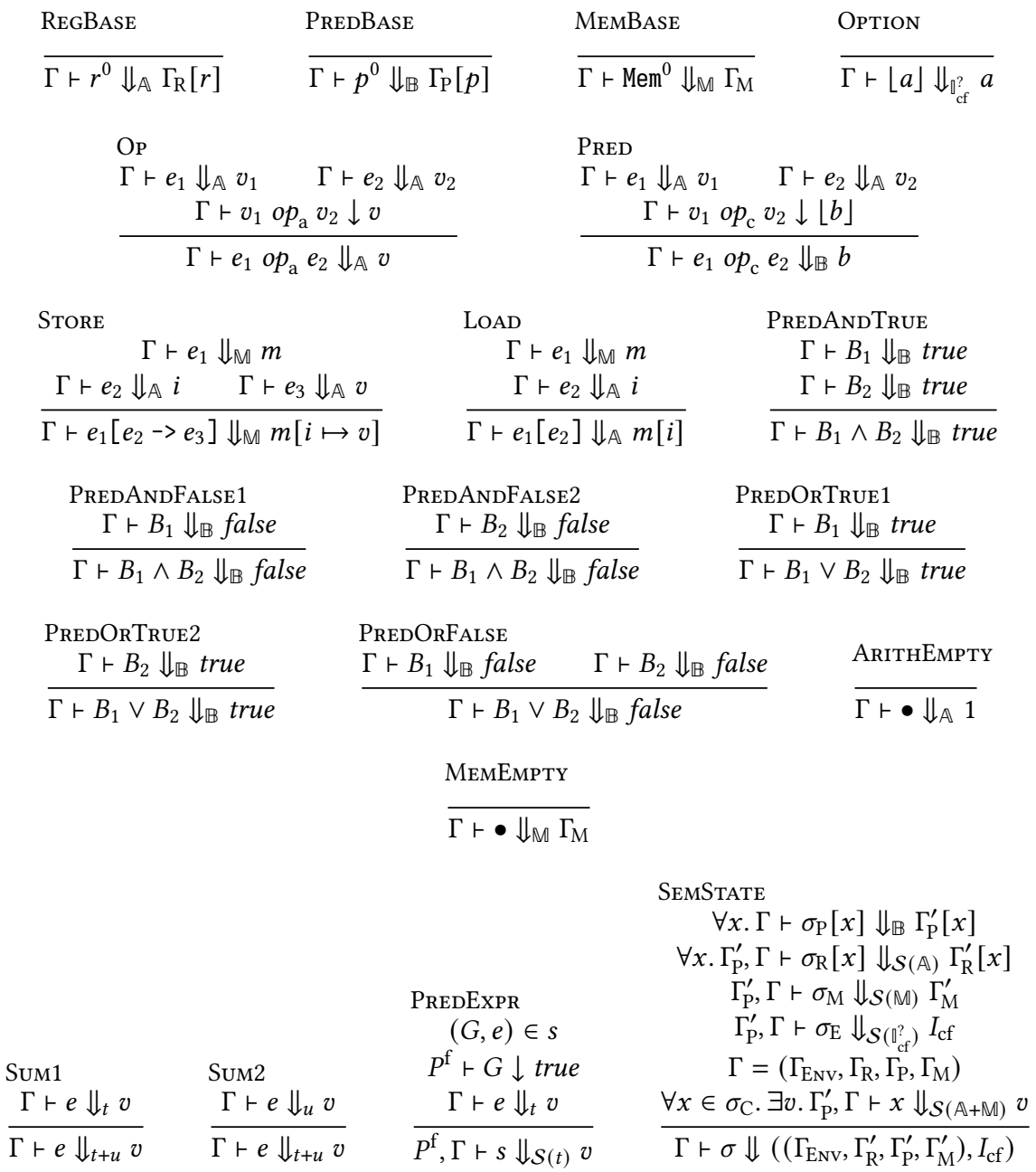


Figure 5.11: Semantics of symbolic states.

may need to consult Γ_{Env} to handle operands that are relative to the stack pointer or a global variable.

- The $\Downarrow_{\mathbb{M}}$ rules map memory expressions to concrete values.
- The ArithEmpty and MemEmpty rules map the dummy expression $\bullet$ to an arbitrary arithmetic value (1), or to an arbitrary memory (the initial memory). This is so we can check that all encountered expressions are evaluable; their actual values are immaterial.
- The $\Downarrow_{\mathbb{B}}$ rules map predicate expressions to Boolean values (*true* and *false*). In the Pred rule, evaluating $v_1\ op_{\text{c}}\ v_2$ returns an option type because v_1 or v_2 might be an invalid pointer.
- The rules for $\wedge$ and $\vee$ are designed to produce a lazy semantics. This is necessitated by the fact that they originate from if-statements in the source program, which must be evaluated lazily. In particular, if B_1 evaluates to *false* and B_2 is unevaluable, then we need $B_1 \wedge B_2$ to evaluate to *false*, not to be unevaluable.
- The PredExpr rule is for evaluating a value summary s of type $\mathcal{S}(t)$. It finds an entry (G, e) for which the guard G evaluates to true in the final state (P^{f}), and then evaluates e at type t. Value summaries are constructed so that the guards are exhaustive and mutually exclusive, so there will always be exactly one such entry.

5.6.2 Establishing the chain of simulations

Now that we have a semantics for symbolic states, we can define the required forward simulation relation, $a \rightsquigarrow b$, where a and b are both blocks, or both symbolic states, or one of each.

Definition 5.1 (Forward lock-step simulation diagram)**.** For every execution of a, there exists an execution of b that, when starting from a matching initial state, results in a matching final state.

$$a \rightsquigarrow b \quad \triangleq \quad \forall \Gamma_1, \Gamma_1', \Gamma_2, I_{\text{cf}}.\, \Gamma_1 \vdash a \Downarrow (\Gamma_1', I_{\text{cf}}) \wedge \Gamma_1 \sim \Gamma_2 \implies \exists \Gamma_2'.\, \Gamma_2 \vdash b \Downarrow (\Gamma_2', I_{\text{cf}}) \wedge \Gamma_1' \sim \Gamma_2' \tag{5.11}$$

First, we show that the forward simulation is transitive.

Lemma 5.2 (Transitivity of the forward simulation)**.**

$$\forall\, A\, B\, C.\;\; A \rightsquigarrow B \wedge B \rightsquigarrow C \implies A \rightsquigarrow C \tag{5.12}$$

The correctness of the scheduler can then be stated in terms of the results of the equivalence check. We want to prove that given an input block H, and given the scheduled block H_{par}, that $H \rightsquigarrow H_{\text{par}}$ when their respective symbolic states are equivalent, i.e. $H^{\#} \approx H^{\#}_{\text{par}}$. We can split up this work into three main steps, first we show that symbolic execution is sound and complete with respect to the semantics given to the symbolic block and the input programs. Next, we need to show that the equivalence check of two symbolic states is correct, and therefore implies equivalent behaviour of the symbolic states.

First, we need to show that symbolic execution is sound and complete, meaning that a behaviour is a valid behaviour of the input program if and only if it is also a valid behaviour of the symbolic state. Both directions are needed, because we need to show that any behaviour of the pre-scheduling hyperblock H is a behaviour of its symbolic state $H^{\#}$, and also that once we have a behaviour of the symbolic state $H^{\#}_{\text{par}}$ associated with the scheduled hyperblock H_{par}, that this will also be the behaviour of H_{par} itself. The former is a soundness property, whereas the latter is a completeness property. It remains to construct the chain $H \rightsquigarrow H^{\#} \rightsquigarrow H^{\#}_{\text{par}} \rightsquigarrow H_{\text{par}}$. The three steps of that chain are captured by the following three lemmas.

Lemma 5.3 (Soundness of symbolic execution)**.** *For every execution of a block H, there exists an equivalent execution of its symbolic state $H^{\#}$. That is: for all H, we have $H \rightsquigarrow H^{\#}$.*

Proof Sketch. By induction on the execution semantics of H. □

Lemma 5.4 (Symbolic refinement implies behavioural refinement)**.** *For all σ, σ', we have $\sigma \gtrsim \sigma' \implies \sigma \rightsquigarrow \sigma'$.*

Proof Sketch. For expressions this is just syntactic equality, while for value summaries this comes down to proving the correctness of the SAT solver. □

Lemma 5.5 (Completeness of symbolic execution)**.** *For every execution of the symbolic state $H^{\#}_{par}$, there exists an equivalent execution of block H_{par}. That is: for all H_{par}, we have $H^{\#}_{par} \rightsquigarrow H_{par}$.*

Proof Sketch. Completeness requires a bit more work, because we are given the final symbolic state and need to show that the whole block that generated it produces the same

result. First we show that any instruction in the original block necessarily produces a value, which follows from the semantics of encountered expressions. From this, we can show that the execution of H_{par} in the current context must produce a state. Next, we use lemma 5.3 to show that $H^{\#}_{\text{par}}$ must produce a state that is equivalent to H_{par}. Finally, because our semantics of symbolic states is deterministic, we can show that this state must be unique, therefore they must be equivalent. □

Finally, we can show the correctness of the verified scheduling implementation.

Theorem 5.6 (Scheduler Correctness)**.** *Whenever $H^{\#}_{par}$ is a symbolic refinement of $H^{\#}$, there is a forward simulation from H to H_{par}. That is: for all H, H_{par}, we have $H^{\#} \gtrsim H^{\#}_{par} \implies H \rightsquigarrow H_{par}$.*

Proof. This follows from lemmas 5.3 to 5.5 and the transitivity of $\rightsquigarrow$. □

5.6.3 Managing complexity in the proof

The proof of theorem 5.6, together with the necessary additions to the Vericert back end, is as large as original Vericert's whole correctness proof. If one then adds the proofs of the if-conversion pass and the changes that had to be made to existing passes, Vericert with hyperblock scheduling has 16681 sloc of Coq definitions and 17426 sloc of Coq proofs, making it 3× larger than the original Vericert implementation. It was therefore particularly important to take steps to manage the proof's complexity, primarily by breaking it up into reusable lemmas.

A substantial portion of the proof involves reasoning about the α function for symbolically executing instructions. As shown in figure 5.10, the definition of this function is naturally broken down into smaller state-updates using the applicative interface for value summaries, <*> (from equation (5.6)). Accordingly, it is desirable to formulate lemmas that follow the same structure. For instance, if we want to show some property holds for F <*> X, we would like a lemma that breaks this down into some related properties holding for F and X separately.

However, it is not possible to reason about the behaviour of value summaries like $\{ (true, (+)) \}$ in isolation, because our current semantics of symbolic states (figure 5.11) gives no meaning to functions such as (+). Our solution is to extend the semantics with a

rule that can handle any value summary.

$$\frac{\textsc{PredExprIdentity} \quad (G, e) \in s \qquad P^{\mathrm{f}} \vdash G \downarrow \mathit{true}}{P^{\mathrm{f}} \vdash s \Downarrow_I e} \tag{5.13}$$

This rule achieves this by making no attempt to *evaluate* the expression e that it selects from the value summary, and instead simply returns it. (In contrast, PredExpr demands that e can be evaluated to a value v.) Hence we call this the *identity semantics* for the value summary. By using identity semantics, it becomes possible to formulate lemmas that capture the behaviour of <*>, such as:

$$(P^{\mathrm{f}} \vdash F \Downarrow_I e) \wedge (P^{\mathrm{f}} \vdash X \Downarrow_I e') \implies P^{\mathrm{f}} \vdash (F \mathbin{<\!*\!>} X) \Downarrow_I e(e') \tag{5.14}$$

We found that only once we were able to formulate lemmas like these did the proof become feasible. Without them, it involved a number of special cases that was simply unworkable.

5.7 Comparison Against Other Validated Schedulers

The most closely related works to ours are those by Tristan and Leroy [2008] and by Six et al. [2022], so this section begins by recapping our main points of similarity and difference.

Tristan and Leroy [2008] were the first to propose adding scheduling to a verified compiler, and we adopt their method for validating schedules – running symbolic execution before and after scheduling and comparing the resultant symbolic states. Their scheduler only *reorders* instructions, so syntactic equality suffices for comparing symbolic states, whereas our scheduler can also *modify* instructions (by manipulating predicates). This means that our state comparisons are more involved, and we turn to a SAT solver to help resolve them (section 5.5.2). Tristan and Leroy also devised the use of constraints to prevent the scheduler introducing undefined behaviour; we adopt this technique too, taking care to extend it to handle predicates in such a way that valid schedules can still be validated (section 5.5.4). We remark that a direct empirical comparison with Tristan and Leroy's work is not feasible because their method was implemented in an old version of CompCert and was not incorporated into its main branch.

Six et al. [2022] formalise superblock scheduling, which is a restricted form of trace

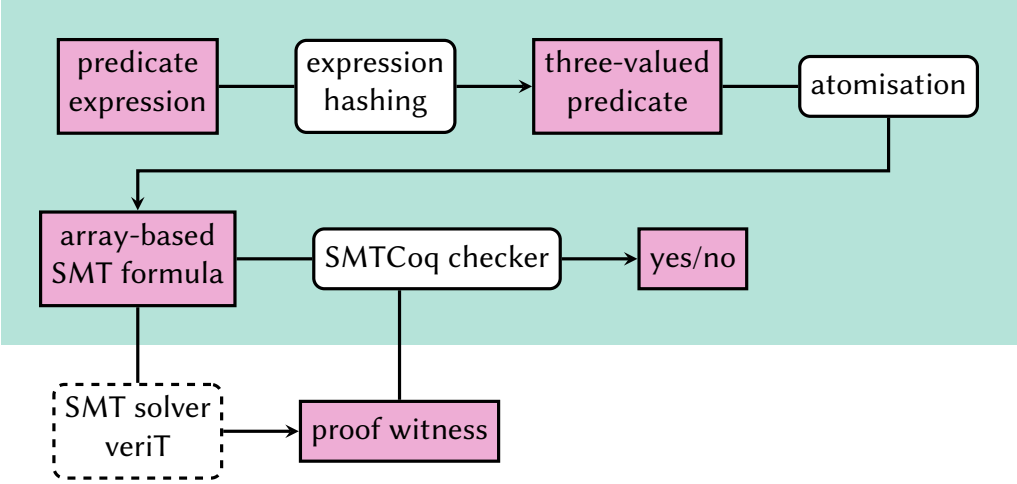


Figure 5.12: Validation of predicate expressions using three-valued logic. An external SMT solver is used to generate a proof witness, which is then given to a formally verified proof witness checker by SMTCoq. The result is then 'yes' if the formula always holds, and 'no' otherwise.

scheduling that is well-suited for VLIW processors. Hyperblock scheduling is more general than superblock scheduling and is well-suited to our application domain, HLS. Six et al.'s scheduler reorders instructions, and also splits instructions up where it is advantageous to do so, so comparing symbolic states is more involved than it was for Tristan and Leroy. A SAT solver was still not required because there are no predicates to reason about. A direct empirical comparison between our scheduler and Six et al.'s is difficult because Six et al.'s is based on an incompatible fork of CompCert called CompCert KVX. The final compiler targets are also wildly different, meaning the schedulers would have to be modified extensively to be comparable and share a back end.

5.8 Validated three-valued Logic Using an SMT Solver

This section describes the implementation and proof of the validated three-valued logic solver using an external proof-generating SMT solver. The goal of this validator is to be able to prove the equivalence of predicate expressions, which as noted in section 5.5 requires three-valued logic. This validator is used when syntactic equality between predicate expressions fails. Figure 5.12 shows the steps performed by the equivalence checker, assuming that the predicate expression that is given as input is encoding the equivalence. We use an SMT solver to generate a proof witness for the formula, which can be checked

$A \rightarrow_Ł B$		B		
		F	U	T
A	F	T	T	T
	U	U	T	T
	T	F	U	T

$A \wedge B$		B		
		F	U	T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

$A \vee B$		B		
		F	U	T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

Figure 5.13: Truth tables corresponding to the three-valued logic operators, where T, U and F are abbreviations for True, Undef and False respectively.

by SMTCoq [Armand et al. 2011; Ekici et al. 2017], a formally verified SMT proof checker for Coq.

We might therefore want to prove the equivalence for the predicate expression from the example in section 5.5.

$$\left(\texttt{p1}^0 \rightarrow \texttt{r2}^0 == 0\right) \wedge \left(\neg\texttt{p1}^0 \rightarrow \texttt{p2}^0\right) \leftrightarrow \psi, \tag{5.15}$$

where ψ abbreviates $\left(\texttt{p1}^0 \rightarrow \left(\left(\neg\texttt{p1}^0 \rightarrow 1 == 0\right) \wedge \left(\neg\neg\texttt{p1}^0 \rightarrow \texttt{r2}^0 == 0\right)\right)\right) \wedge \left(\neg\texttt{p1}^0 \rightarrow \texttt{p2}^0\right)$.

First, this predicate expression is hashed so that it is only composed of variables, and produces a proper three-valued predicate. In this case, all the atoms of the formula could be hashed and only referred to as variables, for example $c_1 \mapsto \texttt{p1}^0$, $c_2 \mapsto \texttt{r2}^0 == 0$ and $c_3 \mapsto \texttt{p2}^0$. The same is done to ψ producing the hashed formula $h(\psi)$. We are then left with the following formula:

$$(c_1 \rightarrow_Ł c_2) \wedge (\neg c_1 \rightarrow_Ł c_3) \leftrightarrow_Ł h(\psi). \tag{5.16}$$

This is a pure three-valued logic formula using Łukasiewicz three-valued logic semantics [Borowski 1970], where the truth tables for 'implication', 'and' and 'or' are given in figure 5.13. This interpretation of three-valued logic is chosen because it allows for tautologies in the logic. In the standard interpretation of three-valued logic where implication is defined in terms of $\wedge$ and $\vee$, every expressable formula will evaluate to Undef if all the variables are set to Undef because Undef $\rightarrow$ Undef $\equiv$ Undef. Łukasiewicz three-valued logic, on the other hand, has the following behaviour for implication Undef $\rightarrow_Ł$ Undef $\equiv$ True, also shown in the truth table in figure 5.13, meaning one can express formulas that are tautologies to show that a certain property will always hold. As a concrete example,

$$\frac{}{rs \models_{\mathrm{p}} \mathsf{True} \Downarrow 1} \qquad \frac{}{rs \models_{\mathrm{p}} \mathsf{False} \Downarrow -1} \qquad \frac{}{rs \models_{\mathrm{p}} \mathsf{Undef} \Downarrow 0} \qquad \frac{}{rs \models_{\mathrm{p}} c \Downarrow rs[c]}$$

$$\frac{}{rs \models_{\mathrm{p}} \overline{c} \Downarrow 1 - rs[c]} \qquad \frac{rs \models_{\mathrm{p}} p_1 \Downarrow b_1 \qquad rs \models_{\mathrm{p}} p_2 \Downarrow b_2}{rs \models_{\mathrm{p}} p_1 \vee p_2 \Downarrow b_1 \max b_2} \qquad \frac{rs \models_{\mathrm{p}} p_1 \Downarrow b_1 \qquad rs \models_{\mathrm{p}} p_2 \Downarrow b_2}{rs \models_{\mathrm{p}} p_1 \wedge p_2 \Downarrow b_1 \min b_2}$$

$$\frac{rs \models_{\mathrm{p}} p_1 \Downarrow b_1 \qquad rs \models_{\mathrm{p}} p_2 \Downarrow b_2}{rs \models_{\mathrm{p}} p_1 \to_{\mathrm{Ł}} p_2 \Downarrow 1 \min (1 - b_1 + b_2)}$$

Figure 5.14: Evaluation of three-valued logic predicates, where rs is a valuation from predicate variables c to $\{ 1, -1, 0 \}$.

$(a \to a) \leftrightarrow \mathsf{True}$ would not be provable in the more standard formulation of three-valued logic, because a can always be set to Undef, whereas $(a \to_{\mathrm{Ł}} a) \equiv \mathsf{True}$ is provable, because even if a is set to Undef, the formula will still evaluate to True.

The final steps of the validation are therefore to translate the three-valued logic equation into a form that is accepted by SMTCoq. For this, we give an interpretation of the three-valued logic in linear arithmetic, which is shown in figure 5.14. Let us try to prove that the following formula always holds: $(c_1 \to_{\mathrm{Ł}} \mathsf{True}) \wedge (\neg c_1 \to_{\mathrm{Ł}} \mathsf{True})$. Using the linear arithmetic interpretation, this would give the following SMT formula, where all the variables are constrained to be between -1 and 1, i.e. $\forall n. -1 \leq c_n \leq 1$. The constraint has been left out from the formulation below for simplicity, but would have to be added to the actual formulation. Finally, we check that this formula does not equal 1, meaning if the SMT solver returns `unsat`, then the formula should always hold.

$$(1 \min (1 - c_1 + 1)) \min (1 \min (1 - (1 - c_1) + 1)) \neq 1. \tag{5.17}$$

Internally, SMTCoq uses an efficient and compact array structure to represent the formula, which is split into three main parts, the array for the atoms of the formula, the array for formula components, and finally the array containing the formula itself, called the roots. When translated to the array representation, the above formula would be represented as follows in SMTCoq, where we have the list of atoms a, which are made up of a boolean true ($\top$) and false ($\bot$), in addition to the two atoms that are needed to encode the formula above, namely 1 and the variable c_1. Then, the formula is constructed as a list of formulas that either reference elements earlier in the list or atoms, and encode the formula above.

Finally, the root contains a reference to the formula that should be checked by the SMT solver.

$$\begin{aligned}
&\text{atoms:} && a = [\top; \bot; 1; c_1] \\
&\text{formulas:} && f = [a[2]; a[3]; f[0] - f[1]; f[2] + f[0]; f[0] \min f[3]; \\
& && \quad f[0] - f[2]; f[5] + f[0]; f[0] \min f[6]; f[4] \wedge f[7]; f[8] \neq f[0]] \\
&\text{roots:} && r = [f[9]].
\end{aligned} \tag{5.18}$$

The original SMT formula can then be passed to an SMT solver to check for unsatisfiability. This is done using veriT [Bouton et al. 2009], which is well supported by SMTCoq. The witness, together with the array representation of the formula are then read by the SMTCoq proof checker, which emits 'yes' or 'no' depending on if checking the witness succeeded, in which case the formula that was checked was unsatisfiable or not. This result can be used to check the equivalence between two predicate expressions at run time and assume that if the check succeeds, that the predicate expressions can be assumed to be equivalent.

5.9 Summary

We have presented the first verified implementations of general if-conversion and hyperblock scheduling, and incorporated them into the Vericert verified HLS tool. The practical value of this work is that it makes verified HLS practical by validating an industry standard scheduling algorithm. On the more conceptual side, our work may prove useful to those implementing other optimisation passes in a verified compiler using solver-powered validation. For example, this back end was also used to verify the construction of gated-SSA, where three-valued logic validation was also needed to show that path predicates were correct [Herklotz et al. 2023].

There are opportunities for further performance improvements by tweaking the if-conversion heuristics and the implementation of the scheduler. Both of these should be relatively straightforward because neither affects the correctness proof. Longer term, we plan to implement further optimisations in Vericert, such as modulo scheduling [Zhang and Liu 2013], which would enable loops to be pipelined.

6 Hardware Generation

This chapter describes the final hardware generation step to go from software-like semantics to hardware Verilog semantics.

Until now the representation of the program has still been in the form of a software program, with virtual, infinite registers, a program counter, as well as a rich but abstract memory model. This representation needs to be transformed into a more suitable representation on which hardware specific transformations can be applied, and which is closer to the structure of the final Verilog design. The main transformation that takes place is converting a control-flow graph into a state machine with data path representation, which is also the structure of the final design. There are various steps involved in this refinement because of the large gap between control-flow graph semantics and state machine semantics. In addition to that, additional components, such as an implementation of a memory that can be efficiently implemented in hardware, need to be added to the hardware to produce a useful design. Finally, until now programs have only been executing sequentially, whereas to produce the final hardware one will have to transform the sequential execution of operations within each state into parallel assignments.

This chapter describes the hardware generation process, starting from RtlPar and producing a final Verilog design. Figure 6.1 shows the intermediate transformations and in which section the transformation is described. Section 6.1 describes the first step in the transformation which separates each sequential block within a state in RtlPar into separate states that can be addressed using the program counter. This matches the addressing that the state register would have to do in the hardware design. Next, section 6.2 describes the generation of Htl, an intermediate language representing the execution of a state machine. This performs the main transformation from a software representation of the program into hardware, making the execution of the program more explicit in the design itself rather than as part of the semantics. Section 6.3 then describes the first hardware-specific optimisation on the state-machine representation of the hardware by adding a specification

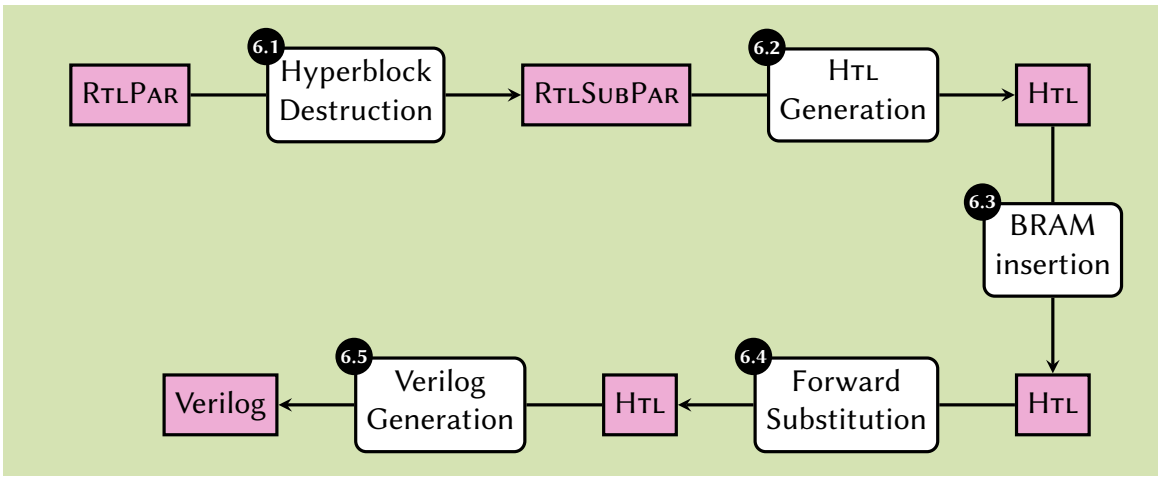


Figure 6.1: Hardware generation transformation passes introduced to convert RTLPAR to Verilog.

of a BRAM to the HTL semantics and replacing any explicit reads and writes to the array representing memory by properly formed reads and writes to the BRAM. Until now, updates to registers have been specified sequentially, so a forward substitution transformation is described in section 6.4 to parallelise the updates to registers. Finally, section 6.5 describes the generation of the final Verilog design, which implements the state-machine that was specified by HTL, in particular implementing the BRAM that was specified.

6.1 Hyperblock Destruction

RTLPAR is a control-flow graph with nodes mapping to hyperblocks. This is useful for the scheduling proof, as each of these hyperblocks can be compared individually. However, in the hardware itself, the individual sequential blocks have to be separated into different states, because within each state the assignments will be performed in parallel. This first hyperblock destruction transformation separates operations that should execute in different clock cycles into their own locations in the control-flow graph.

Figure 6.2 shows an example hyperblock destruction transformation, where a new block is added at location ③ with the contents of the second sequential block, and a `goto` instruction is added to the original block leading to this next block. Fresh locations for new blocks are chosen by keeping track of the greatest location in the current function.

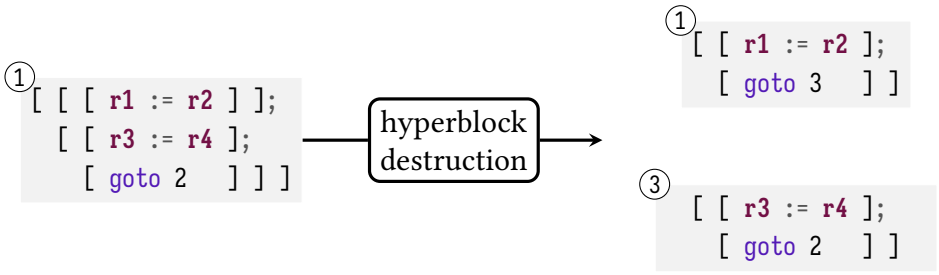


Figure 6.2: Hyperblock destruction transformation splitting up the hyperblock into multiple locations.

6.1.1 Proof of hyperblock destruction

The proof of hyperblock destruction is relatively simple using a 'plus' forward simulation. Then, for each input state, there are one or more output states returned by the hyperblock destruction algorithm that should be equivalent to the execution of the input state when executed sequentially.

6.2 HTL Generation

The main transformation of an HLS tool is the generation of a hardware description from the list of instructions. Eventually the hardware design will be described using Verilog, however, to make the transformation more incremental, we first turn the program represented by a control-flow graph into a program represented by an FSMD. This section will describe the FSMD language, called HTL, by showing the syntax and semantics of the language. Then, the transformation from RTLSUBPAR to HTL is shown, together with an overview of its correctness proof.

6.2.1 HTL structure and semantics

At a high level, HTL is structured like many other CompCert languages, mapping from locations to Verilog statements. However, contrary to many other intermediate languages in CompCert, HTL does not contain any instructions, and its semantics use a smaller state to perform the execution. For example, the state does not have to contain the program counter because there is an explicit state register that keeps track of it.

HTL comprises a lot of metadata pointing to important registers, as well as containing a map from program locations to Verilog statements. The syntax of HTL is shown in figure 6.3.

$$
\begin{aligned}
\text{registers:} \quad & r, \texttt{r1}, \texttt{r2}, \ldots \in \mathbb{r} \\
\text{CFG node labels:} \quad & L \in \mathbb{L} ::= \mathbb{N} \\
\text{Verilog statements:} \quad & V_{stmnt} \in \mathbb{V}_{stmnt} \\
\text{Code:} \quad & c \in \mathbb{L} \rightarrow \mathbb{V}_{stmnt} \\
\text{HTL:} \quad & HTL ::= \{\ params : \mathbb{r}\ \texttt{list}; \\
& \qquad datapath : \mathbb{L} \rightarrow \mathbb{V}_{stmnt}; \\
& \qquad entrypoint : \mathbb{N}; \\
& \qquad st : \mathbb{r}; \\
& \qquad stk : \mathbb{r}; \\
& \qquad stack\_size : \mathbb{N}; \\
& \qquad fin, ret, rst, clk : \mathbb{r}; \\
& \qquad ram : \text{BRAM}^{?}; \\
& \qquad order\_wf : st < fin < ret < stk \\
& \qquad\qquad \wedge\ stk < rst < clk; \\
& \qquad ram\_wf : \forall r.\ ram = \lfloor r \rfloor \implies clk < r.raddr; \\
& \qquad params\_wf : \forall r \in params.\ r < st\ \}
\end{aligned}
$$

Figure 6.3: Syntax of HTL.

First, HTL contains a list of parameters, which are additional input registers to the current module. Next, the *datapath* contains the code for the state machine, as well as the data path associated with each state. This is done by mapping program locations, or states, to Verilog statements $\mathbb{V}_{stmnt}$, which updates registers as part of the data path, but also updates to the state. The computation of the next state often relies on the state of registers, which is why it needs to be performed as part of the data path.

Next, the HTL module contains an entry point, which is the initial starting state of the *st* register. After the reset input wire is asserted, the *st* register will be reset to that value. The *st* register is read at every clock tick and determines the next statement that should be executed from the *datapath*. As mentioned before, the *st* register is a physical representation of the virtual program counter from RTLSUBPAR and other intermediate instruction languages. We then also have *stack* register and an associated *stack_size*, which is a Verilog array storing the contents of the stack frame. Initially, this array will be accessed directly by operations in the data path, however, section 6.3 describes how these direct accesses to the array are instead turned into accesses to a BRAM.

We then have a list of input and output control signals, which are used to return a result by setting the *fin* flag and assigning the *ret* register to the result. Next, the *rst* signal provides a way to reset the state of the internal state machine. Finally, there is a *clk* input to provide the clock to the design. This input is not yet used by the HTL design, as execution

is still performed using state transitions in the semantics, however, a register is already allocated for the clock which will be needed by the final Verilog design. The module then also contains a *ram* which will be further described in section 6.3, because in the first translation pass it is initialised to `None`, and is not used.

Finally, there are three well-formedness criteria which are used to enforce an ordering between the registers, mainly to be able to show that registers are independent from each other.

6.2.2 HTL generation algorithm

The generation of HTL is relatively straight-forward, as most instructions have a direct translation to a Verilog implementation. The Verilog implementation can therefore follow the semantics of each operation and implement their arithmetic behaviour directly. In addition to that, because of the hyperblock destruction translation, each block in the control-flow graph corresponds to a state in the final hardware. First, the translation of individual instructions is described, followed by the translation of memory. Next the translation of control-flow statements is described, followed by the translation of the top-level function.

Translating individual arithmetic instructions

The arithmetic operation is then assigned to the destination register using *blocking* assignment, so as to preserve the sequential nature of the execution of the code and simplify the correctness proof. One subtle aspect of the proof is the translation of registers and predicates into Verilog, because in RTLSUBPAR these were contained in separate maps, whereas in Verilog and HTL they need to be combined into one register association map. This is done by referring to register r in RTLSUBPAR as register $r' = 2r$ in Verilog. Predicate p, on the other hand, is referred to as predicate $p' = 2p + 1$ in Verilog, thereby combining both name spaces into one. A short example of a translation from RTLSUBPAR to HTL is shown in figure 6.4.

Implementing the `Oshrximm` instruction Many of the CompCert instructions map well to hardware, but `Oshrximm` (efficient signed division by a power of two using a logical shift) is expensive if implemented naïvely. The problem is that in CompCert it is specified as a signed division:

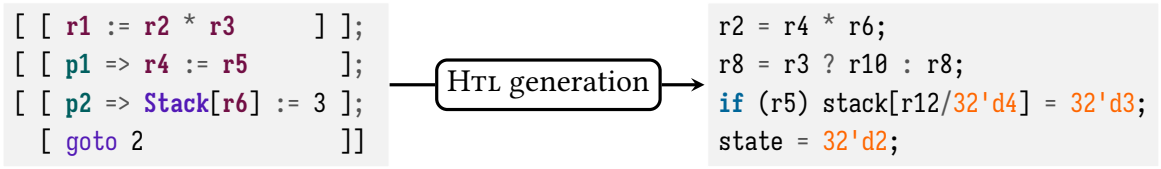


Figure 6.4: Simple translation from an RTLSUBPAR block into an HTL block.

$$\texttt{Oshrximm}\ x\ y = \text{round\_towards\_zero}\left(\frac{x}{2^y}\right) \tag{6.1}$$

(where $x, y \in \mathbb{Z}$, $0 \leq y < 31$, and $-2^{31} \leq x < 2^{31}$) and instantiating divider circuits in hardware is well known to cripple performance. Moreover, Vericert does not yet support pipelined or multi-cycle operations and therefore requires the result of a divide operation to be ready within a single clock cycle, meaning the divide circuit needs to be entirely combinational. This is inefficient in terms of area, but also in terms of latency, because it means that the maximum frequency of the hardware must be reduced dramatically so that the divide circuit has enough time to finish. It should therefore be implemented using a sequence of shifts.

CompCert eventually performs a translation from this representation into assembly code which uses shifts to implement the division, however, the specification of the instruction in RTL itself still uses division instead of shifts, meaning this proof of the translation cannot be reused. In Vericert, the equivalence of the representation in terms of divisions and shifts is proven over the integers and the specification, thereby making it simpler to prove the correctness of the Verilog implementation in terms of shifts.

Translating memory

Translating memory operations and the memory itself is one of the trickiest part of the translation, especially from a correctness point of view, because of the large difference in behaviour between CompCert memories and their Verilog implementation. At the stage of HTL generation, a Verilog array is used to represent the stack frame of the function in RTLSUBPAR. The Verilog array is defined as the following:

```
logic [31:0] stack [STK_LEN-1:0];
```

This is essentially an array of size `STK_LEN` of 32-bit integers. This array is therefore word-addressable. One big difference between C and Verilog is how memory is represented. Although Verilog arrays use similar syntax to C arrays, they must be treated quite differently.

Eventually, this array will have to be replaced by an actual BRAM, which only has a limited set of read and write ports (one of each in our case). To make loads and stores of words as efficient as possible, the BRAM needs to be word-addressable, which means that an entire integer can be loaded or stored in one clock cycle. However, the memory model that CompCert uses for its intermediate languages is byte-addressable [Blazy and Leroy 2005]. If a byte-addressable memory was used in the target hardware, which is closer to CompCert's memory model, then a load and store would instead take four clock cycles, because the BRAM implemented in hardware can only perform one read and write per clock cycle. It therefore has to be proven that the byte-addressable memory behaves in the same way as the word-addressable memory in hardware. Any modifications of the bytes in the CompCert memory model also have to be shown to modify the word-addressable memory in the same way. Since only integer loads and stores are currently supported in Vericert, it follows that the addresses given to the loads and stores will be multiples of four. Translating from byte-addressed memory to word-addressed memory can then be done by dividing the address by four.

As shown in figure 6.4, predicated instructions are translated into blocking assignments of a ternary expression in Verilog. This is the case for all instructions except for the store instruction, which is translated to a conditional statement. This ensures that the memory is only modified when the predicate is set. If that were not the case, and the memory was translated using a ternary statement:

```
p2 => Stack[r6] := 3   ⟶   stack[r12/32'd4] = r5 ? 32'd3 : stack[r12/32'd4]
```

Then, in the case where the predicate `p2` is *false*, that would mean that the statement in RTLSUBPAR would not be executed. However, in the generated Verilog, one would have to execute the statement `stack[r12/32'd4]`, which might not be possible as the register `r6` in RTLSUBPAR, and the corresponding expression `r12/32'd4` in Verilog could be out-of-range of the `stack` array. This would therefore mean that there is no way to execute the Verilog in that case, as one cannot read from the stack when it is out-of-range. Gating it fully with an if-statement, as shown in figure 6.4, ensures that the Verilog statement can always be executed, assuming that the RTLSUBPAR instruction can also be executed. However, this if-statement can be removed once the direct accesses to the array are translated into reads and writes to the BRAM interface.

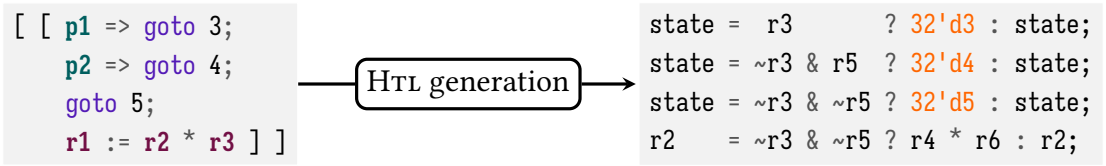


Figure 6.5: Describing the control flow translation from RTLSUBPAR to HTL.

Translating control flow instructions

Most control-flow instructions also map nicely to hardware, however, the way predicated control-flow instructions are handled differs a bit between RTLSUBPAR and HTL. The main problem is that when execution in a RTLSUBPAR block reaches a control-flow instruction that is executed, then it will exit the block immediately. This is not possible in the HTL block, because the next state is determined based on the value of the `state` register at the start of the next clock cycle. Naïvely translating the control-flow instructions into assignments to the `state` register would produce incorrect state transitions in HTL, because a later control-flow instruction could overwrite a previous instruction. As a remedy, we keep track of the current negated exit condition, which accumulates throughout the translation of a block. Figure 6.5 demonstrates this on a chain of control-flow operations. First, the condition of the ternary statement in Verilog corresponds to the predicate in the block, however, these quickly diverge. For the next control-flow instruction, the condition of the ternary statement is set to be the translated predicate associated with `p2`, anded together with the current negated exit condition which is `~r3`. The current negated exit condition is then updated to be `~r3 & ~r5`, which is then the ternary expression condition assigned to the last `state` update.

To avoid all side-effects after a gated control-flow instruction, regular instructions will have to be gated by the same negated control-flow predicate, but do not themselves modify the value of the negated control-flow predicate for any following instructions.

Finally, `return` instructions are translated to an assignment of 1 to the HTL finish register, signalling that the hardware has finished computing, and the result that should be returned is assigned to the return register.

Translating the top-level function

Each RTLSUBPAR function is translated separately, and within each function, each block is translated to a Verilog statement. In addition to that, new registers are created for the various control signals and registers in HTL, ensuring that they follow the ordering present

in the Htl specification. The top-level translation also needs to ensure that it is compiling a translation unit with the `main` function, as linking with other translation units is not supported by the Verilog semantics. As function calls are not supported, only the `main` function is needed, because all other functions are inlined.

6.2.3 Htl generation correctness proof

There is quite a large mismatch between the Htl semantics and the RtlSubPar semantics. This is mainly due to the following two points:

- As already mentioned in section 6.2.2, because the memory model in our Htl semantics is finite and concrete, but the CompCert memory model is more abstract and infinite with permissions that define the bounds of the memory block, the equivalence of these models needs to be proven. Moreover, our memory is word-addressed for efficiency reasons, whereas CompCert's memory is byte-addressed.
- Second, the Htl semantics operates quite differently to the usual intermediate languages in CompCert. All the CompCert intermediate languages use a map from control-flow nodes to instructions. An instruction can therefore be selected using an abstract program pointer. Meanwhile, in the Htl semantics the whole design is executed at every clock cycle, because hardware is inherently parallel. The program pointer is part of the design as well, not just part of an abstract state. This makes the semantics of Htl simpler, but comparing it to the semantics of Rtl becomes more challenging, as one has to map the abstract notion of the state to concrete values in registers.

As Htl is quite far removed from RtlSubPar, this first translation is the most involved and therefore requires a larger proof, because the translation from RtlSubPar instructions to Verilog statements needs to be proven correct in this step. Instead of defining small-step semantics for each construct in Verilog, the semantics are defined over one clock cycle and mirror the semantics defined for Verilog. Lemma 6.1 shows the result that needs to be proven in this subsection.

Lemma 6.1 (Forward simulation from RtlSubPar to Htl)**.** *Writing* `tr_htl` *for the translation from RtlSubPar to Htl, we have:*

$$\forall c, h, \mathcal{B} \notin \texttt{Wrong}, \quad \texttt{tr\_htl}(c) = \lfloor h \rfloor \wedge c \Downarrow \mathcal{B} \implies h \Downarrow \mathcal{B}. \tag{6.2}$$

Proof sketch. I prove this lemma by first establishing a specification of the translation function `tr_htl` that captures its important properties, and then splitting the proof into two parts: one to show that the translation function does indeed meet its specification, and one to show that the specification implies the desired simulation result. This strategy is in keeping with standard CompCert practice.

□

Forward simulation proof of translation

To prove that the translation described in section 6.2 results in the desired forward simulation, we must first define a relation that matches each RtlSubPar state to an equivalent Htl state. This relation also captures the assumptions made about the RtlSubPar code that we receive from CompCert. These assumptions then have to be proven to always hold assuming the Htl code was created by the translation algorithm. Some of the assumptions that need to be made about the RtlSubPar and Htl code for a pair of states to match are:

- The RtlSubPar register file R needs to be 'less defined' than the Htl register map Γ_r (written $R \leq \Gamma_r$). This means that all entries should be equal to each other, unless a value in R is undefined, in which case any value can match it.
- There is a single allocation that was performed in RtlSubPar, with the size of the allocation being equal to the stack size of the `main` function, which was performed when the `main` function was initially called; that is: $|M| \leq \texttt{main}.\textit{stacksize}$.
- The BRAM values represented by each Verilog array in Γ_a need to match the RtlSubPar function's stack contents, which are part of the memory M; that is: $M \leq \Gamma_a$.
- The state is well formed, which means that the value of the state register matches the current value of the program counter; that is: $pc = \Gamma_r[st]$.

We also define the following set $\mathcal{I}$ of invariants that must hold for the current state to be valid:

- all pointers in the program use the stack as a base pointer,
- any loads or stores to locations outside of the bounds of the stack result in undefined behaviour (and hence they do not need to be handled),

- *rst* and *fin* are not modified and therefore stay at a constant 0 throughout execution, and
- the stack frames match. As no function calls are performed, as they are all inlined, the stack frames will always be empty.

We can now define the simulation diagram for the translation. The RtlSubPar state can be represented by the tuple (R, M, pc), which captures the register file, memory, and program counter. The Htl state can be represented by the pair (Γ_r, Γ_a), which captures the states of all the registers and arrays in the module. Finally, $\mathcal{I}$ stands for the other invariants that need to hold for the states to match.

Lemma 6.2. *Given the RtlSubPar state* (R, M, pc) *and the matching Htl state* (Γ_r, Γ_a)*, assuming one step in the RtlSubPar semantics produces state* (R', M', pc')*, there exist one or more steps in the Htl semantics that result in matching states* (Γ'_r, Γ'_a)*. This is all under the assumption that the specification* `spec_htl` *holds for the translation.*

$$\begin{array}{ccc} R, M, pc & \xrightarrow{\mathcal{I} \wedge (R \leq \Gamma_r) \wedge (M \leq \Gamma_a) \wedge (pc = \Gamma_r[st]) \wedge |M| \leq \texttt{main.}stacksize} & \Gamma_r, \Gamma_a \\ \downarrow & & \downarrow{\scriptstyle +} \\ R', M', pc' & \xrightarrow{\mathcal{I} \wedge (R' \leq \Gamma'_r) \wedge (M' \leq \Gamma'_a) \wedge (pc' = \Gamma'_r[st]) \wedge |M'| \leq \texttt{main.}stacksize} & \Gamma'_r, \Gamma'_a \end{array}$$

Proof sketch. This simulation diagram is proven by induction over the operational semantics of RtlSubPar, which allows us to find one or more steps in the Htl semantics that will produce the same final matching state. The 'plus' simulation diagram is needed because the return instruction is translated into register assignments, which first need to be executed normally and a separate step is needed to move to the return state. □

6.3 BRAM insertion

The simplest way to implement loads and stores in Vericert would be to access the Verilog array directly from within the data path as is currently the case after the Htl generation. This would be correct, but when a Verilog array is accessed at several program points, the synthesis tool is unlikely to detect that it can be implemented as a BRAM, and will resort

```
// BRAM interface
(* ram_style = "block" *)
logic [31:0] stack [1:0];
always @(negedge clk)
  if ({u_en != en}) begin
    if (wr_en) stack[addr] <= d_in;
    else d_out <= stack[addr];
    en <= u_en;
  end
```

Figure 6.6: Verilog implementation of the BRAM interface generated by Vericert.

to using registers instead, increasing the circuit's area and affecting performance. This is because reads and writes to a BRAM need to follow a certain pattern to be suitable to be replaced by BRAM reads and writes. For example, the synthesis tool will have to check that the array is only written to once per clock cycle, and is only read from once as well. To avert this, we arrange that the data path does not access memory directly, but instead accesses the memory through the BRAM interface. By factoring all the memory accesses out into a separate interface, we ensure that the underlying array is only accessed from a single program point in the Verilog code, and thus ensure that the synthesis tool will correctly infer a BRAM block.[1]

There are two interesting parts to the inserted BRAM interface, where the final Verilog implementation is shown in figure 6.6. Firstly, the memory updates are triggered on the negative (falling) edge of the clock, out of phase with the rest of the design which is triggered on the positive (rising) edge of the clock. The advantage of this is that instead of loads and stores taking three clock cycles and one clock cycles respectively, they only take one clock cycle and half a clock cycle instead, greatly improving the performance. Using the negative edge of the clock is supported by synthesis tools and FPGAs, but in general it reduces the time that is available for arithmetic operations in the positive edge of the clock, making it harder to schedule operations.

Secondly, the logic in the enable signal of the BRAM (`en != u_en`) is also atypical in hardware designs. Enable signals are normally manually controlled and inserted into the appropriate states, by using a check like the following in the BRAM: `en == 1`. This means that the BRAM only turns on when the enable signal is set. However, to make the proof simpler and avoid reasoning about possible side effects introduced by the BRAM being enabled but not used, a BRAM which disables itself after every use would be ideal. One

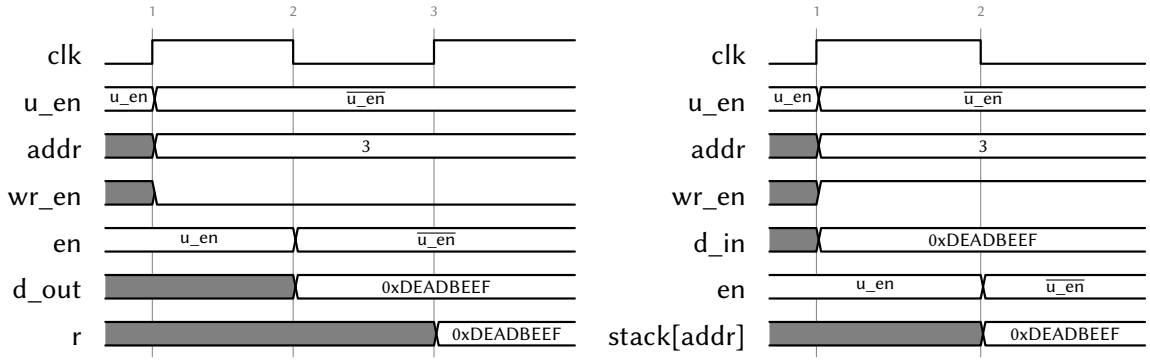
[1] Interestingly, the Verilog code shown for the BRAM interface must not be modified, because the synthesis tool will only generate a BRAM when the code matches a small set of specific patterns.

method for implementing this would be to insert an extra state after each load or store that disables the BRAM, but this extra state would eliminate the speed advantage of the negative-edge-triggered BRAM. Another method would be to determine the next state after each load or store and disable the BRAM in that state, but this could quickly become complicated, especially in the case where the next state also contains a memory operation, and hence the disable signal should not be added. The method I ultimately chose was to have the BRAM become enabled not when the enable signal is high, but when it *toggles* its value. This can be arranged by keeping track of the old value of the enable signal in `en` and comparing it to the current value `u_en` set by the data path. When the values are different, the BRAM gets enabled, and then `en` is set to the value of `u_en`. This ensures that the BRAM will always be disabled straight after it was used, without having to insert or modify any other states.

This transformation pass therefore translates direct accesses to the Verilog array in Htl and replaces them by signals that access the BRAM interface in a separate always-block. The translation is performed by going through all the instructions and replacing each load and store expression in turn. Stores can be replaced by the necessary wires to the BRAM directly. Loads are a little more subtle: loads that use the BRAM interface take one clock cycles where a direct load from an array happens instantly, so this pass inserts an extra state after each load. The scheduling algorithm described in section 5.4 can already take this into account as well, and can ensure that the next clock cycle after a load does not perform a load or a store, however, this transformation currently does not take advantage of that.

6.3.1 BRAM model semantics

Figure 6.7 gives an example of how the BRAM interface behaves when values are loaded and stored. There is a `wr_en` signal that determines if a load or store is being performed. Then, if at the falling edge `u_en` and `en` are different, the read or write at address `addr` is executed. At the same time, the value of `u_en` is then assigned to `en`, which would disable it if there is no other action by the data path on the next clock cycle. However, if the data path toggles the value of `u_en` on the next clock cycle, the BRAM would be enabled again.



(a) Timing diagram for loads. At time 1, the `u_en` signal is toggled to enable the BRAM. At time 2, `d_out` is set to the value stored at the address in the BRAM, which is finally assigned to the register `r` at time 3.

(b) Timing diagram for stores. At time 1, the `u_en` signal is toggled to enable the BRAM, and the address `addr` and the data to store `d_in` are set. On the negative edge at time 2, the data is stored into the BRAM.

Figure 6.7: Timing diagrams showing the execution of loads and stores over multiple clock cycles.

$$\textsc{Idle}\quad \frac{\Gamma_r[r.\text{en}] = \Gamma_r[r.\text{u\_en}]}{((\Gamma_r, \Gamma_a), \Delta, r) \downarrow_{\text{ram}} \Delta}$$

$$\textsc{Load}\quad \frac{\Gamma_r[r.\text{en}] \neq \Gamma_r[r.\text{u\_en}] \qquad \Gamma_r[r.\text{wr\_en}] = 0}{((\Gamma_r, \Gamma_a), (\Delta_r, \Delta_a), r) \downarrow_{\text{ram}} (\Delta_r[r.\text{en} \mapsto r.\text{u\_en}, r.\text{d\_out} \mapsto (\Gamma_a[r.\text{mem}])[r.\text{addr}]], \Delta_a)}$$

$$\textsc{Store}\quad \frac{\Gamma_r[r.\text{en}] \neq \Gamma_r[r.\text{u\_en}] \qquad \Gamma_r[r.\text{wr\_en}] = 1}{((\Gamma_r, \Gamma_a), (\Delta_r, \Delta_a), r) \downarrow_{\text{ram}} (\Delta_r[r.\text{en} \mapsto r.\text{u\_en}], \Delta_a[r.\text{mem} \mapsto (\Gamma_a[r.\text{mem}])[r.\text{addr} \mapsto r.\text{d\_in}]])}$$

Figure 6.8: Specification for the memory implementation in Htl, where r is the BRAM, which is then implemented by equivalent Verilog code.

6.3.2 BRAM insertion and correctness proof

HTL can only represent a single state machine, so we must model the BRAM abstractly to reason about the correctness of replacing the direct read and writes to the array by loads and stores to a BRAM. The specification for the BRAM is shown in figure 6.8, which defines how the BRAM r will behave for all the possible combinations of the input signals. This specification is part of the HTL semantics and runs in parallel to the state machine. However, as the BRAM is triggered by the falling edge of the clock, it will execute in between standard clock cycles, and a merge of the association maps is performed in between each one.

From implementation to specification

The first step in proving the simulation correct is to build a specification of the translation algorithm. There are five possibilities for the transformation of an instruction. For each Verilog statement in the map at location i, the statement is either a load, a store, a predicated load, a predicated store, or neither. The load, store, predicated load and predicated store is translated to the equivalent representation using the BRAM specification and all other instructions are left intact. The specification of the translation is shown in figure 6.9, where st is state register, r is the BRAM, d is the data path map of the original HTL, d' is the data path of the translated HTL, and i is the current state. The newly inserted state is denoted by n, which only applies to the translation of loads. The specification shown in the figure relates the original HTL data path d with the data path d' in which BRAM accesses were inserted. For each type of direct memory access in the original HTL, there is a rule relating it to the translated data path in terms of BRAM control signals. Note that the translation is quite strict, and the only statements that are allowed to be in the HTL state are the ones that are listed in the relation. Ideally, arbitrary statements could be placed before or after the direct memory accesses, however, proving that the translation is semantics preserving is more difficult due to the fact that the BRAM stores values in the next negative edge and loads values in the next cycle. One would therefore have to show that there are no conflicting direct memory accesses in the state.

Conditional loads and stores are also handled, by adding logic to the enable signal which inhibits the flip of the r.u_en when the condition does not hold. This is implemented by the following logic, where $\oplus$ is the `xor` operation:

$$r\text{.u\_en} = (c \texttt{ != 32'b0}) \oplus r\text{.u\_en}; \tag{6.3}$$

If c is *false*, then this means that `32'b0!=32'b0` evaluates to `1'b0`, and `1'b0`$\oplus r$.u_en = r.u_en, meaning the value of r.u_en remains unchanged. However, if c evaluates to true, then the `xor` operation acts like a toggle, meaning the memory is activated normally.

This specification has to be shown to hold with respect to the top-level BRAM insertion transformation function `tr_ram_ins`. We therefore need to show that the following lemma holds so that the correctness theorem can only reason about the cases in the specification. This is especially useful in this transformation, because the specification states that each statement in the original Htl only has five cases it could be transformed into, and this is independent of any other statement in the data path.

Lemma 6.3 (BRAM insertion specification holds)**.** *We need to show that given that the translation succeeded, that the specification of the translation holds for every statement in the data path.*

$$\begin{aligned} &\forall h\ h'.\quad \texttt{tr\_ram\_ins}(h) = (h') \wedge h'.ram = \lfloor r \rfloor \implies \\ &\qquad \forall i.\ \exists n.\quad \texttt{spec\_ram\_tr}\ h.st\ r\ h.datapath\ h'.datapath\ i\ n \end{aligned} \tag{6.4}$$

From specification to simulation

Another simulation proof is performed to prove that the insertion of the BRAM is a forward simulation. As in lemma 6.2, we require some invariants that always hold at the start and end of the simulation. The invariants needed for the simulation of the BRAM insertion are quite different to the previous ones, so we can define these invariants $\mathcal{I}_\mathrm{r}$ to be the following:

- The association map for arrays Γ_a always needs to have the same arrays present, and these arrays should never change in size.
- The BRAM should always be disabled at the start of each simulation step. (This is why self-disabling BRAM is needed.)

The other invariants and assumptions for defining two matching states in Htl are quite similar to the simulation performed in lemma 6.2, such as ensuring that the state registers have the same value, and that the values in the registers are less defined. In particular, the less defined relation matches up all the registers, except for the new registers introduced by the BRAM.

Store Transl

$$\frac{d[i] = \left(\begin{array}{l} r.\text{mem}[e_1] = e_2; \\ st = e_3; \end{array}\right) \qquad d'[i] = \left(\begin{array}{l} r.\text{u\_en} = \neg r.\text{u\_en}; \\ r.\text{wr\_en} = 1; \\ r.\text{d\_in} = e_2; \\ r.\text{addr} = e_1; \\ st = e_3; \end{array}\right)}{\texttt{spec\_ram\_tr}\ st\ r\ d\ d'\ i\ n}$$

Predicated Store Transl

$$\frac{d[i] = \left(\begin{array}{l} \texttt{if}(c)\ r.\text{mem}[e_1] = e_2; \\ st = e_3; \end{array}\right) \qquad d'[i] = \left(\begin{array}{l} r.\text{u\_en} = (c\ \texttt{!=}\ \texttt{32'b0}) \oplus r.\text{u\_en}; \\ r.\text{wr\_en} = \texttt{1'b1}; \\ r.\text{d\_in} = c\ ?\ e_2 : r.\text{d\_in}; \\ r.\text{addr} = c\ ?\ e_1 : r.\text{addr}; \\ st = e_3; \end{array}\right)}{\texttt{spec\_ram\_tr}\ st\ r\ d\ d'\ i\ n}$$

Load Transl

$$\frac{d[i] = \left(\begin{array}{l} r_\text{d} = r.\text{mem}[e_1]; \\ st = e_2; \end{array}\right) \qquad \begin{array}{l} d'[i] = \left(\begin{array}{l} r.\text{u\_en} = \neg r.\text{u\_en}; \\ r.\text{wr\_en} = \texttt{1'b0}; \\ r.\text{addr} = e_1; \\ st = n; \end{array}\right) \\ d'[n] = \left(\begin{array}{l} r_\text{d} = r.\text{d\_out}; \\ st = e_2; \end{array}\right) \end{array}}{\texttt{spec\_ram\_tr}\ st\ r\ d\ d'\ i\ n}$$

Predicated Load Transl

$$\frac{d[i] = \left(\begin{array}{l} r_\text{d} = c\ ?\ r.\text{mem}[e_1] : r_\text{d}; \\ st = e_2; \end{array}\right) \qquad \begin{array}{l} d'[i] = \left(\begin{array}{l} r.\text{u\_en} = (c\ \texttt{!=}\ \texttt{32'b0}) \oplus r.\text{u\_en}; \\ r.\text{wr\_en} = \texttt{1'b0}; \\ r.\text{addr} = c\ ?\ e_1 : r.\text{addr}; \\ st = n; \end{array}\right) \\ d'[n] = \left(\begin{array}{l} r_\text{d} = c\ ?\ e_1 : r.\text{d\_out}; \\ st = e_2; \end{array}\right) \end{array}}{\texttt{spec\_ram\_tr}\ st\ r\ d\ d'\ i\ n}$$

Default Transl

$$\frac{\begin{array}{c} (\forall e_1\ e_2\ e_3.\ d[i] \neq (r.\text{mem}[e_1] = e_2;\ st = e_3)) \\ (\forall c\ e_1\ e_2\ e_3.\ d[i] \neq (\texttt{if}(c)\ r.\text{mem}[e_1] = e_2;\ st = e_3)) \\ (\forall r_\text{d}\ e_1\ e_2.\ d[i] \neq (r_\text{d} = r.\text{mem}[e_1];\ st = e_2)) \\ (\forall c\ r_\text{d}\ e_1\ e_2.\ d[i] \neq (r_\text{d} = c\ ?\ r.\text{mem}[e_1] : r_\text{d};\ st = e_2)) \qquad d[i] = d'[i] \end{array}}{\texttt{spec\_ram\_tr}\ st\ r\ d\ d'\ i\ n}$$

Figure 6.9: Memory transformation specification.

Lemma 6.4 (Forward simulation from Htl to Htl after inserting the BRAM)**.** *Given an Htl program, the forward-simulation relation should hold after inserting the BRAM and wiring the load, store, and control signals.*

$$\forall h,\ h',\ \mathcal{B} \notin \texttt{Wrong}. \quad \texttt{tr\_ram\_ins}(h) = h' \wedge h \Downarrow \mathcal{B} \implies h' \Downarrow \mathcal{B}. \tag{6.5}$$

Proof sketch. By using the specification defined earlier, one can handle the five transformation cases independently. The trickiest part of the proof is dealing with the rearrangement of the order in which assignments are performed, as well as the different times at which the memory is accessed. This transformation requires a 'plus' simulation diagram because the loads require one cycle to complete. □

6.4 Register Forward Substitution

Until now, only blocking assignment has been used to assign variables sequentially, being more faithful to the instructions provided by the input language. The transformation is shown in figure 6.10, and it turns sequential, blocking assignment into parallel, nonblocking assignment by substituting register definitions within a state. Each assignment in the translated block is independent from the other blocks, meaning each assignment can be executed in parallel. Note that the `state` is assigned twice using nonblocking assignment. In that case, only the last nonblocking assignment is kept. In addition to that, `r2` is no longer used in the block, as it has been replaced by its definition in the assignment to `r8`. It is likely that `r2` is not used in any other block either, in which case it should be removed. This is not performed by this transformation pass, but in practice synthesis tools can reliably remove a register that is assigned and never referenced in the design. There are two reasons why this transformation is needed.

1. In a clocked always-block, nonblocking assignment should be used for all registers that could interact with other always-blocks. As we have a BRAM block in a separate always-block, we should at least use nonblocking assignment for the registers that interact with memory. In this case, it is not actually required, because the BRAM is executing on the negative edge of the clock, however, if it is ever replaced by a BRAM that executes on the positive edge of the clock, or supplemented with other functional units that execute on the positive edge of the clock, then any registers communicating with those blocks will need to use nonblocking assignment.

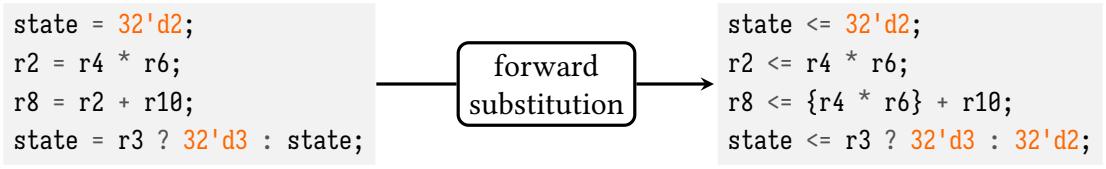


Figure 6.10: Simple example of the forward substitution transformation.

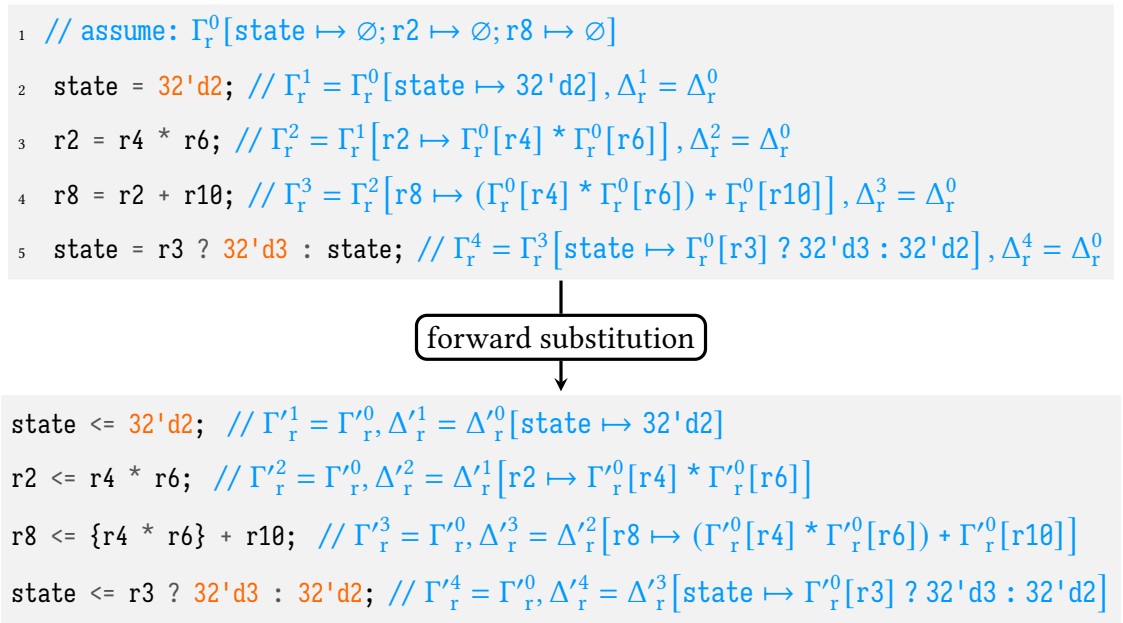


Figure 6.11: Simple forward substitution transformation together with the runtime value of the blocking assignment association map (Γ_r) and the nonblocking assignment association map (Δ_r).

2. Some synthesis tools do not seem to be able to optimise the designs generated by Vericert with blocking assignments, as they seem to remove less unused registers, resulting in slightly lower maximum frequency and slightly larger area.[2] In the case of the example given in figure 6.10, it seems like some synthesis tool versions do not remove r2 when it is referenced in the assignment to r8 and not referenced anywhere else in the design. This should not be the case as in this example r2 should just become a wire.

The transformation is performed by traversing each block and storing for each register the expression that is being assigned to it. If the same register is encountered multiple times, the expression being assigned is always substituted first and then replaces the

[2]This seems to be the case with Vivado 2017.1 and seems to have been fixed by Vivado 2023.1.

current mapping from register to expression. As the BRAM insertion already removed all the load and store operations in the data path, only regular register assignments need to be accounted for.

First, I describe the substitution of expressions in definition 6.1, and then I describe the substitution of statements in definition 6.2.

Definition 6.1 (Substitute expressions)**.** Expressions are substituted based on a map t from registers to Verilog expressions. Each register within the expression is replaced by the expression in the map.

$$\textsc{SubstReg}\ \frac{t[r] = \lfloor e' \rfloor}{\texttt{subst\_expr}\ t\ r = e'} \qquad \textsc{SubstRegNotIn}\ \frac{t[r] = \texttt{None}}{\texttt{subst\_expr}\ t\ r = r}$$

$$\textsc{SubstBinaryOp}\ \frac{}{\texttt{subst\_expr}\ t\ (e_1 + e_2) = \texttt{subst\_expr}\ t\ e_1 + \texttt{subst\_expr}\ t\ e_2} \qquad \ldots$$

Definition 6.2 (Substitute statements)**.** Only two types of statements need to be substituted, blocking assignments and sequential composition of statements. For blocking assignments, substitution means that it is turned into nonblocking assignment by substituting the expression with the current expression substitution map t, in addition to updating the map itself with a new expression mapping for the register r.

$$\textsc{SubstBlocking}\ \frac{e' = \texttt{subst\_expr}\ t\ e}{\texttt{subst\_stmnt}\ t\ (r = e) = \lfloor r <= e', t[r \mapsto e'] \rfloor}$$

$$\textsc{SubstSeq}\ \frac{\texttt{subst\_stmnt}\ t\ s_1 = \lfloor s_1', t' \rfloor \qquad \texttt{subst\_stmnt}\ t'\ s_2 = \lfloor s_2', t'' \rfloor}{\texttt{subst\_stmnt}\ t\ (s_1\texttt{;}\ s_2) = \lfloor s_1'\texttt{;}\ s_2', t'' \rfloor}$$

$$\textsc{SubstOther}\ \frac{(\forall s_1\ s_2.\ s \neq s_1\texttt{;}\ s_2) \qquad (\forall r\ e.\ s \neq r = e)}{\texttt{subst\_stmnt}\ t\ s = \texttt{None}}$$

It might seem restrictive to only support sequential composition of statements and

blocking assignment, but those are the only constructs that are generated by the HTL generation. Supporting additional statements would not require a significant change to the transformation. Even adding support for nonblocking assignment in the input statement should be safe, as long as the expressions are correctly substituted, and are *not* added to the expression substitution map t. The main property one would have to check about the input statement is that a register is not assigned using blocking assignment after a nonblocking assignment, as otherwise the input and output statement would have different results.

6.4.1 Forward substitution correctness proof

The main lemma that is needed to prove the forward simulation correct is that the execution of statements in each state results in equivalent *merged* association maps from the blocking and nonblocking assignments. Figure 6.11 shows the forward substitution example with annotated runtime states to explain the proof and invariants needed to prove the forward simulation for this transformation. The correctness argument in figure 6.11 corresponds to showing that $\Gamma^4_{\mathrm{r}} \mathbin{//} \Delta^0_{\mathrm{r}} = \Gamma'^0_{\mathrm{r}} \mathbin{//} \Delta'^4_{\mathrm{r}}$.

Lemma 6.5 (Equivalence of statement substitution)**.** *After merging the maps of a statement s and the substituted statements s', the contents of the merged blocking association map Γ' and nonblocking association map Δ' after executing s should be equivalent to merging the maps Γ'' and Δ'' after executing s'.*

$$\begin{aligned}
&\texttt{subst_stmnt}\ t\ s = \lfloor s', t' \rfloor \implies \\
&((\Gamma, \Delta), s) \downarrow_{\text{stmnt}} (\Gamma', \Delta') \implies \\
&((\Gamma, \Delta), s') \downarrow_{\text{stmnt}} (\Gamma'', \Delta'') \implies \\
&\Gamma' \mathbin{//} \Delta' = \Gamma'' \mathbin{//} \Delta''
\end{aligned} \tag{6.6}$$

Proof sketch. By induction on the definition of a statement, by applying lemma 6.6 to show equivalence of expressions within the statements. □

There are two main invariants that have to be maintained when proving the lemma above, which relates the run time association map used to execute expressions encountered during the execution of the original statement with the contents of the expression substitution map t. The first invariant, described in definition 6.3, states that the evaluation of an expression in the substitution map t should correspond to the value associated with that register in the current execution of the block. For example, in figure 6.11 after line 3, the substitution map

would contain the entry: $t[\texttt{r2} \mapsto t[\texttt{r4}] * t[\texttt{r6}]]$. Executing this expression $t[\texttt{r4}] * t[\texttt{r6}]$ with the initial blocking assignment at the start of the statement Γ_r^0 should be the same as indexing the current run time association map at the register r2, i.e. $\Gamma_r^2[\texttt{r2}]$, which is the case.

Definition 6.3 (In substitution map)**.** If the register r is in the substitution map t and it maps to expression e, then the value in the current association map Γ_r should be the same as the value obtained from evaluating expression e with the initial association map Γ_r^0.

$$\begin{aligned}&\texttt{in\_subst\_map}\ \Gamma_r^0\ \Gamma_r\ \Gamma_a\ t \triangleq \\ &\quad \forall r\ e.\ t[r] = \lfloor e \rfloor \implies \exists v.\ ((\Gamma_r^0, \Gamma_a), e) \downarrow_{\text{expr}} v \wedge \Gamma_r[r] = \lfloor v \rfloor\end{aligned} \tag{6.7}$$

Definition 6.4 describes the relationship between the initial map and the current run time map when a register is not in the expression substitution map. For example, this would be the case for register r2 before line 3. In that case, r2 is not in t and so $\Gamma_r^1[\texttt{r2}] = \Gamma_r^0[\texttt{r2}]$.

Definition 6.4 (Not in substitution map)**.** If the register r is not in the substitution map t, then the value for that register in the blocking assignment association map Γ_r should be the same as the value of the register in the initial association map Γ_r^0 at the start of the execution of the statement.

$$\begin{aligned}&\texttt{not\_in\_subst\_map}\ \Gamma_r^0\ \Gamma_r\ t \triangleq \\ &\quad \forall r.\ t[r] = \texttt{None} \implies \Gamma_r^0[r] = \Gamma_r[r]\end{aligned} \tag{6.8}$$

These invariants are then used to prove the relationship between substituted expressions and the original expressions. This is used to prove the correctness of forward substitution of statements.

Lemma 6.6 (Forward substitution of expressions)**.** *Given a map from registers to expression t, an expression before forward substitution e, and the result of forward substituting e with map t resulting in expression e', then executing e with the dynamically updated association map Γ_r should be equivalent to executing e' with the initial state of all the registers Γ_r^0.*

$$\begin{aligned}&\texttt{subst\_expr}\ t\ e = \lfloor e' \rfloor \implies \\ &\texttt{in\_subst\_map}\ \Gamma_r^0\ \Gamma_r\ \Gamma_a\ t \implies \\ &\texttt{not\_in\_subst\_map}\ \Gamma_r^0\ \Gamma_r\ t \implies \\ &\forall v.\ ((\Gamma_r, \Gamma_a), e) \downarrow_{\text{expr}} v \implies ((\Gamma_r^0, \Gamma_a), e') \downarrow_{\text{expr}} v\end{aligned} \tag{6.9}$$

Proof sketch. Using the invariants, one can show that the modified expression e' will behave like the original expression e at the current context (Γ_r, Γ_a), when it is evaluated in the initial context at the start of the state, i.e. (Γ_r^0, Γ_a). □

6.5 Verilog Generation

Finally, Verilog generation produces proper Verilog from HTL. The main two transformations that take place are:

1. Converting the mapping from states to Verilog statements into a case statement, with reset logic to reset the state.
2. Instantiating the BRAM specification as a Verilog always-block.

This translation is shown in figure 6.12, where the HTL design shown in figure 6.12a is translated to the Verilog shown in figure 6.12b. The main state machine is then turned into a single always-block with an explicit `reset`, setting the state to the starting state present in HTL. If the `reset` is not set, then there is a case statement that implements the state machine from HTL, where each state has the same Verilog statements as the corresponding state in HTL.

In addition to that, the implicit memory that is part of the HTL semantics is made explicit by a separate always-block implementing the memory interface. This interface follows a standard BRAM memory template, and can therefore be detected by the synthesis tool and will become a proper memory in the synthesised netlist.

6.5.1 Forward simulation from HTL to Verilog

The HTL-to-Verilog simulation is conceptually simple, as the only transformation is from the map representation of the code to the case-statement representation. The proof is more involved, as the semantics of a map structure is quite different to that of the case-statement to which it is converted.

Lemma 6.7 (Forward simulation from HTL to Verilog)**.** *In the following, I write* `tr_verilog` *for the translation from HTL to Verilog. (Note that this translation cannot fail, so we do not need the* $\lfloor \cdot \rfloor$ *constructor here.)*

$$\forall h\ V\ \mathcal{B} \notin \texttt{Wrong}. \quad \texttt{tr\_verilog}(h) = V \wedge h \Downarrow \mathcal{B} \implies V \Downarrow \mathcal{B}. \tag{6.10}$$

```
main() {
  8: {
    state <= 32'd18;
    u_en <= ( ~ u_en);
    wr_en <= 32'd0;
    addr <= {{{reg_6 + 32'd0}
              + {reg_2 * 32'd4}}
             / 32'd4};
  }
}
```

(a) Original HTL representation of the design, with the starting state set to 8.

```
module main(reset, clk, finish, return_val);
  // Register declarations
  // ...

  // BRAM interface
  (* ram_style = "block" *)
  logic [31:0] stack [1:0];
  always @(negedge clk)
    if ({u_en != en}) begin
      if (wr_en) stack[addr] <= d_in;
      else d_out <= stack[addr];
      en <= u_en;
    end

  // Finite-state machine with data path
  always @(posedge clk)
    if ({reset == 32'd1}) state <= 32'd8;
    else
      case (state)
        32'd8: begin
          state <= 32'd18;
          u_en <= ( ~ u_en);
          wr_en <= 32'd0;
          addr <= {{{reg_6 + 32'd0}
                    + {reg_2 * 32'd4}}
                   / 32'd4};
        end
        default:;
      endcase
endmodule
```

(b) Translated Verilog design with the explicit reset and the explicit memory instantiation.

Figure 6.12: Instantiation of BRAM specification with Verilog implementation.

Proof sketch. The translation from maps to case-statements is done by turning each node of the tree into a case-expression containing the same statements. The main difficulty is that a random-access structure is being transformed into an inductive structure where a certain number of constructors need to be called to get to the correct case. In addition to that, we need to prove that the BRAM template implements the BRAM specification and that the negative edge trigger matches the HTL semantics of executing the memory. □

6.6 Summary

This chapter described the translation from the scheduled code, which was still represented in a language with software semantics, down to Verilog with faithful hardware semantics. On the way, some hardware specific optimisations and transformations had to be performed. First, any direct assignments to the memory had to be translated to a more standard interaction with memory, using read and write ports of a BRAM. Next, any blocking assignment had to be turned into nonblocking assignment to parallelise the Verilog design, helping improve synthesis results in some cases. Finally, proper Verilog had to be generated, implementing the implicit behaviour present in HTL as explicit behaviour in Verilog.

7 Evaluation

The evaluation aims to answer the following research questions:

RQ1 Is Vericert competitive with unverified HLS tools?

RQ2 Does adding scheduling to Vericert lead to a significant improvement in the quality of the generated hardware (in terms of area and delay)?

RQ3 Is hyperblock scheduling better than naïve list scheduling?

RQ4 Did the design decisions (e.g. section 5.5.3) lead to an acceptable compilation time?

RQ5 How effective is the correctness theorem in Vericert?

7.1 Experimental Setup

Choice of HLS tool for comparison. Vericert is compared against Bambu HLS 2023.1 [Ferrandi et al. 2021], because it is open-source and hence easily accessible, but it can still 'produce faster accelerators in almost all the cases' (Pilato and Ferrandi [2013]) when compared to LegUp HLS, which itself is said to produce hardware 'of comparable quality to a commercial high-level synthesis tool' (Canis et al. [2011]). Even though these quotes are quite old, Bambu is still in active development and implements many advanced optimisations like speculation. The baseline Bambu HLS version has all the default automatic optimisations turned on. Vericert is also compared against an unoptimised version of Bambu HLS, where all optimisations that can be toggled are turned off. This affects optimisations like deep if-conversion that Bambu HLS would normally automatically perform. Additionally, three versions of Vericert are compared to Bambu HLS, so that optimisations performed by Vericert can also be tested. In the first version of Vericert scheduling is disabled completely, generating sequential hardware where an instruction is executed every clock cycle. Next, a version of Vericert with list scheduling is implemented by disabling

if-conversion, thereby only scheduling basic blocks. Finally, a version of Vericert with the full hyperblock scheduling optimisation is shown, including all optimisations that are implemented.

Choice and preparation of benchmarks. I evaluate Vericert using the PolyBench/C benchmark suite (version 4.2.1) [Pouchet 2020], which is a collection of 30 numerical kernels. PolyBench/C is popular in the HLS context [Choi and Cong 2018; Pouchet et al. 2013; Zhao et al. 2017; Zuo et al. 2013], since it has affine loop bounds, making it attractive for streaming computation on FPGA architectures. It was also used as part of Six et al. [2022] evaluation of their scheduling optimisation. I was able to use 27 of the 30 programs; three had to be discarded (correlation, gramschmidt and deriche) because they involve square roots, requiring floats, which Vericert does not support. I configured PolyBench/C's parameters so that only integer types are used and use PolyBench/C's smallest datasets for each program to ensure that data can reside within on-chip memories of the FPGA, avoiding any need for off-chip memory accesses. The benchmarks have not been modified to make them run through Bambu HLS optimally, e.g. by adding pragmas that trigger more advanced optimisations.

Vericert implements divisions and modulo operations in C using the corresponding built-in Verilog operators. These built-in operators are designed to complete within a single clock cycle, and this causes substantial penalties in clock frequency. Other HLS tools, including Bambu, supply their own multi-cycle division/modulo implementations, and I plan to do the same in future versions of Vericert. In the meantime, I have prepared an alternative version of the benchmarks in which each division/modulo operation is replaced with my own implementation that uses repeated division and multiplications by 2. Comparing Vericert against Bambu HLS on benchmarks with division without using the C implementation of division would be futile, as preliminary tests show that because of the use of the default combinational divider circuit generated by the synthesis tool, the maximum clock frequency achieved by Vericert is around 30× less than the maximum clock frequency achieved by Bambu, which uses a pipelined division implementation. The following comparisons are therefore performed on a modified version of PolyBench/C with divisions replaced with function calls to our C divider implementation. The final result produced by the benchmarks remain unchanged.

Synthesis setup For each benchmark, the resulting Verilog hardware design was simulated using Verilator to get the total cycle count. Each design was synthesised, placed,

and routed onto a Xilinx series 7 FPGA (part number: `xc7z020clg484-1`) using Vivado 2023.1 to get its total area and its maximum frequency. Next, the total execution time was calculated as total execution time = $\frac{\text{total clock cycles}}{\text{maximum frequency}}$. I ensured that every design met the timing constraints of a 100MHz clock.

Figure 7.1 visualises the results of simulation and synthesis of the PolyBench/C benchmark. We use default Vericert (Vericert-original), Vericert with list scheduling (Vericert-list), and Vericert with hyperblock scheduling (Vericert-hyperblock). We also use the state-of-the-art open-source HLS tool Bambu in two modes: one where all default optimisations are enabled (Bambu-default), and one where as many optimisations as possible are disabled (Bambu-no-opt). Several optimisations are built into Bambu though and cannot be disabled, such as list scheduling and loop flattening.

7.2 RQ1: Is Vericert Competitive With Unverified Tools

To assess how Vericert-hyperblock fares against the unverified HLS tool Bambu. Bambu is used in two modes: one where all default optimisations are enabled (Bambu-default), and one where as many optimisations as possible are disabled (Bambu-no-opt). Note that several 'optimisations' are built into Bambu and cannot be disabled, such as list scheduling and loop flattening.

Most the bars in figure 7.1 are relative to Bambu-default. The pink bars show Bambu-no-opt. We see that although Vericert-hyperblock is well behind Bambu-default (its designs require 3.6× the execution time), it only performs slightly worse when compared to Bambu-no-opt (1.57× the execution time). This is encouraging because the main reason for the slowdown in execution time is the higher critical path delay – Vericert-hyperblock performs comparably with Bambu-no-opt in terms of cycle count, with its designs needing only 1.04× the cycles. Cycle count is under the direct control of the scheduler, and therefore shows the effect of the scheduler most clearly, whereas the critical path delay is sensitive to which optimisations the downstream synthesis tool decides to perform. As mentioned previously, estimating the delay of operations and predicting when downstream optimisations will fire is an active research area and is currently implemented conservatively. For example, even Bambu-no-opt failed to predict the delay of the critical path correctly for the fdtd-2d benchmark, and failed timing for it, as did Vericert-list on the cholesky benchmark.

In terms of area, Vericert-hyperblock designs are 1.7× larger than Bambu-default and 1.2× larger than Bambu-no-opt, however, this is tightly linked to the fact that some

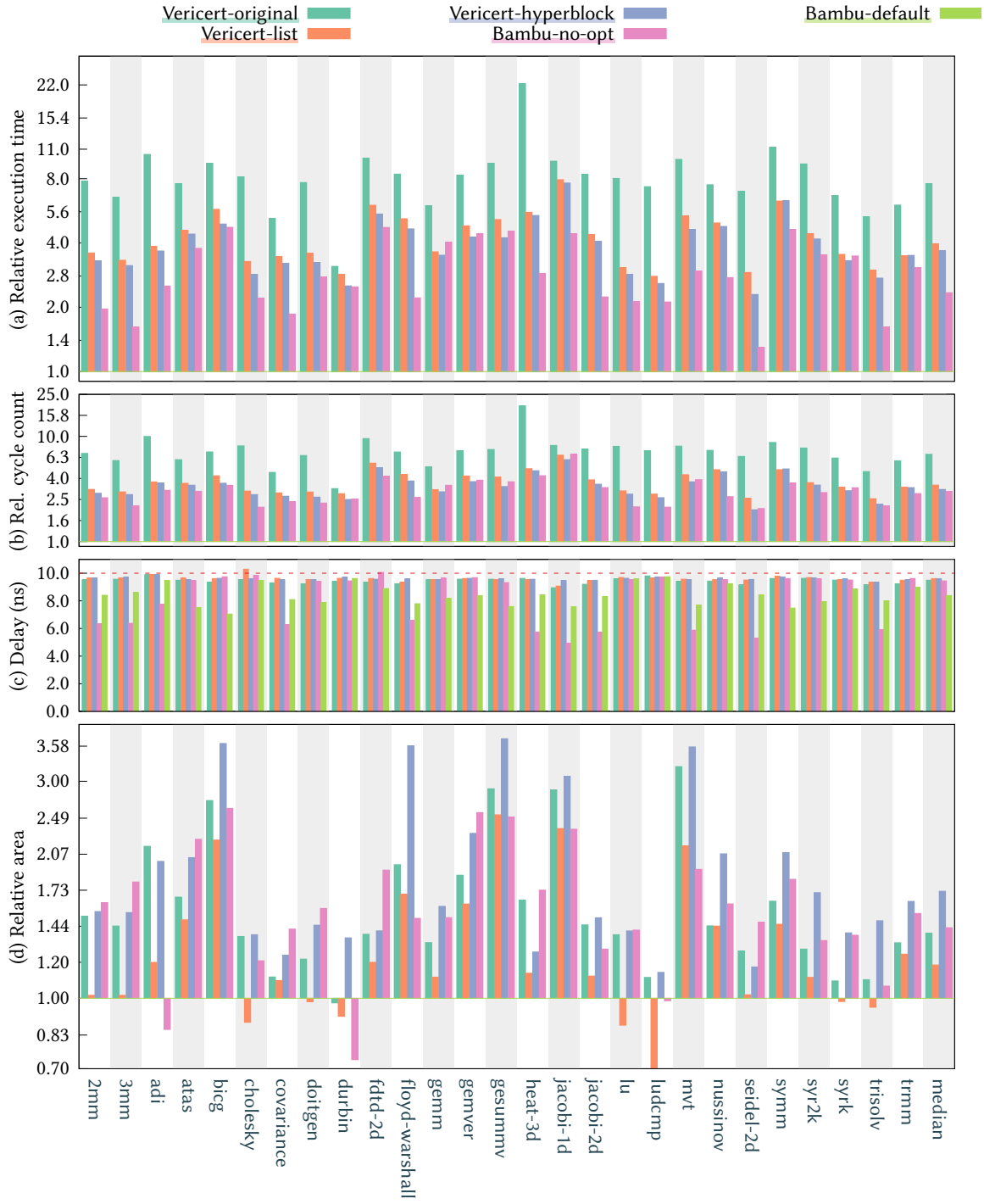


Figure 7.1: Results of simulating and synthesising the PolyBench/C benchmark suite using a range of HLS tool configurations. All measurements are relative to Bambu-default except the greatest critical path delay, which is given absolutely. The dashed red line in that graph corresponds to the target clock with a period of 10ns, or a frequency of 100MHz.

chaining optimisations are currently missed by the downstream synthesis tool. Tweaking the representation of predicated instructions could improve the area significantly.

7.3 RQ2: Area and Delay Improvements of Vericert

To assess whether adding scheduling to Vericert leads to better hardware designs, we compare the hardware produced by Vericert-original with that produced by Vericert-hyperblock. We see that, on average, hyperblock scheduling leads to hardware that requires only 0.48× the time to execute a benchmark (figure 7.1a). This can be attributed to the scheduled hardware requiring only 0.46× the number of cycles (figure 7.1b) and the minimum clock period being nearly identical between the scheduled and unscheduled designs (figure 7.1c). This improvement is unsurprising given that Vericert-original only executes a single instruction per clock cycle. In terms of area (figure 7.1d), hyperblock scheduling leads, on average, to a slight increase in area of 23.6%. This can mostly be attributed to the additional circuitry needed to check the value of predicates and gate instructions. However, in some cases, like floyd-warshall, the area increase is quite egregious, which can be attributed to some back-end optimisations like operation fusing not triggering because the Verilog code does not match a specific pattern the synthesis tool recognises. Tweaking the syntax of the generated code should allow us to produce designs with similar area to those of Vericert-original.

7.4 RQ3: Hyperblock Scheduling Compared to Naïve Scheduling

Hyperblock scheduling is considerably more complicated to implement and verify than list scheduling because it requires if-conversion to combine basic blocks into hyperblocks, as well as predicate-aware scheduling. If we omit if-conversion entirely (hence avoiding predication too), we obtain list scheduling as a special case. Does hyperblock scheduling yield enough of a performance improvement over list scheduling to justify its additional complexity?

To answer this, figure 7.1 measures the hardware produced by Vericert-list. On average, list scheduling leads to hardware requiring 0.52× the total time compared to Vericert-original and 1.08× the total time compared to Vericert-hyperblock. We expect hyperblock scheduling to extend its small lead over list scheduling once the if-conversion heuristics

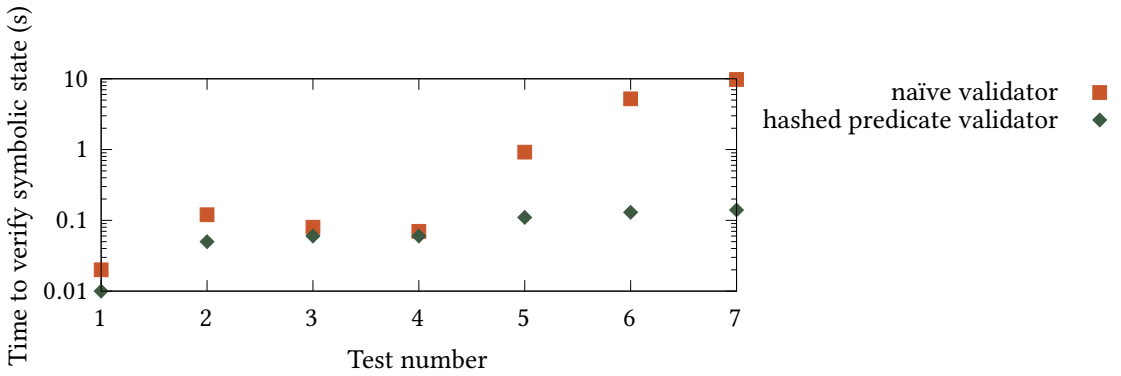


Figure 7.2: Comparing the performance of the naïve hyperblock scheduling validator without hashing, compared to the validator with predicate hashing. The blocks that are tested are manually written and manually scheduled, and are sorted by the time taken by the hashed predicated validator to prove the equivalence.

are improved. In particular, our predictions of predicated instruction latency are currently quite conservative to ensure that timing constraints are met; improving these estimates is an active research area [Rizzi et al. 2023; Tan et al. 2015; Ustun et al. 2020; Wang et al. 2023; Zheng et al. 2014]. On the other hand, the latency of instructions without predicates is much more predictable and its estimation less conservative, leading to better overall performance. We believe list scheduling is at its best, whereas hyperblock scheduling could be greatly improved with better predictions.

In terms of area, we see that Vericert-list leads to the smallest hardware designs, on average 0.69× the size of Vericert-hyperblock designs. This can be attributed to the downstream logic synthesis tool being able to save area by optimising chained operations, such as multiply–accumulate, while not having to handle the predicates that are introduced with Vericert-hyperblock.

7.5 RQ4: Compilation Times of Vericert

To assess whether Vericert-hyperblock has acceptable compilation times, I also compare it against Bambu. Compilation times did not deviate for Bambu, all of them being around 3s mainly due to long startup costs. Vericert-hyperblock compiled each benchmark in 0.9s, also without much variation, showing that verification was not overly costly. As for the question about whether our design decisions led to these compilation times: I remark that if the 'final-state predicates' innovation that we introduced in section 5.5.3 is disabled, that none of the benchmarks compile within a few minutes and eventually the machine runs

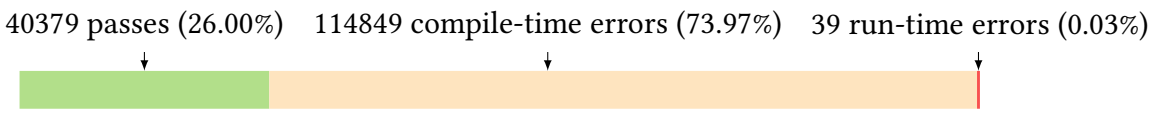


Figure 7.3: Results of fuzzing Vericert using 155267 random C programs generated by Csmith.

out of memory.

To get a better idea of the difference between using the hashed, final-state predicates compared to the more naïve validator, figure 7.2 shows detailed times of how long validation took on some hand-crafted examples, that would not time-out the verification for the naïve validator. The example test programs differ roughly in the number of predicates that are present and how often they are used to justify reorderings of instructions. This shows that using final-state predicates meant that validation time stayed mostly constant, whereas the naïve validator quickly took exponentially more time to validate the same schedule.

7.6 RQ5: Effectiveness of Vericert's Correctness Theorem

> *'Beware of bugs in the above code; I have only proved it correct, not tried it.'*
>
> – D. E. Knuth (1977)

To gain further confidence that the Verilog designs generated by Vericert are actually correct, and that the correctness theorem is indeed effective, I fuzzed Vericert using Csmith [Yang et al. 2011]. Yang et al. previously used Csmith in an extensive fuzzing campaign on CompCert and found a handful of bugs in the unverified parts of that compiler, so it is natural to explore whether it can find bugs in Vericert too. Herklotz et al. [2021a] have recently used Csmith to fuzz other HLS tools including LegUp, so I configured Csmith in a similar way. In addition to the features turned off by Herklotz et al., I turned off the generation of global variables and non-32-bit operations. The generated designs were tested by simulating them and comparing the output value to the results of compiling the test-cases with GCC 10.3.0.

The results of the fuzzing run are shown in Fig. 7.3. Out of 155267 test-cases generated by Csmith, 26% of them passed, meaning they compiled without error and resulted in the same final value as GCC. Most of the test-cases, 73.97%, failed at compile time. The most common reasons for this were unsigned comparisons between integers (Vericert

requires them to be signed), and the presence of 8-bit operations (which Vericert does not support, and which I could not turn off due to a limitation in Csmith). Because the test-cases generated by Csmith could not be tailored exactly to the C fragment that Vericert supports, such a high compile-time failure rate is expected. Finally, and most interestingly, there were a total of 39 run-time failures, which the correctness theorem should be proving impossible. However, all 39 of these failures are due to a bug in the pretty-printing of the final Verilog code, where a logical negation (!) was accidentally used instead of a bitwise negation (~). Once this bug was fixed, all test-cases passed.

7.7 Summary

Vericert with hyperblock scheduling seems to generate hardware around the same cycle count as the hardware generated by Bambu HLS, which is encouraging. However, due to if-conversion and scheduling sometimes mispredicting the latency of certain operations, especially predicated operations, the final execution time of Vericert is 1.57× that of Bambu HLS. With better latency estimation, I am confident that this could get closer to unoptimised Bambu HLS. However, to get closer to optimised Bambu HLS, more optimisations will need to be implemented. One important optimisation that is missing from Vericert is loop scheduling, which would close the gap to Bambu HLS, especially on a loop-centric benchmark such as PolyBench/C.

8 Conclusion

8.1 Coq mechanisation

The lines of code for the implementation and proof of Vericert can be found in table 8.1. Overall, it took about 3 person-years to build Vericert – about 6 person-months on implementation and 30 person-months on proofs. The largest proofs were by far the scheduling proof and the three-valued logic validator. The scheduling proof was difficult, and different attempts with different proof styles were needed to complete it. The main difficulty was minimising the need for the three-valued validator, as initially it appeared to be unnecessary. In the end, it was needed to prove the equivalence of predicates in the absence of structural equality. This led to the proof of the three-valued logic validator. Even though it looks like a very large proof, it was mainly quite straight-forward and purely technical, taking roughly 1.5 months to complete. The second hardest proof was the correctness proof for the HTL generation, which required equivalence proofs between all integer operations supported by CompCert and those supported in hardware. A large percentage of the proof is dedicated to the load and store instruction translation. These were tedious to prove correct because of the substantial difference between the memory models used, and the need to prove properties such as stores outside of the allocated memory being undefined, so that a finite array could be used. In addition to that, since pointers in HTL and Verilog are represented as integers, instead of as a separate 'pointer' type like in the CompCert semantics, one had to show that the integer arithmetic was correct with respect to the pointer arithmetic in CompCert. Many new theorems had to be proven about them in Vericert to prove the conversion from pointer to integer. Moreover, another large proof in the back end describing the correct BRAM generation includes many proofs about the extensional equality of array operations, such as merging arrays with different assignments. Due to the negative edge implying that two merges take place every clock cycle, it can become tedious to handle merges of changes performed during each of the clock edges, as the extensional equality over merges needs to be specified for each possible place the

Table 8.1: Statistics about the numbers of lines of code in the proof and implementation of Vericert, counted using `coqwc`.

	OCaml	Spec	Proofs	Total
Data structures and libraries	—	1099	771	1870
Integers and values	—	393	520	913
RTLBlock and RTLPar semantics	—	748	286	1034
Hyperblock generation	—	1716	1820	3536
Hyperblock scheduling	1083	3939	5597	10619
three-valued logic validator	1276	4054	5040	10370
HTL semantics	—	249	31	280
HTL generation	—	2248	2996	5244
BRAM generation	—	1867	2890	4757
forward substitution	—	481	425	906
Verilog semantics	—	628	124	752
Verilog generation	—	261	283	544
Top-level driver, pretty printers	1404	273	223	1900
Total	3763	17956	21006	42725

register assignment could have taken place.

8.2 Limitations and Future Work

There are various limitations in Vericert compared to other HLS tools due to the fact that our main focus was on formally verifying the translation from RTL to Verilog. In this section, we outline the current limitations of our tool and suggest how they can be overcome in future work, first describing limitations to the generated hardware, and then describing the limitations on the software input that Vericert accepts.

8.2.1 Limitations to the generated hardware

Lack of support for external IP core

Intellectual property cores (IP cores) and other external hardware cores are often used by HLS tools to implement specific operations efficiently. One example would be the implementation of a division core, which can be designed as a fully pipelined hardware design, executing different stages of an operation in parallel. This is in contrast to using the default division operation that generates the division circuit in a single clock cycle.

This means that the otherwise long-running division operation can be performed over multiple, shorter cycles, leaving the overall maximum frequency of the design unchanged. In HTL, IP cores could be represented in a similar fashion to load and store instructions, by using wires to communicate with an abstract computation block modelled in HTL and could later be replaced by a hardware implementation.

Furthermore, support for IP cores with a specific interface, such as a simple ready/finished interface, could have a general interface specification, so that hardware that follows this pattern could be directly integrated into Vericert. Integrating a new core would only require a proof of correctness of the specification and the Verilog implementation, as well as an equivalence with the operation that it should be replacing. This would allow for a more general implementation of the memory interface, for example, making it possible to implement both loads and stores in the usual positive edge of the clock, as well as making it possible to pipeline these loads and stores.

Lack of Loop Pipelining

Another critical HLS optimisation that is often integrated with the instruction scheduling optimisation is loop pipelining, also known as modulo scheduling. This is an important optimisation for HLS, making it possible to execute parts of different loop iterations in parallel. The ideal scenario is that the whole function can eventually be pipelined so that it can accept a stream of inputs every clock cycle. This type of hardware design is currently not possible with Vericert. However, there is work on proving software loop pipelining correct in CompCert [Tristan and Leroy 2010], which could be adapted and extended to support generating hardware pipelines, by using predicated execution [Rau et al. 1992b], which are already supported, and a rotating register file [Rau et al. 1992a] to remove the need for a prologue and epilogue in the software pipelined loop.

Limitations with I/O

Vericert is currently limited in terms of I/O because the main function cannot accept any arguments if the CLIGHT program is to be well-formed.[1] Moreover, external function calls that produce traces have not been implemented yet, but these could enable the C program to read and write values on a bus that is shared with various other components in the

[1] Technically, Vericert (and indeed, CompCert) can compile main functions that have arbitrary arguments and will handle those inputs appropriately, but the correctness theorem offers no guarantees about such programs.

hardware design.

8.2.2 Limitations on the software input

Lack of support for global variables

In CompCert, each global variable is stored in its own memory. A generalisation of our translation of the stack frame into a BRAM block could therefore translate global variables in the same manner. This would require a generalisation of pointers so that they store provenance information to ensure that each pointer accesses the right BRAM. It would also be necessary to generalise the BRAM interface so that it decodes the provenance information and indexes the correct array.

Other language restrictions

C and Verilog handle signedness quite differently. By default, all operators and registers in Verilog (and HTL) are unsigned, so to force an operation to handle the bits as signed, both operators have to be forced to be signed. Moreover, Verilog implicitly resizes expressions to the largest needed size by default, which can affect the result of the computation. This feature is not supported by the Verilog semantics we adopted, so to match the semantics to the behaviour of the simulator and synthesis tool, braces are placed around all expressions to inhibit implicit resizing. Instead, explicit resizing is used in the semantics, and operations can only be performed on two registers that have the same size.

Furthermore, equality checks between *unsigned* variables are actually not supported, because this requires supporting the comparison of pointers, which should only be performed between pointers with the same provenance. In Vericert there is currently no way to determine the provenance of a pointer, and it therefore cannot model the semantics of unsigned comparison in CompCert. This is not a severe restriction in practice however, because in the absence of dynamic allocation, equality comparison of pointers is rarely needed, and equality comparison of integers can still be performed by casting them both to signed integers.

Finally, the `mulhs` and `mulhu` instructions, which fetch the upper bits of a 32-bit multiplication, are not translated by Vericert, because 64-bit numbers are not supported. These instructions are only generated to optimise divisions by a constant that is not a power of two, so turning off constant propagation will allow these programs to pass without error.

8.2.3 The Future of Vericert

Designing Correct Hardware

It would be interesting to use the Vericert correctness theorem with the Verified Software Toolchain [Appel 2011] to prove that the hardware implements the same specification as the software. Specifications would become more interesting with more support for I/O, however, it could already be used to prove the correctness of the output of a Vericert design.

In addition to that, it would be interesting to use a verified synthesis tool to build a complete workflow from software to hardware. Lööw [2021] already showed that a Vericert design with registers with a bit-width could be synthesised into LUTs using Lutsig. One could then imagine trying to prove that a specification that is proven to hold about the C code would also hold about the netlist. The main issue is that this proof can not be performed in one theorem prover, as Lutsig is proven correct in HOL4.

Interaction With Software and Hardware

Another direction that could be explored is hardware/software co-design or the interaction of HLS designs with externally defined hardware in a more modular fashion. CompCert-O [Koenig and Shao 2021] makes it possible to reason about the interaction of separate programs with different semantics through the C calling convention of CompCert. It would be interesting to extend the Vericert Verilog semantics with a calling convention that could be used to interact and reason about the hardware at this level. In this way, one could either verify heterogeneous systems, or verify and reason about the interaction of CompCert C designs destined to be synthesised into hardware with externally verified hardware designs.

8.3 Summary

In conclusion, the need for a correct high-level synthesis led to the design of Vericert, a verified HLS tool based on CompCert. We showed that with the implementation of hyperblock scheduling, the performance of Vericert was around 1.6× performance of the unoptimised version of Bambu HLS, which is promising, while being around 3.6× slower than optimised Bambu HLS. We hope that Vericert will enable HLS to be used in spaces where the correctness of the hardware is paramount, for example for compiling

cryptographic algorithms from C, where performance might only be a secondary concern. This should still allow for efficient application-specific accelerators, and together with a proof of correctness between the netlist and the hardware design, a near end-to-end correctness theorem for the accelerator.

Bibliography

[SW] AbsInt, *CompCert release 19.10* 2019. URL: https://www.absint.com/releasenotes/compcert/19.10/ (cit. on pp. 90, 91).

Frances E. Allen. 1970. 'Control Flow Analysis'. In: *Proceedings of a Symposium on Compiler Optimization*. Association for Computing Machinery, Urbana-Champaign, Illinois, 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479 (cit. on p. 34).

J. R. Allen, Ken Kennedy, Carrie Porterfield and Joe Warren. 1983. 'Conversion of Control Dependence to Data Dependence'. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '83). Association for Computing Machinery, Austin, Texas, 177–189. ISBN: 0897910907. DOI: 10.1145/567067.567085 (cit. on p. 91).

AMD. 2023a. *Vitis Forums*. Relevant quote from AMD: "If-Conversion aims to convert a sequence of blocks into a single block for better optimization result." (2023). Retrieved 2nd June 2023 from https://bit.ly/vitisifc (cit. on p. 89).

AMD. 2023b. *Vitis High-level Synthesis*. (2023). Retrieved 21st May 2023 from https://bit.ly/41R0204 (cit. on pp. 19, 32, 39, 45, 47, 65).

Damian P. Anderson and John Ainscough. May 1994. 'The Verification of Scheduling Algorithms'. In: *IEE Colloquium on Structured Methods for Hardware Systems Design*. (May 1994), 7/1–7/5 (cit. on p. 50).

Andrew W. Appel. 2011. 'Verified Software Toolchain'. In: *Programming Languages and Systems*. Ed. by Gilles Barthe. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. ISBN: 978-3-642-19718-5. DOI: 10.1007/978-3-642-19718-5_1 (cit. on p. 159).

Apple. June 2022. *Deploying Transformers on the Apple Neural Engine*. (June 2022). Retrieved 13th Jan. 2024 from https://machinelearning.apple.com/research/neural-engine-transformers (cit. on p. 19).

Arm. 11th Nov. 2011. *ARMv8 Instruction Set Overview*. Tech. rep. PRD03-GENC-010197 15.0. (11th Nov. 2011) (cit. on p. 38).

Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry and Benjamin Werner. 2011. 'A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses'. In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Springer Berlin Heidelberg, Berlin, Heidelberg, 135–150. ISBN: 978-3-642-25379-9. DOI: 10.1007/978-3-642-25379-9_12 (cit. on pp. 44, 118).

Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul and Robin Watts. 1996. 'Handel-C Language Reference Guide'. *Computing Laboratory. Oxford University, UK* (cit. on p. 32).

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek and Krste Asanović. 2012. 'Chisel: Constructing hardware in a Scala embedded language'. In: *DAC Design Automation Conference 2012*. IEEE, 1212–1221. DOI: 10.1145/2228360.2228584 (cit. on p. 66).

Kenneth R. Baker. 2019. *Principles of sequencing and scheduling*. eng. (Second edition. ed.). Wiley series in operations research and management science. Wiley, Hoboken, NJ. ISBN: 1-119-26259-3 (cit. on pp. 40, 89).

Thomas Ball and James R. Larus. 1993. 'Branch Prediction for Free'. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (PLDI '93). Association for Computing Machinery, Albuquerque, New Mexico, USA, 300–313. ISBN: 0897915984. DOI: 10.1145/155090.155119 (cit. on p. 98).

Kunal Banerjee, Chandan Karfa, Dipankar Sarkar and Chittaranjan Mandal. Aug. 2014. 'Verification of Code Motion Techniques Using Value Propagation'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33, 8, (Aug. 2014), 1180–1193. DOI: 10.1109/TCAD.2014.2314392 (cit. on pp. 21, 45, 48).

Haniel Barbosa, Clark Barrett, Martin Brain et al.. 2022. 'cvc5: A Versatile and Industrial-Strength SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Springer International Publishing, Cham, 415–442. ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24 (cit. on pp. 43, 44).

Clark Barrett, Pascal Fontaine and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa. www.SMT-LIB.org (cit. on p. 43).

Gilles Barthe, Delphine Demange and David Pichardie. Mar. 2014. 'Formal Verification of an SSA-Based Middle-End for CompCert'. *ACM Trans. Program. Lang. Syst.*, 36, 1, (Mar. 2014). DOI: 10.1145/2579080 (cit. on p. 99).

[SW] Michel Berkelaar, *lp_solve v5.5* 2010. URL: https://lpsolve.sourceforge.net/5.5/ (cit. on p. 99).

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg. DOI: 10.1007/978-3-662-07964-5 (cit. on pp. 44, 52).

Frédéric Besson, Sandrine Blazy and Pierre Wilke. Nov. 2018. 'CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics'. *Journal of Automated Reasoning*, 63, 2, (Nov. 2018), 369–392. DOI: 10.1007/s10817-018-9496-y (cit. on p. 86).

Sandrine Blazy and Xavier Leroy. 2005. 'Formal Verification of a Memory Model for C-Like Imperative Languages'. In: *Formal Methods and Software Engineering*. Ed. by Kung-Kiu Lau and Richard Banach. Springer, Berlin, Heidelberg, 280–299. ISBN: 978-3-540-32250-4. DOI: 10.1007/11576280_20 (cit. on p. 127).

Ludwik Borowski. 1970. *Selected Works of J. Łukasiewicz*. Nort Holland (cit. on p. 118).

Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala and Arvind. 2020. 'The Essence of Bluespec: A Core Language for Rule-Based Hardware Design'. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). Association for Computing Machinery, London, UK, 243–257. ISBN: 9781450376136. DOI: 10.1145/3385412.3385965 (cit. on pp. 47, 51, 66).

Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe and Pascal Fontaine. 2009. 'veriT: An Open, Trustable and Efficient SMT-Solver'. In: *Automated Deduction – CADE-22*. Ed. by Renate A. Schmidt. Springer Berlin Heidelberg, Berlin, Heidelberg, 151–156. ISBN: 978-3-642-02959-2. DOI: 10.1007/978-3-642-02959-2_12 (cit. on pp. 43, 44, 120).

Andrew Boutros and Vaughn Betz. 2021. 'FPGA Architecture: Principles and Progression'. *IEEE Circuits and Systems Magazine*, 21, 2, 4–29. DOI: 10.1109/MCAS.2021.3071607 (cit. on p. 27).

Lucy Bowen and Chris Lupo. 2020. 'The Performance Cost of Software-Based Security Mitigations'. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (ICPE '20). Association for Computing Machinery, Edmonton AB, Canada, 210–217. ISBN: 9781450369916. DOI: 10.1145/3358960.3379139 (cit. on p. 20).

Matthew Bowen. 1998. 'Handel-C Language Reference Manual'. *Embedded Solutions Ltd*, 2 (cit. on p. 32).

Thomas Braibant and Adam Chlipala. 2013. 'Formal Verification of Hardware Synthesis'. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Springer

Berlin Heidelberg, Berlin, Heidelberg, 213–228. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_14 (cit. on pp. 51, 66).

Robert Brummayer and Armin Biere. 2009. 'Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Stefan Kowalewski and Anna Philippou. Springer Berlin Heidelberg, Berlin, Heidelberg, 174–177. ISBN: 978-3-642-00768-2. DOI: 10.1007/978-3-642-00768-2_16 (cit. on p. 43).

Mihai Budiu and Seth Copen Goldstein. 2002. 'Compiling Application-Specific Hardware'. In: *Field-Programmable Logic and Applications, Reconfigurable Computing Is Going Mainstream, 12th International Conference, FPL 2002, Montpellier, France, September 2-4, 2002, Proceedings* (Lecture Notes in Computer Science). Ed. by Manfred Glesner, Peter Zipf and Michel Renovell. Vol. 2438. Springer, 853–863. DOI: 10.1007/3-540-46117-5_88 (cit. on pp. 38, 89).

[SW] Cadence, *Conformal Equivalence Checker* 2023. URL: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/logic-equivalence-checking/conformal-equivalence-checker.htmlRetrieved 20th Dec. 2023 from (cit. on p. 43).

[SW] Cadence, *Jasper C2RTL* 2023. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-c-formal-verification.htmlRetrieved 20th Dec. 2023 from (cit. on p. 48).

Timothy J. Callahan and John Wawrzynek. 1998. 'Instruction-Level Parallelism for Reconfigurable Computing'. In: *Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm, 8th International Workshop, FPL'98, Tallinn, Estonia, August 31 - September 3, 1998, Proceedings* (Lecture Notes in Computer Science). Ed. by Reiner W. Hartenstein and Andres Keevallik. Vol. 1482. Springer, 248–257. DOI: 10.1007/BFb0055252 (cit. on pp. 38, 89).

Philip L. Campbell, Ksheerabdhi Krishna and Robert A. Ballance. Feb. 1993. *Refining and defining the program dependence web*. Tech. rep. 93-6. The University of New Mexico, Albuquerque, NM 87131, (Feb. 1993), 1–33 (cit. on p. 35).

Andrew Canis. 2015. 'Legup: open-source high-level synthesis research framework'. PhD thesis (cit. on pp. 63, 89).

Andrew Canis, Stephen D. Brown and Jason H. Anderson. Sept. 2014. 'Modulo SDC scheduling with recurrence minimization in high-level synthesis'. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. (Sept. 2014), 1–8. DOI: 10.1109/FPL.2014.6927490 (cit. on p. 41).

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown and Tomasz Czajkowski. 2011. 'LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems'. In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (FPGA '11). Association for Computing Machinery, Monterey, CA, USA, 33–36. ISBN: 9781450305549. DOI: 10.1145/1950413.1950423 (cit. on p. 147).

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown and Jason H. Anderson. Sept. 2013. 'Legup: an Open-Source High-Level Synthesis Tool for Fpga-Based Processor/accelerator Systems'. *ACM Trans. Embed. Comput. Syst.*, 13, 2, (Sept. 2013). DOI: 10.1145/2514740 (cit. on pp. 19, 30, 32, 39, 45, 47, 65, 68).

Luca P. Carloni, Kenneth L. McMillan and Alberto L. Sangiovanni-Vincentelli. Sept. 2001. 'Theory of Latency-Insensitive Design'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20, 9, (Sept. 2001), 1059–1076. DOI: 10.1109/43.945302 (cit. on p. 41).

Pohua P. Chang, Scott A. Mahlke and Wen-mei W. Hwu. 1991. 'Using Profile Information to Assist Classic Code Optimizations'. *Softw. Pract. Exp.*, 21, 12, 1301–1321. DOI: 10.1002/spe.4380211204 (cit. on p. 98).

Richard Chapman, Geoffrey Brown and Miriam Leeser. Mar. 1992. 'Verified high-level synthesis in BEDROC'. In: *[1992] Proceedings The European Conference on Design Automation.* IEEE Computer Society, (Mar. 1992), 59–63. DOI: 10.1109/EDAC.1992.205894 (cit. on pp. 45, 47, 49).

Pankaj Chauhan. 2020. *Formally Ensuring Equivalence between C++ and RTL designs.* SLEC. (2020). https://bit.ly/2KbT0ki (cit. on pp. 21, 48).

Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide and John Regehr. 2013. 'Taming Compiler Fuzzers'. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 197–208. DOI: 10.1145/2491956.2462173 (cit. on p. 63).

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala and Arvind. Aug. 2017. 'Kami: a Platform for High-Level Parametric Hardware Specification and Its Modular Verification'. *Proc. ACM Program. Lang.*, 1, ICFP, (Aug. 2017), 24:1–24:30. DOI: 10.1145/3110268 (cit. on p. 51).

Young-kyu Choi and Jason Cong. 2018. 'HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds'. In: *2018 IEEE/ACM International*

Conference on Computer-Aided Design (ICCAD), 1–8. DOI: 10.1145/3240765.3240815 (cit. on p. 148).

Ramanuj Chouksey and Chandan Karfa. 2020. 'Verification of Scheduling of Conditional Behaviors in High-Level Synthesis'. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1–14. DOI: 10.1109/TVLSI.2020.2978242 (cit. on pp. 21, 45, 48).

Ramanuj Chouksey, Chandan Karfa and Purandar Bhaduri. July 2019. 'Translation Validation of Code Motion Transformations Involving Loops'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38, 7, (July 2019), 1378–1382. DOI: 10.1109/TCAD.2018.2846654 (cit. on pp. 21, 45, 48).

Edmund Clarke, Daniel Kroening and Karen Yorav. June 2003. 'Behavioral consistency of C and Verilog programs using bounded model checking'. In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. (June 2003), 368–371. DOI: 10.1145/775832.775928 (cit. on p. 47).

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers and Zhiru Zhang. Apr. 2011. 'High-Level Synthesis for Fpgas: From Prototyping To Deployment'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30, 4, (Apr. 2011), 473–491. DOI: 10.1109/TCAD.2011.2110592 (cit. on p. 68).

Jason Cong and Zhiru Zhang. July 2006. 'An efficient and versatile scheduling algorithm based on SDC formulation'. In: *2006 43rd ACM/IEEE Design Automation Conference*. (July 2006), 433–438. DOI: 10.1145/1146909.1147025 (cit. on pp. 22, 40, 68, 99).

George A. Constantinides, Peter Y.K. Cheung and Wayne Luk. Mar. 2001. 'The Multiple Wordlength Paradigm'. In: *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. (Mar. 2001), 51–60 (cit. on p. 30).

Philippe Coussy, Daniel D. Gajski, Michael Meredith and Andres Takach. July 2009. 'An Introduction To High-Level Synthesis'. *IEEE Design Test of Computers*, 26, 4, (July 2009), 8–17. DOI: 10.1109/MDT.2009.69 (cit. on p. 30).

Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi and Jeyavijayan Rajendran. Aug. 2019. 'HardFails: Insights into Software-Exploitable Hardware Bugs'. In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, (Aug. 2019), 213–230. ISBN: 978-1-939133-06-9. https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky (cit. on p. 20).

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds and Clark Barrett. 2017. 'SMTCoq: A Plug-In for Integrating SMT Solvers into Coq'.

In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Springer International Publishing, Cham, 126–133. ISBN: 978-3-319-63390-9. DOI: 10.1007/978-3-319-63390-9_7 (cit. on p. 118).

John R. Ellis. 1985. 'Bulldog: A compiler for VLIW architectures'. PhD thesis. Yale University (cit. on p. 89).

Martin Ellis. 2008. 'Correct Synthesis and Integration of Compiler-Generated Function Units'. PhD thesis. Newcastle University. https://theses.ncl.ac.uk/jspui/handle/10443/828 (cit. on pp. 46, 47, 51).

Karine Even-Mendoza, Cristian Cadar and Alastair Donaldson. Jan. 2021. 'Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative About Undefined Behaviour'. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (ASE '20). Association for Computing Machinery, Virtual Event, Australia, (Jan. 2021), 1219–1223. ISBN: 9781450367684. DOI: 10.1145/3324884.3418933 (cit. on p. 65).

Paolo Faraboschi, Joseph A. Fisher and Cliff Young. Nov. 2001. 'Instruction Scheduling for Instruction Level Parallel Processors'. *Proceedings of the IEEE*, 89, 11, (Nov. 2001), 1638–1659. DOI: 10.1109/5.964443 (cit. on p. 36).

Fabrizio Ferrandi. 2014. *PandA-Bambu release notes*. (2014). Retrieved 16th Nov. 2023 from https://github.com/ferrandi/PandA-bambu/blob/c443bf14c33a9a74008ada12f56e7a62e30e5efe/NEWS#L304 (cit. on p. 89).

Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato and Antonino Tumeo. Dec. 2021. 'Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications'. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, (Dec. 2021), 1327–1330. DOI: 10.1109/DAC18074.2021.9586110 (cit. on p. 147).

Joseph A. Fisher. 1981. 'Trace Scheduling: A Technique for Global Microcode Compaction'. *IEEE Transactions on Computers*, C-30, 7, 478–490. DOI: 10.1109/TC.1981.1675827 (cit. on p. 89).

Dan Gajski, Todd Austin and Steve Svoboda. 2010. 'What input-language is the best choice for high level synthesis (HLS)?' In: *Design Automation Conference*, 857–858. DOI: 10.1145/1837274.1837489 (cit. on p. 65).

Stephane Gauthier and Zubair Wadood. 2020. *High-Level Synthesis: Can it outperform hand-coded HDL?* White paper. (2020). https://info.silexica.com/high-level-synthesis/1 (cit. on p. 19).

Georges Gonthier. 2008. 'Formal Proof–the Four-Color Theorem'. *Notices of the AMS*, 55, 11, 1382–1393 (cit. on p. 44).

[SW] Google, *Google XLS* 2024. URL: https://google.github.io/xls/Retrieved 21st Mar. 2024 from (cit. on pp. 30, 32).

Google. 2023. *XLS: Accelerated HW Synthesis*. The XLS scheduler refers to using an SMT solver to merge mutually exclusive nodes. (2023). Retrieved 14th Nov. 2023 from https://github.com/google/xls/blob/dde7095ff1050b09c37cb44d1977bff1af8de050/xls/scheduling/mutual_exclusion_pass.h#L112 (cit. on p. 89).

Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux and Alexandre Bérard. Oct. 2023. 'Formally Verifying Optimizations with Block Simulations'. *Proc. ACM Program. Lang.*, 7, OOPSLA, Article 224, (Oct. 2023), 30 pages. DOI: 10.1145/3622799 (cit. on p. 61).

David J. Greaves. 2019. *Research Note: An Open Source Bluespec Compiler*. (2019). arXiv: 1905.03746 [cs.PL] (cit. on p. 66).

David J. Greaves and Satnam Singh. 2008. 'Kiwi: Synthesis of FPGA Circuits from Parallel Programs'. In: *FCCM*. IEEE Computer Society, 3–12. DOI: 10.1109/FCCM.2008.46 (cit. on p. 66).

Monika Gupta. 4th Oct. 2023. *Google Tensor G3: The new chip that gives your Pixel an AI upgrade*. Google. (4th Oct. 2023). Retrieved 13th Jan. 2024 from https://blog.google/products/pixel/google-tensor-g3-pixel-8/ (cit. on p. 19).

Sumit Gupta, Nikil Dutt, Rajesh Gupta and Alex Nicolau. Jan. 2003. 'SPARK: a high-level synthesis framework for applying parallelizing compiler transformations'. In: *16th International Conference on VLSI Design, 2003. Proceedings.* (Jan. 2003), 461–466. DOI: 10.1109/ICVD.2003.1183177 (cit. on p. 48).

Nicholas Halbwachs, Paul Caspi, Pascal Raymond and Daniel Pilaud. 1991. 'The Synchronous Data Flow Programming Language LUSTRE'. *Proceedings of the IEEE*, 79, 9, 1305–1320. DOI: 10.1109/5.97300 (cit. on p. 35).

Paul Havlak. 1994. 'Construction of thinned gated single-assignment form'. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee, David Gelernter, Alex Nicolau and David Padua. Springer Berlin Heidelberg, Berlin, Heidelberg, 477–499. ISBN: 978-3-540-48308-3. DOI: 10.1007/3-540-57659-2_28 (cit. on p. 35).

Yann Herklotz, Delphine Demange and Sandrine Blazy. 2023. 'Mechanised Semantics for Gated Static Single Assignment'. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2023). Association for Computing

Machinery, Boston, MA, USA, 182–196. ISBN: 9798400700262. DOI: 10.1145/3573105.3575681 (cit. on p. 120).

Yann Herklotz, Zewei Du, Nadesh Ramanathan and John Wickerson. 2021a. 'An Empirical Study of the Reliability of High-Level Synthesis Tools'. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 219–223. DOI: 10.1109/FCCM51124.2021.00034 (cit. on pp. 21, 63, 153).

Yann Herklotz, James D. Pollard, Nadesh Ramanathan and John Wickerson. Oct. 2021b. 'Formal Verification of High-Level Synthesis'. *Proceedings of the ACM on Programming Languages*, 5, OOPSLA, (Oct. 2021). DOI: 10.1145/3485494 (cit. on p. 63).

[SW] Yann Herklotz, James D. Pollard, Nadesh Ramanathan and John Wickerson, *Vericert v2.0.0-rc2* version v2.0.0-rc2, Jan. 2024. DOI: 10.5281/zenodo.10808233 (cit. on p. 23).

Yann Herklotz and John Wickerson. 2020. 'Finding and Understanding Bugs in FPGA Synthesis Tools'. In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA '20). Association for Computing Machinery, Seaside, CA, USA, 277–287. ISBN: 9781450370998. DOI: 10.1145/3373087.3375310.

Yann Herklotz and John Wickerson. June 2024. 'Hyperblock Scheduling for Verified High-Level Synthesis'. *Proceedings of the ACM on Programming Languages*, 8, PLDI, Article 225, (June 2024), 25 pages. DOI: 10.1145/3656455 (cit. on p. 89).

Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat and Timo Hönig. 2021. 'The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level'. In: *Proceedings of the 14th European Workshop on Systems Security* (EuroSec '21). Association for Computing Machinery, Online, United Kingdom, 8–14. ISBN: 9781450383370. DOI: 10.1145/3447852.3458721 (cit. on p. 20).

C.A.R. Hoare. Aug. 1978. 'Communicating Sequential Processes'. *Commun. ACM*, 21, 8, (Aug. 1978), 666–677. DOI: 10.1145/359576.359585 (cit. on p. 30).

Ekawat Homsirikamol and Kris Gaj. 2014. 'Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study'. In: *ReConFig*. IEEE, 1–8. DOI: 10.1109/ReConFig.2014.7032504 (cit. on p. 19).

Gregory Littell Hopwood. 1978. 'Decompilation'. PhD thesis (cit. on p. 69). AAI7811860.

Enoch Hwang, Frank Vahid and Yu-Chin Hsu. 1999. 'FSMD Functional Partitioning for Low Power'. In: *Proceedings of the conference on Design, automation and test in Europe*, 7–es. DOI: 10.1109/DATE.1999.761092 (cit. on pp. 48, 72).

Ed. by B. R. Rau and J. A. Fisher. 'The Superblock: An Effective Technique for VLIW and Superscalar Compilation'. *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*. Springer US, Boston, MA, 229–248. ISBN: 978-1-4615-3200-2. DOI: 10.1007/978-1-4615-3200-2_7 (cit. on pp. 37, 61, 89).

IEEE. 2024. 'IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language', 1–1354. DOI: 10.1109/IEEESTD.2024.10458102 (cit. on p. 69).

IEEE. Apr. 2006. 'IEEE Standard for Verilog Hardware Description Language'. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, (Apr. 2006), 1–590. DOI: 10.1109/IEEESTD.2006.99495 (cit. on pp. 29, 79).

IEEE. 2005. 'IEEE Standard for Verilog Register Transfer Level Synthesis'. *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1*, 1–116. DOI: 10.1109/IEEESTD.2005.339572 (cit. on p. 79).

Intel. 2020a. *High-level Synthesis Compiler*. (2020). Retrieved 18th Nov. 2020 from https://intel.ly/2UDiWr5 (cit. on pp. 19, 65).

Intel. 2020b. *SDK for OpenCL Applications*. (2020). Retrieved 20th July 2020 from https://intel.ly/30sYHz0 (cit. on pp. 30, 32, 39, 45, 47).

James A. Jablin, Thomas B. Jablin, Onur Mutlu and Maurice Herlihy. 2014. 'Warp-Aware Trace Scheduling for GPUs'. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (PACT '14). Association for Computing Machinery, Edmonton, AB, Canada, 163–174. ISBN: 9781450328098. DOI: 10.1145/2628071.2628101 (cit. on p. 39).

He Jifeng, Ian Page and Jonathan Bowen. 1993. 'Towards a provably correct hardware implementation of occam'. In: *Correct Hardware Design and Verification Methods*. Ed. by George J. Milne and Laurence Pierre. Springer Berlin Heidelberg, Berlin, Heidelberg, 214–225. ISBN: 978-3-540-70655-7. DOI: 10.1007/BFb0021726 (cit. on pp. 49, 50).

Lana Josipović, Philip Brisk and Paolo Ienne. Sept. 2017. 'An Out-of-Order Load-Store Queue for Spatial Computing'. *ACM Trans. Embed. Comput. Syst.*, 16, 5s, Article 125, (Sept. 2017), 19 pages. DOI: 10.1145/3126525 (cit. on p. 42).

Lana Josipović, Radhika Ghosal and Paolo Ienne. 2018. 'Dynamically Scheduled High-level Synthesis'. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA '18). ACM, Monterey, CALIFORNIA, USA, 127–136. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174264 (cit. on pp. 41, 42).

Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne and Jordi Cortadella. Nov. 2021. 'Buffer Placement and Sizing for High-Performance Dataflow Circuits'. *ACM Trans. Reconfigurable Technol. Syst.*, 15, 1, Article 4, (Nov. 2021), 32 pages. DOI: 10.1145/3477053 (cit. on p. 42).

Jacques-Henri Jourdan, François Pottier and Xavier Leroy. 2012. 'Validating LR(1) Parsers'. In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416. ISBN: 978-3-642-28869-2. DOI: 10.1007/978-3-642-28869-2_20 (cit. on pp. 66, 77).

John B. Kam and Jeffrey D. Ullman. Jan. 1976. 'Global Data Flow Analysis and Iterative Algorithms'. *J. ACM*, 23, 1, (Jan. 1976), 158–171. DOI: 10.1145/321921.321938 (cit. on p. 35).

Chandan Karfa, Chittaranjan Mandal and Dipankar Sarkar. July 2012. 'Formal Verification of Code Motion Techniques Using Data-Flow-Driven Equivalence Checking'. *ACM Trans. Des. Autom. Electron. Syst.*, 17, 3, (July 2012). DOI: 10.1145/2209291.2209303 (cit. on p. 48).

Chandan Karfa, Chittaranjan Mandal, Dipankar Sarkar, S. R. Pentakota and Chris Reade. 2006. 'A Formal Verification Method of Scheduling in High-level Synthesis'. In: *Proceedings of the 7th International Symposium on Quality Electronic Design* (ISQED '06). IEEE Computer Society, Washington, DC, USA, 71–78. ISBN: 0-7695-2523-7. DOI: 10.1109/ISQED.2006.10 (cit. on pp. 21, 45, 48).

Chandan Karfa, Dipankar Sarkar and Chittaranjan Mandal. Mar. 2010. 'Verification of Datapath and Controller Generation Phase in High-Level Synthesis of Digital Circuits'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29, 3, (Mar. 2010), 479–492. DOI: 10.1109/TCAD.2009.2035542 (cit. on p. 48).

Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal and Pramod Kumar. Mar. 2008. 'An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27, 3, (Mar. 2008), 556–569. DOI: 10.1109/TCAD.2007.913390 (cit. on p. 48).

Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal and Chris Reade. 2007. 'Hand-in-hand Verification of High-level Synthesis'. In: *Proceedings of the 17th ACM Great Lakes Symposium on VLSI* (GLSVLSI '07). ACM, Stresa-Lago Maggiore, Italy, 429–434. ISBN: 978-1-59593-605-9. DOI: 10.1145/1228784.1228885 (cit. on p. 48).

John Kessenich, Boaz Ouriel and Raun Krisch. Jan. 2018. 'Spir-V Specification 1.2'. *Khronos Group*, 4, (Jan. 2018), 211 (cit. on p. 38).

Gary A. Kildall. 1973. 'A Unified Approach to Global Program Optimization'. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '73). Association for Computing Machinery, Boston, Massachusetts, 194–206. ISBN: 9781450373494. DOI: 10.1145/512927.512945 (cit. on p. 35).

Gerwin Klein, Kevin Elphinstone, Gernot Heiser et al.. 2009. 'seL4: formal verification of an OS kernel'. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (SOSP '09). Association for Computing Machinery, Big Sky, Montana, USA, 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596 (cit. on p. 44).

Jens Knoop, Oliver Rüthing and Bernhard Steffen. July 1994. 'Optimal Code Motion: Theory and Practice'. *ACM Trans. Program. Lang. Syst.*, 16, 4, (July 1994), 1117–1155. DOI: 10.1145/183432.183443 (cit. on p. 50).

Alfred Koelbl, Reily Jacoby, Himanshu Jain and Carl Pixley. Apr. 2009. 'Solver technology for system-level to RTL equivalence checking'. In: *2009 Design, Automation & Test in Europe Conference & Exhibition*. (Apr. 2009), 196–201. DOI: 10.1109/DATE.2009.5090657 (cit. on p. 43).

[SW] Alfred Koelbl, Kiran Vittal and Pratik Mahajan, *Verifying Complex Datapath Designs with HECTOR* 23rd Feb. 2021. Synopsys. URL: https://www.synopsys.com/blogs/chip-design/verifying-complex-datapath-designs-with-hector.html (cit. on p. 48).

Jérémie Koenig and Zhong Shao. 2021. 'CompCertO: Compiling Certified Open C Components'. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2021). Association for Computing Machinery, Virtual, Canada, 1095–1109. ISBN: 9781450383912. DOI: 10.1145/3453483.3454097 (cit. on p. 159).

David Koeplinger, Matthew Feldman, Raghu Prabhakar et al.. 2018. 'Spatial: A Language and Compiler for Application Accelerators'. In: *PLDI*. ACM, 296–311. DOI: 10.1145/3192366.3192379 (cit. on p. 66).

Daniel Kroening and Michael Tautschnig. 2014. 'CBMC – C Bounded Model Checker'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391. ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_26 (cit. on p. 43).

Sudipta Kundu, Sorin Lerner and Rajesh Gupta. Nov. 2007. 'Automated refinement checking of concurrent systems'. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. (Nov. 2007), 318–325. DOI: 10.1109/ICCAD.2007.4397284 (cit. on p. 48).

Sudipta Kundu, Sorin Lerner and Rajesh Gupta. 2008. 'Validating High-Level Synthesis'. In: *Computer Aided Verification*. Ed. by Aarti Gupta and Sharad Malik. Springer Berlin Heidelberg, Berlin, Heidelberg, 459–472. ISBN: 978-3-540-70545-1. DOI: 10.1007/978-3-540-70545-1_44 (cit. on pp. 47, 48).

Sakari Lahti, Panu Sjövall, Jarno Vanne and Timo D. Hämäläinen. May 2019. 'Are We There Yet? a Study on the State of High-Level Synthesis'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38, 5, (May 2019), 898–911. DOI: 10.1109/TCAD.2018.2834439 (cit. on p. 20).

Chris Lattner and Vikram Adve. 2004. 'LLVM: a compilation framework for lifelong program analysis & transformation'. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. DOI: 10.1109/CGO.2004.1281665 (cit. on p. 31).

Chris Lattner, Mehdi Amini, Uday Bondhugula et al.. Feb. 2021. 'MLIR: Scaling Compiler Infrastructure for Domain Specific Computation'. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. (Feb. 2021), 2–14. DOI: 10.1109/CGO51591.2021.9370308 (cit. on p. 31).

Marco Lattuada and Fabrizio Ferrandi. 2015. 'Code Transformations Based on Speculative SDC Scheduling'. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 71–77. DOI: 10.1109/ICCAD.2015.7372552 (cit. on p. 41).

K. Rustan M. Leino. 2010. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370. ISBN: 978-3-642-17511-4. DOI: 10.1007/978-3-642-17511-4_20 (cit. on p. 43).

Xavier Leroy. 2009a. 'A Formally Verified Compiler Back-End'. *Journal of Automated Reasoning*, 43, 4, 363. DOI: 10.1007/s10817-009-9155-4 (cit. on p. 98).

Xavier Leroy. 2006. 'Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant'. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '06). Association for Computing Machinery, Charleston, South Carolina, USA, 42–54. ISBN: 1595930272. DOI: 10.1145/1111037.1111042 (cit. on pp. 21, 52).

Xavier Leroy. July 2009b. 'Formal Verification of a Realistic Compiler'. *Commun. ACM*, 52, 7, (July 2009), 107–115. DOI: 10.1145/1538788.1538814 (cit. on pp. 21, 52, 65, 66).

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister and Christian Ferdinand. Jan. 2016. 'CompCert - A Formally Verified Optimizing Compiler'. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE.

Toulouse, France, (Jan. 2016). https://inria.hal.science/hal-01238879 (cit. on pp. 21, 52).

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen and Jian Zhang. 2018. 'Fuzzing: State of the art'. *IEEE Transactions on Reliability*, 67, 3, 1199–1218. DOI: 10.1109/TR.2018.2834476 (cit. on p. 63).

Christopher Lidbury, Andrei Lascu, Nathan Chong and Alastair F. Donaldson. 2015. 'Many-Core Compiler Fuzzing'. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '15). Association for Computing Machinery, Portland, OR, USA, 65–76. ISBN: 9781450334686. DOI: 10.1145/2737924.2737986 (cit. on pp. 20, 63).

Andreas Lööw. 2021. 'Lutsig: A Verified Verilog Compiler for Verified Circuit Development'. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2021). Association for Computing Machinery, Virtual, Denmark, 46–60. ISBN: 9781450382991. DOI: 10.1145/3437992.3439916 (cit. on pp. 46, 47, 51, 159).

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson and Anthony Fox. 2019. 'Verified Compilation on a Verified Processor'. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019). ACM, Phoenix, AZ, USA, 1041–1053. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314622 (cit. on pp. 46, 66, 79).

Andreas Lööw and Magnus O. Myreen. 2019. 'A Proof-producing Translator for Verilog Development in HOL'. In: *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering* (FormaliSE '19). IEEE Press, Montreal, Quebec, Canada, 99–108. DOI: 10.1109/FormaliSE.2019.00020 (cit. on pp. 22, 46, 79, 81, 88).

Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank and Roger A. Bringmann. Dec. 1992. 'Effective Compiler Support for Predicated Execution Using the Hyperblock'. *SIGMICRO Newsl.*, 23, 1-2, (Dec. 1992), 45–54. DOI: 10.1145/144965.144998 (cit. on pp. 38, 89).

Conor McBride and Ross Paterson. 2008. 'Applicative Programming With Effects'. *Journal of Functional Programming*, 18, 1, 1–13. DOI: 10.1017/S0956796807006326 (cit. on p. 109).

Patrick Meredith, Michael Katelman, José Meseguer and Grigore Roşu. July 2010. 'A formal executable semantics of Verilog'. In: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. (July 2010), 179–188. DOI: 10.1109/MEMCOD.2010.5558634 (cit. on pp. 66, 79).

David Monniaux and Sylvain Boulmé. 2022. 'The Trusted Computing Base of the CompCert Verified Compiler'. In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Springer International Publishing, Cham, 204–233. ISBN: 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_8 (cit. on p. 77).

Leonardo de Moura and Nikolaj Bjørner. 2008. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 43).

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn and Jakob von Raumer. 2015. 'The Lean Theorem Prover (System Description)'. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Springer International Publishing, Cham, 378–388. ISBN: 978-3-319-21401-6. DOI: 10.1007/978-3-319-21401-6_26 (cit. on p. 44).

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu and Yale N. Patt. 2011. 'Improving GPU Performance via Large Warps and Two-Level Warp Scheduling'. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-44). Association for Computing Machinery, Porto Alegre, Brazil, 308–317. ISBN: 9781450310536. DOI: 10.1145/2155620.2155656 (cit. on p. 38).

Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson and Zhiru Zhang. 2020. 'Predictable Accelerator Design with Time-Sensitive Affine Types'. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). Association for Computing Machinery, London, UK, 393–407. ISBN: 9781450376136. DOI: 10.1145/3385412.3385974 (cit. on pp. 45, 47).

Rishiyur Nikhil. 2004. 'Bluespec System Verilog: efficient, correct RTL from high level specifications'. In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. DOI: 10.1109/MEMCOD.2004.1459818 (cit. on pp. 32, 66).

Daniel H. Noronha, Jose P. Pinilla and Steven J. E. Wilton. 2017. 'Rapid circuit-specific inlining tuning for FPGA high-level synthesis'. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 1–6. DOI: 10.1109/RECONFIG.2017.8279807 (cit. on p. 70).

Karl J. Ottenstein, Robert A. Ballance and Arthur B. MacCabe. 1990. 'The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages'. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (PLDI '90). Association

for Computing Machinery, White Plains, New York, USA, 257–271. ISBN: 0897913647. DOI: 10.1145/93542.93578 (cit. on p. 35).

J. Ou and V.K. Prasanna. Apr. 2005. 'MATLAB/Simulink based hardware/software co-simulation for designing using FPGA configured soft processors'. In: *19th IEEE International Parallel and Distributed Processing Symposium.* (Apr. 2005), 1–8. DOI: 10.1109/IPDPS.2005.275 (cit. on p. 30).

Ian Page and Wayne Luk. 1991. 'Compiling occam into field-programmable gate arrays'. In: *FPGAs, Oxford Workshop on Field Programmable Logic and Applications.* Vol. 15. Citeseer, 271–283 (cit. on pp. 32, 49, 50, 66).

Barry M. Pangrle and Daniel D. Gajski. 1987. 'Design Tools for Intelligent Silicon Compilation'. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 6, 6, 1098–1112. DOI: 10.1109/TCAD.1987.1270350 (cit. on p. 92).

Michalis Pardalos, Yann Herklotz and John Wickerson. 2022b. 'Resource Sharing for Verified High-Level Synthesis'. In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, 1–6. DOI: 10.1109/FCCM53951.2022.9786208.

P.G. Paulin and J.P. Knight. 1989. 'Force-Directed Scheduling for the Behavioral Synthesis of ASICs'. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8, 6, 661–679. DOI: 10.1109/43.31522 (cit. on p. 40).

Lawrence C Paulson. 1994. *Isabelle: A generic theorem prover.* Springer (cit. on p. 44).

Maxime Pelcat, Cédric Bourrasset, Luca Maggiani and François Berry. 2016. 'Design productivity of a high level synthesis compiler versus HDL'. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 140–147. DOI: 10.1109/SAMOS.2016.7818341 (cit. on p. 19).

Juan Perna and Jim Woodcock. 2012. 'Mechanised Wire-Wise Verification of Handel-C Synthesis'. *Science of Computer Programming*, 77, 4, 424–443. DOI: 10.1016/j.scico.2010.02.007 (cit. on pp. 46, 47, 50).

Juan Perna, Jim Woodcock, Augusto Sampaio and Juliano Iyoda. 1st Dec. 2011. 'Correct Hardware Synthesis'. *Acta Informatica*, 48, 7, (1st Dec. 2011), 363–396. DOI: 10.1007/s00236-011-0142-y (cit. on p. 46).

Christian Pilato and Fabrizio Ferrandi. 2013. 'Bambu: A modular framework for the high level synthesis of memory-intensive applications'. In: *2013 23rd International Conference*

on Field programmable Logic and Applications, 1–4. DOI: 10.1109/FPL.2013.6645550 (cit. on pp. 19, 23, 32, 65, 147).

A. Pnueli, M. Siegel and E. Singerman. 1998. 'Translation validation'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166. ISBN: 978-3-540-69753-4. DOI: 10.1007/BFb0054170 (cit. on pp. 21, 45, 48).

Louis-Noël Pouchet. 2020. *PolyBench/C: the Polyhedral Benchmark suite*. (2020). http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (cit. on p. 148).

Louis-Noël Pouchet, Peng Zhang, Ponnuswamy Sadayappan and Jason Cong. 2013. 'Polyhedral-based data reuse optimization for configurable computing'. In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 29–38. DOI: https://doi.org/10.1145/2435264.2435273 (cit. on p. 148).

B. Ramakrishna Rau. 1st Feb. 1996. 'Iterative Modulo Scheduling'. *International Journal of Parallel Programming*, 24, 1, (1st Feb. 1996), 3–64. DOI: 10.1007/BF03356742 (cit. on p. 41).

B. Ramakrishna Rau, M. Lee, P. P. Tirumalai and M. S. Schlansker. 1992a. 'Register Allocation for Software Pipelined Loops'. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (PLDI '92). Association for Computing Machinery, San Francisco, California, USA, 283–299. ISBN: 0897914759. DOI: 10.1145/143095.143141 (cit. on p. 157).

B. Ramakrishna Rau, Michael S. Schlansker and P. P. Tirumalai. 1992b. 'Code Generation Schema for modulo Scheduled Loops'. In: *Proceedings of the 25th Annual International Symposium on Microarchitecture* (MICRO 25). IEEE Computer Society Press, Portland, Oregon, USA, 158–169. ISBN: 0818631759. DOI: 10.1145/144965.145795 (cit. on p. 157).

Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi and Jeremy Kepner. 2020. 'Survey of Machine Learning Accelerators'. In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–12. DOI: 10.1109/HPEC43674.2020.9286149 (cit. on p. 19).

Carmine Rizzi, Andrea Guerrieri and Lana Josipović. July 2023. 'An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits'. In: *Proceedings of the 60rd ACM/IEEE Design Automation Conference*. San Francisco, CA, (July 2023). DOI: 10.1109/DAC56929.2023.10247686 (cit. on p. 152).

Jeff Roane. 2023. *Automated HW/SW Co-Design of DSP Systems Composed of Processors and Hardware Accelerators*. Tech. rep. Cadence. Retrieved 14th Dec. 2023 from https://www.cadence.com/en_US/home/resources/white-papers/automated-hw-sw-co-design-of-dsp

-systems-composed-of-processors-and-hardware-accelerators-wp.html (cit. on pp. 19, 39).

Fabian Schuiki, Andreas Kurth, Tobias Grosser and Luca Benini. 2020. 'LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages'. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). ACM, London, UK, 258–271. ISBN: 9781450376136. DOI: 10.1145/3385412.3386024 (cit. on p. 66).

Koushik Sen, George Necula, Liang Gong and Wontae Choi. 2015. 'MultiSE: Multi-Path Symbolic Execution Using Value Summaries'. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2015). Association for Computing Machinery, Bergamo, Italy, 842–853. ISBN: 9781450336758. DOI: 10.1145/2786805.2786830 (cit. on pp. 103, 108, 109).

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan and Peter Sewell. June 2013. 'CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency'. *J. ACM*, 60, 3, (June 2013). DOI: 10.1145/2487241.2487248 (cit. on p. 86).

Siemens. 2021. *Catapult High-Level Synthesis*. (2021). Retrieved 14th Nov. 2023 from https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/ (cit. on pp. 21, 32, 39, 45, 47).

Cyril Six, Sylvain Boulmé and David Monniaux. Nov. 2020. 'Certified and Efficient Instruction Scheduling: Application to Interlocked VLIW Processors'. *Proc. ACM Program. Lang.*, 4, OOPSLA, (Nov. 2020). DOI: 10.1145/3428197 (cit. on pp. 58–60).

[SW] Cyril Six, Léo Gourdin, Benjamin Bonneau, Alexandre Bérard, Sylvain Boulmé and David Monniaux, *CompCert KVX* 2023. Grenoble-INP, CNRS and Kalray. URL: https://www-verimag.imag.fr/the-KVX-CompCert-Compiler.html (cit. on p. 58).

Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse and Nicolas Nardino. 2022. 'Formally Verified Superblock Scheduling'. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2022). Association for Computing Machinery, Philadelphia, PA, USA, 40–54. ISBN: 9781450391825. DOI: 10.1145/3497775.3503679 (cit. on pp. 23, 61, 90, 92, 93, 116, 117, 148).

Chengnian Sun, Vu Le, Qirun Zhang and Zhendong Su. 2016. 'Toward Understanding Compiler Bugs in GCC and LLVM'. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 294–305. DOI: 10.1145/2931037.2931074 (cit. on p. 63).

[SW] Synopsys, *VC Formal: Leading Formal Innovations* 2023. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.htmlRetrieved 20th Dec. 2023 from (cit. on p. 43).

Mingxing Tan, Steve Dai, Udit Gupta and Zhiru Zhang. Feb. 2015. 'Mapping-aware constrained scheduling for LUT-based FPGAs'. In: *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, (Feb. 2015), 190–9. DOI: 10.1145/2684746.2689063 (cit. on p. 152).

David B. Thomas. 2016. 'Synthesisable Recursion for C++ HLS Tools'. In: *ASAP*. IEEE Computer Society, 91–98. DOI: 10.1109/ASAP.2016.7760777 (cit. on p. 70).

Jean-Baptiste Tristan and Xavier Leroy. 2010. 'A Simple, Verified Validator for Software Pipelining'. In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '10). Association for Computing Machinery, Madrid, Spain, 83–92. ISBN: 9781605584799. DOI: 10.1145/1706299.1706311 (cit. on p. 157).

Jean-Baptiste Tristan and Xavier Leroy. 2008. 'Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '08). Association for Computing Machinery, San Francisco, California, USA, 17–27. ISBN: 9781595936899. DOI: 10.1145/1328438.1328444 (cit. on pp. 21, 23, 58, 59, 90, 92, 93, 102, 106, 107, 111, 116, 117).

Peng Tu and David Padua. 1995. 'Efficient Building and Placing of Gating Functions'. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (PLDI '95). Association for Computing Machinery, La Jolla, California, USA, 47–55. ISBN: 0897916972. DOI: 10.1145/207110.207115 (cit. on p. 35).

Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li and Zhiru Zhang. Nov. 2020. 'Accurate operation delay prediction for FPGA HLS using graph neural networks'. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. Virtual, (Nov. 2020), 1–9. DOI: 10.1145/3400302.3415657 (cit. on p. 152).

Hanyu Wang, Carmine Rizzi and Lana Josipović. Oct. 2023. 'MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization'. In: *Proceedings of the 42nd IEEE/ACM Intl. Conference on Computer-Aided Design*. San Francisco, CA, (Oct. 2023). DOI: 10.1109/ICCAD57390.2023.10323639 (cit. on p. 152).

Yuting Wang, Xiangzhe Xu, Pierre Wilke and Zhong Shao. Nov. 2020. 'CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files'. *Proc. ACM Program. Lang.*, 4, OOPSLA, (Nov. 2020). DOI: 10.1145/3428265 (cit. on p. 86).

Neil H. E. Weste and David Money Harris. 2010. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson. ISBN: 9780321547743 (cit. on p. 79).

Xuejun Yang, Yang Chen, Eric Eide and John Regehr. 2011. 'Finding and Understanding Bugs in C Compilers'. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11). Association for Computing Machinery, San Jose, California, USA, 283–294. ISBN: 9781450306638. DOI: 10.1145/1993498.1993532 (cit. on pp. 21, 50, 63, 77, 153).

[SW] YosysHQ, *SymbiYosys (sby)* 2023. URL: https://github.com/YosysHQ/SymbiYosys Retrieved 20th Dec. 2023 from (cit. on p. 43).

Youngsik Kim, S. Kopuri and N. Mansouri. Mar. 2004. 'Automated formal verification of scheduling process using finite state machines with datapath (FSMD)'. In: *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*. (Mar. 2004), 110–115. DOI: 10.1109/ISQED.2004.1283659 (cit. on pp. 21, 45, 48).

Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu and Zhendong Su. 2019. 'Finding and Understanding Bugs in Software Model Checkers'. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 763–773. DOI: 10.1145/3338906.3338932 (cit. on p. 63).

Zhiru Zhang and Bin Liu. Nov. 2013. 'SDC-based modulo scheduling for pipeline synthesis'. In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. (Nov. 2013), 211–218. DOI: 10.1109/ICCAD.2013.6691121 (cit. on pp. 41, 120).

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin and Steve Zdancewic. 2012. 'Formalizing the LLVM Intermediate Representation for Verified Program Transformations'. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '12). Association for Computing Machinery, Philadelphia, PA, USA, 427–440. ISBN: 9781450310833. DOI: 10.1145/2103656.2103709 (cit. on p. 66).

Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang and Bingsheng He. 2017. 'COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications'. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 430–437. DOI: 10.1109/ICCAD.2017.8203809 (cit. on p. 148).

Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow and Deming Chen. 2014. 'Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs'. In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA '14). Association for Computing Machinery, Monterey, California, USA, 1–10. ISBN: 9781450326711. DOI: 10.1145/2554688.2554775 (cit. on p. 152).

Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong and Jason Cong. Sept. 2013. 'Improving polyhedral code generation for high-level synthesis'. In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. (Sept. 2013), 1–10. DOI: 10.1109/CODES-ISSS.2013.6659002 (cit. on p. 148).