Imperial College London

MEng Final Year Project

Imperial College London

Department of Electrical and Electronic Engineering

Formally verified resource sharing for High Level Synthesis

Author:
Michail Pardalos

Supervisor:
Dr. John Wickerson

Second Marker:
Dr. James J. Davis

Abstract

We present an extension to Vericert, a verified High Level Synthesis Compiler to implement resource sharing. Vericert currently generates the hardware to implement a given function in an input program multiple times, matching the number of calls to the function. We remove this duplication, reducing the area usage of generated hardware. Our benchmarking shows the generated hardware having a resource usage of 87.9% of the original on average and 41% in the best case, for only a 0.2% average decrease in max frequency and 0.7% average increase in cycle count. We partially extend the formal proof of correctness of the compiler to our changes, increasing confidence in their correctness, and outline the steps to be taken for full verification of our work.

Contents

CHAPTER 1

Introduction

The need for faster, more energy-efficient computation has, in recent years, caused a surge in the need for custom hardware accelerators. Such devices are commonly designed using a hardware description language such as Verilog or VHDL. The complexities of designing hardware in such a language, as well as the abundance of engineers trained in software rather than hardware development has meant that high-level synthesis tools have become an enticing option. These tools, examples of which include Intel's i++ [12], and Vitis HLS from Xilinx [18], allow one to generate a hardware design from a program written in a high-level software programming language, usually C or C++. This allows an engineer's software programming skills to be applied to hardware design with less training necessary compared to learning a dedicated hardware design language. This capability is especially useful for quick prototyping of new products or features, which is more easily done in a language like C with a High Level Synthesis tool as opposed to a dedicated hardware design language [7].

These tools, while incredibly useful, are also known to be unreliable. Previous work by Du et al. in 2021 [6] has found extensive *miscompilation* bugs in commercial HLS tools including Vivado HLS (now Vitis HLS) [18], Intel i++ [12] and LegUp [3]. This instability can be a significant hindrance in the development process. This is compounded by the longer iteration times of hardware design compared to software, making the setback when a bug is encountered much more significant. It is therefore essential to ensure that all software used in this process, including the high-level synthesis tool, is as reliable as possible.

Vericert [8] is a High-Level Synthesis compiler which aims to address this issue. It has been verified correct by the highest possible standard: machine-checked formal proof. It achieves that by providing a proof, checked using the Coq proof assistant, that every step of its translation from C to Verilog preserves the semantics of (i.e. behaves the same way as) its input program. This proof means that we can always trust any Verilog produced

by Vericert to behave the same way as the C program given to it as input. It is based on the CompCert [13] verified C compiler.

Clearly, however, it is not enough for a high-level synthesis tool to simply be *correct*. The generated hardware needs to also satisfy a number of other qualities, including high throughput, low latency, and *area efficiency*; that is, using as few hardware resources as possible. This is desirable as it can allow for fitting "more" hardware on the same chip, or simply using a smaller chip, reducing costs. A common optimisation in HLS tools for improving area usage is *resource sharing*. That is, avoiding generating the same hardware more than once, and, instead, re-using that hardware for more than one purpose. Vericert does not currently perform this optimisation. It was the goal of this project to implement it, and, keeping with the rest of Vericert, formally prove it correct to the greatest possible extent.

1.1. Project Aims

We define the following as goals of our project:

1. Implement resource sharing in Vericert.
2. Provide a tangible improvement in area usage of generated hardware.
3. Minimise the impact on other performance metrics, including clock frequency and cycle count.
4. Prove the optimisation correct to the greatest extend possible.

CHAPTER 2

Background

2.1. Coq

Coq [2] is the language and proof assistant on which this project is based. It is an established system that has been used in both proving mathematical theorems and verified software development, with one of the biggest projects developed using it being CompCert [13], a formally verified C compiler. This project is indirectly based on CompCert, which is explained in greater detail in the next section.

The core language of the Coq system, Gallina, is a pure functional language, similar in many ways to other typed functional languages such as Haskell or OCaml. Its main distinguishing feature is *dependent types*. This means that types in Gallina can *depend* (i.e. contain) values or arbitrary terms. For example, one may encode the type of length-indexed vectors as follows:

```
1 Inductive Vec (A : Type) : nat -> Type :=
2   | Nil : Vec A 0
3   | Cons : forall (n:nat), A -> Vec A n -> Vec A (1 + n).
```

This defines a type `Vec` with two constructors, `Nil` which constructs a vector of length 0, and `Cons`, which constructs a vector of length $1 + n$ from an element of type `A` and another vector of length n. We can then write functions which operate on terms of this type while also preserving certain properties. For example, we can write a function to concatenate two Vecs and encode *in its type* that its output has a length equal to the sum of the length of its inputs.

```
Fixpoint concat {A : Type} {n k : nat}
         (l1 : Vec A n)
         (l2 : Vec A k)
  : Vec A (n + k) :=
  match l1 with
  | Nil => l2
  | Cons x xs => Cons x (concat xs l2)
  end.
```

The syntax of Gallina is similar to languages in the family of the ML programming language. Function application is simply juxtaposition ("`f x`" is `f` applied to `x`), the `match` construct performs pattern matching, and "`:`" means "has type".

Types are usually seen as simply encoding the set of values a term can take. Saying that some term `x` has type `nat` means that `x` has a value in the set $0, 1, \ldots$. The same holds for function types. If a function has type `A -> B` then it is an element of the set of mappings from values of `A` to values of `B`.

There is however an alternative interpretation. Types can be seen as *logical propositions*, where if a term of a certain type exists, then the proposition represented by it is true. A term can be seen as a *proof* of its type. For most types, the proposition stated by them is trivial. Certain types can be valuable as propositions, however. One can, for example, define a type for equality:

```
Inductive Equal {A: Type} : A -> A -> Type :=
    | Refl : forall x, Equal x x.
```

The type `equal` only has a term when its two type parameters are the same. So, `Refl : equal 1 1` and `Refl : equal (1+2) 3`, but `Refl` cannot have type `equal 1 2` because the term `1` is different from the term `2`.

We can then compose types to create more complex propositions. Logical conjunction (the "and" operator) can be encoded as the pair type `A * B`. A term of type `A * B` can only be constructed from a proof (term) of `A` and a proof (term) of `B`. It is therefore a proof that both `A` and `B` are true. Logical disjunction (the "or" operator) can be encoded as a sum type with two constructors, so that a term of type `Or l r` can be constructed from either a term of type `l` *or* a term of type `r`:

```
Inductive Or (l : Type) (r : Type) : Type :=
  | Or_left : l -> Or l r
  | Or_right : r -> Or l r.
```

Functions are equivalent to logical implication. A function of type `A -> B` is proof that if `A` is true (a term of type `A` exists) then `B` is true (a term of `B` exists).

This idea of a correspondence between types and logical propositions, is known as the "Curry-Howard Isomorphism" [9], and is at the core of what allows Coq to function as

both a programming language and a proof assistant. In Coq, and in other dependently-typed languages, there is no distinction between types and propositions, or proofs and programs. They are one and the same. We can, therefore, write a program `prog` in Coq to perform the task we want (in our case, compile C to Verilog) and then write another program which has type `prf : IsCorrect prog` for some appropriate definition of `IsCorrect` to serve as a proof that our "functional" code in `prog` behaves as we expect. This second program (or proof, depending on how one prefers to view it) need not be executed. It establishes that the proposition stated by its type holds simply by having that type. The two programs together form a verified implementation of our specification.

2.2. CompCert

Originally introduced in Leroy 2009 [13], CompCert is a verified compiler for C99. It compiles the vast majority of C99 (with small caveats), and generates code faster than GCC's `-O0` optimisation level, (no optimisations enabled) but slightly slower that `-O1` (containing only some optimisations).

It is internally architected in terms of ten intermediate languages. This both simplifies implementation, as each pass has fewer changes to make, and verification, as semantic preservation proofs become easier for the simpler transformations and can be subsequently composed into the proof for the entire compiler. An indirect benefit of this architecture is that it allows for extensibility. A backend can be added as a transformation from an intermediate language into the target. The only proof that is then needed is of that translation. A backend could also introduce other intermediate languages if that makes its implementation or proof easier.

This project will only interface with one of CompCert's intermediate languages: RTL. This is a minimal 3-address-code language represented as a control-flow graph where each node contains one instruction. It operates on an infinite supply of registers and supports arithmetic, memory loads and stores, function calls (direct and indirect) and branching. In normal C-to-assembly flow, the next stage after this language is performing register allocation, moving into the LTL language.

2.3. Vericert

Introduced by Herklotz et al. in 2020 [8], Vericert is a verified C-to-Verilog High Level Synthesis tool. It supports "all C constructs except for case statements, function pointers, recursive function calls, integers larger than 32 bits, floats, and global variables." [8]. It is an extension of CompCert, essentially adding a Verilog backend to the existing verified C compiler. In its current form, it performs no optimisations, other than those performed by CompCert in stages earlier than those added by Vericert. This results in performance generally about 1 order of magnitude slower than the designs generated by comparable, unverified tools like LegUp [3].

Vericert adds two languages to CompCert: HTL, which is generated from CompCert's RTL, and Verilog which is generated from HTL and is the output of the compiler. HTL

is the representation of a Finite State Machine with Datapath (FSMD) [10]. It contains Verilog statements split into a data- and control-path which execute concurrently. The two paths are structured as Control Flow Graphs, same as in CompCert's RTL, but with matching states in both paths. The control path sets a register controlling the next state, while computations happen in the datapath.

Listing 1 demonstrate a simple example of HTL code. Note that this is only a textual representation of this intermediate language, and so it is missing some of the details of its representation inside of Vericert. The statements in the control- and datapaths execute concurrently, and the state which executes is chosen at each step (or clock cycle) based on the value of the `status` register. Setting the `finish` register signals termination of the module's execution, using the value of the `return` register as the output value.

```c
int main () {
    int x = 1;
    int y = 2;
    int z = x + y;
    return z;
}
```

(a) Example C Code

```
main() {
  datapath {
    5: x <= 32'd1;
    4: y <= 32'd2;
    3: z <= {{x + y} + 32'd0};
    2: reg_1 <= z;
    1: finish = 32'd1;
       return = reg_1;
  }

  controllogic {
    5: state <= 32'd4;
    4: state <= 32'd3;
    3: state <= 32'd2;
    2: state <= 32'd1;
    1: ;
  }
}
```

(b) Corresponding HTL Code

Listing 1: A simple example of the HTL intermediate language

Since HTL consists of Verilog statements, translation to full Verilog is straightforward. An HTL program is translated into a Verilog state machine controlled by two always blocks, one containing the control path and one containing the datapath.

One core design decision of Vericert that will be addressed in this project is how function calls are implemented. Vericert performs an inlining pass on the entire program at the RTL level, eliminating function calls. This in turn means that the RTL code for functions that are called more than once during the program is duplicated and therefore the corresponding Verilog code is also generated more than once. Given the sequential nature of the generated code, this is strictly unnecessary. Only one of those instances will have control at any one time. This decision therefore increases design area consumption for no gain in performance. It is this problem that this project aims to improve on.

2.4. Formal Verilog semantics

While Verilog is based on a standard [11], that standard is written in natural language, which does not allow for formal reasoning. This means that in order to construct proofs about Verilog programs, a formalisation of this standard needs to be created. This is not entirely straightforward due to Verilog being inherently concurrent, as well having several complex features.

Vericert uses a modified version of formalisation due to Lööw and Myreen [14] which we briefly introduce here.

2.4.1. Language Semantics

Giving a formal semantics to a language allows for abstractly and symbolically reasoning about the language and its execution. Language semantics can be divided into various different types. The semantics presented here would be classified as *operational semantics*, meaning that they describe computation purely in terms of objects within the language. This is in contrast to another type like *denotational semantics* which maps the objects being reasoned about into a separate domain with existing rules which allow for reasoning. Operational semantics can be further classed into *small-step* and *big-step*. *Small-step* operational semantics describe individual steps of a computation, whereas *big-step* operational semantics map expressions to their end results directly.

2.4.2. Lööw and Myreen

The semantics of Lööw and Myreen in 2019 [14] are presented in three levels. It presents a big-step semantics for Verilog expressions and statements and a small-step semantics for evaluating modules, where each step corresponds to a clock cycle. All 3 levels operate on the module state, as well as a map from variable names to values representing "external state", such as non-deterministic inputs, called *fext* in the paper.

The semantics for statements are given as a function from *fext*, the current module state and a statement to the next module state. Similarly for expressions, but returning the value of the expression instead.

The semantics for modules are somewhat more interesting as they need to account for continuous assignment. All always blocks are executed in order, keeping continuous writes separate and only "committing" them to the module state at the end of every cycle.

This formalisation does not support any form of continuous assignment or module instantiation. It also requires that different always blocks not "interfere" with each other. This is because the semantics is deterministic, and as such is invalid for programs which have multiple possible interleavings of multiple always blocks. This is enforced by two *syntactic* conditions:

1. No two blocks write to the same variable.
2. No two blocks perform a blocking read and a blocking write to the same variable.

2.4.3. Vericert

The semantics used in Vericert [8] are heavily based on Lööw and Myreen. The main differences come from needing to support arrays and having to conform to CompCert's execution model.

According to the Vericert paper, in the Lööw and Myreen semantics, writing to an array using both continuous and blocking assignment on different indices in the same cycle would treat the array as a single variable and thus the blocking assignment's value would be overridden. As Vericert uses this pattern, this has been corrected by separating array writes from other writes and including the index in the "queued" write.

CompCert's computational model assumes that programs in all intermediate languages are divided into functions which can be called and will eventually return. This is represented by execution state having three variants, the Callstate, entered when a function is called, Returnstate entered when a function returns and plain State which is active at all other times. The Verilog semantics therefore need to map to these states. To support this, an explicit program counter is added, as well as a reset signal (which is asserted when the CallState is entered), a `return` output signal, a `done` signal, and an explicit stack. This stack is only used for values which will be accessed by pointer (for example arrays). As Vericert inlines function calls away, the call and return states are entered only once (at the start and end of the main module's execution) and only one stack frame is ever created.

Another important decision made was to remove the support for external inputs to modules (other than the control signals already listed). Vericert currently generates a single module, generated from the main function in the input C code. The main function is also assumed to have no arguments. There is therefore no need to allow external inputs to a generated module other than control signals.

Other changes to the semantics include adding register declarations explicitly to the semantics in order to prove they are being generated correctly, and simplifying values from their representation as arrays of booleans in Lööw and Myreen to 32-bit integers.

Generating verilog outside of this subset would be trivial. However, extending the semantics would likely entail significant effort. We have therefore chosen to stay within the existing semantics and implement any new features using only them.

2.5. Resource Sharing

Resource sharing is a feature shared (and expected) by most HLS compilers. Coussy et al. 2009 [5] presents a typical architecture for an HLS compiler. In the typical architecture generated by HLS, a number of "functional components" are selected from an RTL component library to be *allocated* in the generated design. The number of instances of these components can vary based on the needs of the specific design. Operations which use these components are then *scheduled* to a clock cycle (or to a series of clock cycles) during which an instance of the required component is unused and the inputs to the operation are available. In commercial HLS compilers such as Intel i++[12] or Xilinx Vitis[18] sharing can also be at the level of functions. The programmer can guide

the compiler as to which functions should be shared using appropriate pragmas for each compiler.

2.5.1. Resource sharing in verified High Level Synthesis

Handel-C [1] is a variant of C extended with parallel constructs, intended for describing hardware. There has been a verified HLS compiler for Handel-C introduced by Perna et al. in 2011 [16] and extended in 2012 [15]. This compiler works applying reduction steps to the source program, reducing it to a "highly parallel state machine". This is possible, and practical for Handel-C, as the language allows the programmer to describe the parallelism explicitly. This is different from Vericert's model of compilation which follows the more traditional compilation method of translating the input program statement by statement. BEDROC [4] is another, older, verified High Level Synthesis tool for the HardwarePal language. It also does not implement resources sharing on the function level as "Recursion is not allowed in HardwarePal, and all procedure and function calls are expanded in the front-end." [4].

2.5.2. Resource sharing in Vericert

Vericert currently performs allocation of hardware resources in a very direct manner. It creates one instance of the hardware required for an operation or function for every instance of the operation in the source program. It also performs performs no explicit scheduling. It, approximately, maps every line of the source C program to one state in the output state machine, in source program order. These two facts mean that, while multiple instances of the same hardware exist, they are all scheduled to different clock cycles. Any potential for parallelism is therefore not exploited, while more area than required for such a sequential design is used.

This project is the foundation of the solution to this problem. We allow for Vericert to instantiate the hardware for functions which are called multiple times only once, as opposed to the multiple, redundant copies it generates currently. This is akin to making the "allocation" step in the typical HLS flow explicit. The immediate effect of this change is reducing area usage. As a side-effect, we expect making this process explicit in the compiler to enable further improvements, chief among which is better scheduling. We outline these possible improvements in Chapter 6.

CHAPTER 3

Analysis and Design

The design had to take multiple external factors into account. Chief among these are the constraints of the existing code to which this project forms an extension, this is the code of CompCert and Vericert. Furthermore, we had to ensure that, not only will we generate compact and performant Verilog – our "performance" goal – but also do so in a manner amenable to verification – the "correctness" goal.

We decided on implementing resource sharing at the level of source functions. A more fine-grained level of resource sharing was certainly possible, for example sharing the implementations of only certain operations, or extracting program fragments into shared resources. Functions, we decided, would provide the best ratio of impact on resource usage to implementation effort. They are provided by the programmer, and so no analysis in necessary to split code into shareable fragments. Instead, we need only decide *which* functions could and should be shared. Furthermore, in terms of the proof, the semantics added to the generated code to support this resource sharing could be matched to CompCert's call semantics. This would be a much more difficult task if trying to apply sharing on any other sections of code.

As described in Section 2.3, Vericert outputs Verilog describing an FSMD. With our planned changes, we would be translating each RTL function into a separate state machine. Calls in RTL could then map to HTL states which transfer control between state machines. It was also decided early on that these separate state machines would all have to be part of the same Verilog module. While encapsulating each in its own module would be the most "natural", it required significant changes to Vericert's semantics for Verilog, and was therefore decided against.

In the current implementation of Vericert, *all* functions are inlined after RTL generation and before HTL generation. This means that generated HTL consists of a single module, and that the HTL generation pass does not have to handle any RTL call instructions as input (since inlining removes them). The single generated HTL module is then

translated into a single Verilog module with no significant changes (since HTL consists of Verilog statements). In order to share the hardware for functions, the first step would be to preserve the call structure that is eliminated in this inlining pass. To this end, we initially decided to eliminate the inlining pass, preserving the entire call structure to be used by HTL generation. Here, however, we encountered a significant problem: memory accesses.

The current correctness proof of the HTL generation pass assumes that all pointers in the program refer to data in a single stackframe: that of the main function. This assumption is valid because, due to inlining, the only RTL function that HTL generation sees is the main. Removing inlining would break this assumption. The parts of the proof which use this assumption are to do with the translation of the `load` and `store` RTL instructions, and are the most complex in the entire HTL generation proof. Altering them was decided to be out of the scope of the project. As an alternative, we settled on only *partially* eliminating the inlining pass. We decided to inline all RTL functions containing `load` or `store` instructions, but keep functions which contained neither. The "all loads and stores refer to the main stackframe" assumption, would therefore remain valid. In terms of generated hardware, this change means that functions that perform loads or stores would still be duplicated at all call sites, but those which have neither would be shared.

3.1. Changes to HTL and the externctrl map

With functions and calls preserved past RTL generation, we could then translate each function into a separate HTL module. Almost all RTL instructions would be translated in the same manner as before, with the exception of call instructions. In the final Verilog, transfer of control between state machines should be done by setting the appropriate registers to set the arguments and initiate a call, and then waiting for the callee's `finish` register to be asserted to read the return value and continue execution of the caller. This is *not* be directly representable in HTL. The problem is that, while HTL consists of Verilog statements, each HTL module is translated independently, *with its own register space*. This means that there is no way for code in the calling module to refer to control registers in a called module.

This problem, a most other in computer science, can be solved by adding another level of indirection. We decided to add a mapping, labelled `externctrl`, allowing local registers in a module to map to registers in other modules. This will be part of every HTL module. It will map local registers names to pairs of module identifiers and register names in those modules. Then, in our semantics for HTL, we would equate setting the value of a register which appears as a key in the `externctrl` map, to setting that value on the register it maps to, and vice-versa.

This added map, however, has no direct equivalent in Verilog. Therefore, before the final step of Verilog generation, we would need to eliminate this `externctrl` map. This means making sure that registers connected through the `externctrl` map have *the same name* in the final Verilog. This has the same effect as the `externctrl` map describes

semantically: the two registers always have the same value. In HTL it happens because we decide so in the semantics, and in Verilog because they are the same register.

We perform this elimination in two steps. First, we perform a renaming pass through the whole program, making all register names globally unique. This is necessary to avoid unintended conflicts between register names in different modules, as register names are originally only unique within their own module. We then do a second pass, renaming registers present in `externctrl` to the name of the register they target.

The final step of generating Verilog can be, for the most part, left as-is, but with one major exception: the Verilog translation of an HTL module would recursively include the Verilog translations of all HTL modules it calls. This is because the Verilog semantics in Vericert do not support module instantiations. Mapping HTL modules to Verilog modules would therefore simply complicate the semantics for no benefit, as instantiating a Verilog module is equivalent to simply inserting all its code.

CHAPTER 4

Implementation

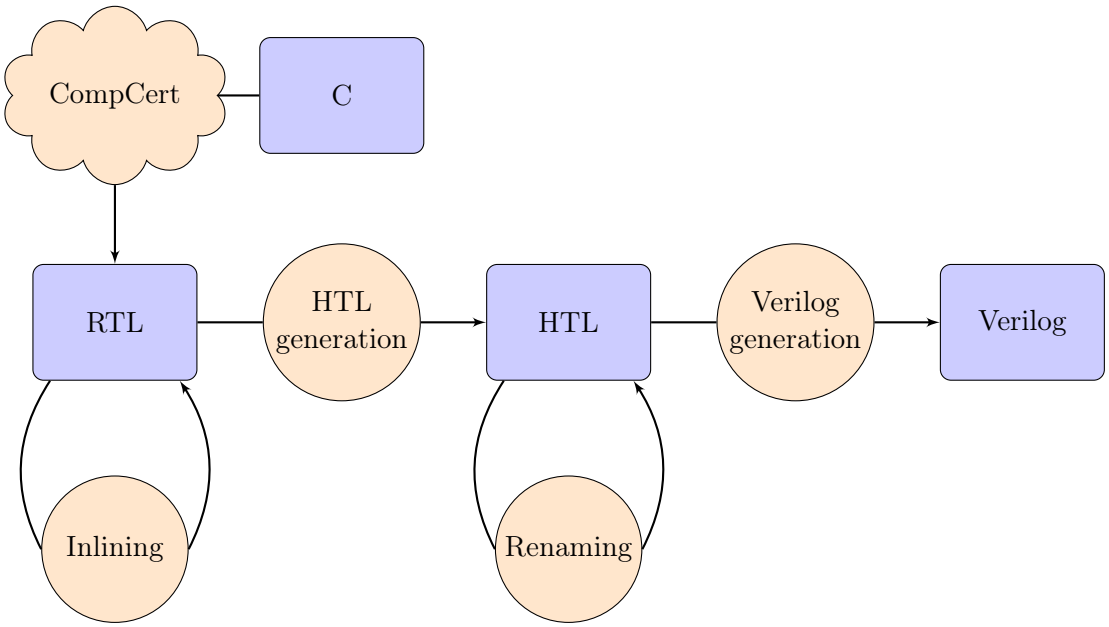

Figure 4.1.: Vericert backend passes

We will present the implementation of the ideas outlined in Chapter 3. As described in that section, the aim of the project was to synthesise hardware for certain functions in a program only once, and to then use that hardware (or resource) for every call to that function, i.e. to share it. The details of that translation will be described in terms of the separate passes of the compiler, providing examples of translated code in listings. The flow of the input program through these passes and the intermediate languages is show

```c
int add(int a, int b) {
    return a + b;
}

int main() {
    int v = 0;
    v = add(v, 1);
    v = add(v, 2);
    return v;
}
```

(a) C code

```
add(x2, x1) {
    2:  x3 = x2 + x1 + 0 (int)
    1:  return x3
}

main() {
    9:  x3 = 0
    8:  x6 = 1
    7:  x1 = "add"(x3, x6)
    6:  x3 = x1
    5:  x5 = 2
    4:  x2 = "add"(x3, x5)
    3:  x3 = x2
    2:  x4 = x3
    1:  return x4
}
```

(b) RTL code

Listing 2: C and RTL code of example

in Figure 4.1. We end the chapter with an evaluation of the generated code in terms of speed and area usage. The correctness proof is not addressed in this chapter, as it is the subject of Chapter 5

As a running example, we will use the C program in Listing 2a.

4.1. Inlining

The first pass in Vericert's pipeline, beginning with the RTL language, is the inlining pass. Here, we inline certain functions whose hardware cannot be shared in the latter passes. As established in Chapter 3, functions which perform load or stores are chosen to be inlined, while all other functions are skipped. Since this process is recursive, it ensures that `load` or `store` instructions are only present in the main function. As explained in Chapter 3, this is necessary not due to technical reasons but due to the complexity of revising the proof. We plan on removing this limitation in future version of Vericert, as outlined in Section 6.2.

This pass is implemented as a modification of the existing inlining pass used as an optimisation in CompCert. The code for this pass is split into two sections. One that chooses functions to inline and one that performs the inlining. Only the latter is verified. The proof shows that the pass performs *any* chosen inlining correctly. It is therefore correct irrespective of the choice of functions to be inlined. Since our modification only affects the choice of functions, there was no need to make any changes to the proof.

4.2. HTL generation

HTL is Vericert's first "hardware" language, generated from CompCert's RTL, a 3-address-code (3AC) language. It represents an FSMD, where the instructions in each state are Verilog statements. See Section 2.3 for more details. Vericert's HTL generation step covers the majority of RTL instructions, with the exception of `jumptable`, `call`, and `tailcall` instructions. Our contribution in this translation step is to allow the translation of `call` instructions. As described in Chapter 3, this is done through the addition of the `externctrl` map.

HTL generation done independently for each RTL function. It proceeds by translating each RTL instruction to one or more HTL states, as well as (possibly) adding register declarations. RTL `call` instructions are assumed to have been removed in a previous inlining pass and so the compiler simply throws an error in case one is encountered.

Our goal at this stage is to translate RTL call instructions, as well as to make sure to add any additional structure required to make modules callable. To that end, as outlined in Chapter 3, we add the `externctrl` map, mapping register names to pairs of module names and labels. We then allow the translation function to add entries to this map. When the translation requires setting a control register in a called module a new register is created and added to `externctrl`, mapping it to the appropriate module and control register.

An RTL call translates to 2 states in the HTL state machine. The first copies the argument values over to the appropriate parameter registers (which are also in `externctrl`), asserts the called module's `reset` signal, and proceeds to the following state on the next clock cycle. The following state de-asserts the reset signal, allowing the called module to proceed in its execution, and does not proceed to the next state until the called module's `finish` signal is asserted, meanwhile continuously assigning the `return` signal of the called module to the call's destination register. This has the effect of blocking while the called module executes and then proceeding to the next state once its execution is complete, with the result copied to the destination register.

We also extend the translation of `return` instructions, making modules enter an "idle" state after returning. This state (marked as "Idle State" in Listing 3) simply holds the module's finish signal de-asserted and never proceeds to another state (meaning that the only way to exit it is by resetting the module). This is to ensure that, the next time the module is called, its finish signal is found de-asserted. If this state were not present, and the module stayed in its return state after returning, calling a module a second time would find its `finish` signal asserted and so the call would be completed immediately, before allowing the module to fully execute. See Figure 4.2 for a pictorial explanation.

An example of this translation can be seen in Listing 3. We have added the `externctrl` map in the textual representation of the HTL code.

```c
int add(int a, int b) {
    return a + b;
}

int main() {
    int v = 0;
    v = add(v, 1);
    v = add(v, 2);
    return v;
}
```

(a) C code

```
add(x2, x1) {
    2:  x3 = x2 + x1 + 0 (int)
    1:  return x3
}

main() {
    9:  x3 = 0
    8:  x6 = 1
    7:  x1 = "add"(x3, x6)
    6:  x3 = x1
    5:  x5 = 2
    4:  x2 = "add"(x3, x5)
    3:  x3 = x2
    2:  x4 = x3
    1:  return x4
}
```

(b) RTL code

```
add(a, b) {
  externctrl { clk -> main.clk }
  controllogic {
    2:  reg_4 <= 1;
    1:  reg_4 <= 3;
    3:  ;
  }
  datapath {
    2:  reg_3 <= {{a + b} + 0};
    1:  finish = 1; return = reg_3;
    3:  finish <= 0;
  }
}

main() {
  externctrl {
    add_1_a -> add.param_0; add_1_b -> add.param_1;
    add_1_finish -> add.finish; add_1_rst -> add.rst;
    add_1_return -> add.return;

    add_0_a -> add.param_0; add_0_b -> add.param_1;
    add_0_finish -> add.finish; add_0_rst -> add.rst;
    add_0_return -> add.return;

    clk -> main.clk;
    }
  controllogic {
      9:  reg_7 <= 8;
      8:  reg_7 <= 7;
      7:  reg_7 <= 12;
     12:  if ({add_0_finish == 1}) reg_7 <= 6;
      6:  reg_7 <= 5;
      5:  reg_7 <= 4;
      4:  reg_7 <= 10;
     10:  if ({add_1_finish == 1}) reg_7 <= 3;
      3:  reg_7 <= 2;
      2:  reg_7 <= 1;
      1:  reg_7 <= 11;
     11:  ;
  }
  datapath {
      9:  reg_3 <= 0;
      8:  reg_6 <= 1;
      7:  add_0_rst <= 1; add_0_a <= reg_3; add_0_b <= reg_6;
     12:  add_0_rst <= 0; reg_1 <= add_0_return;
      6:  reg_3 <= reg_1;
      5:  reg_5 <= 2;
      4:  add_1_rst <= 1; add_1_a <= reg_3; add_1_b <= reg_5;
     10:  add_1_rst <= 0; reg_2 <= add_1_return;
      3:  reg_3 <= reg_2;
      2:  reg_4 <= reg_3;
      1:  finish = 1; return = reg_4;
     11:  finish <= 0;
    }
}
```

Idle State

(c) HTL code

Listing 3: **Translation from C to RTL to HTL.** Matching colours indicate sections of code which translate to each-other

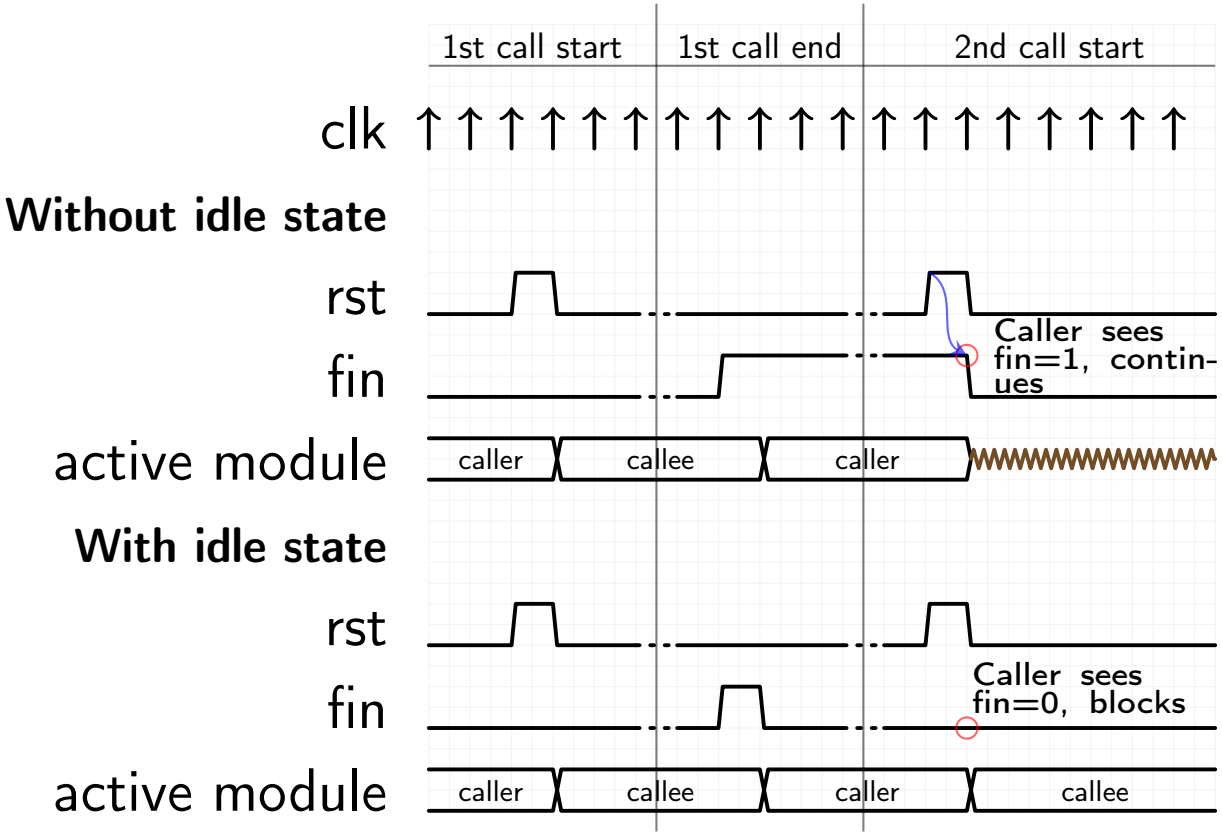

Figure 4.2.: Timing diagram comparison of call with idle state versus without

4.3. Register renaming

In preparation for the next and final phase, Verilog generation, we need to ensure that register names are globally unique. The Verilog output includes the code for all HTL modules in a single Verilog module and so we need to rename registers to avoid any conflicts. In addition, the `externctrl` map present in HTL does not have a direct equivalent in Verilog. It is therefore necessary to eliminate it before or during the Verilog translation. This will be achieved by renaming registers which are tied together through the externctrl map to the same name in the final Verilog. These two problems can be solved with two very similar but distinct passes, performing two different kinds of register renaming.

It is important to note here that, in Vericert, (and also in CompCert more generally) register names are simply positive integers. This simplifies many different operations as numbers are simpler to manipulate and reason about than a different representation such as strings. It is also important for these compiler passes.

The first renaming pass ensures global uniqueness of register names by renaming all registers to fresh numbers. The first register in the first module in the program is renamed to 1, and so are all subsequent instances of that register in the same module, the second

register and all its instances to 2 and so on. When one module is done, the same process is followed for the next module, beginning from the *successor of the highest register number used in the last module*. Repeating this process ensures that there are no register names shared between modules.

Subsequently, we want to re-introduce some "conflicts" between modules: Those described by the `externctrl` map. This is done by the next renaming pass. Here, we rename registers which appear in their module's `externctrl` map to have the same name as the register they are associated to in the map. So if, for example module `A`'s `externctrl` map contains the entry $10 \mapsto (\texttt{B}, \texttt{reset})$, and `B`'s reset register is 20, all instances of register 10 in module `A` will be renamed to `20`. This pass means that in the next phase, Verilog generation, when all HTL modules become part of the same Verilog module the `externctrl` map can be simply discarded, and the registers inside it will be directly tied to the modules that they are meant to control.

4.4. Verilog generation

And so we arrive at the last step in Vericert's backend. This stage required (conceptually) minimal changes from how it functioned before the implementation of this project. We nevertheless present this pass in its entirety for the sake of completeness.

Let us first consider the simple case of a single HTL module, with an empty `externctrl` map, i.e. one generated from a C function containing no calls. For the purposes of this example we will also disregard pointers, as they complicate this process, and are not significant for this project. An example of such a module and its Verilog translation is given in Figure 4.3. The code present inside the state machine described by the HTL is not altered in any way for Verilog generation. This is possible because the code inside HTL states is in Verilog. What is added by this stage is structure around this state machine required to make it a syntactically valid Verilog program, as well as cover its semantic differences from Verilog.

The state machine described by an HTL program is converted to Verilog as a module containing two `always`-blocks, one each for the control- and datapath. Each block contains a single `case`-statement, branching on the module's state register. The branches of the `case`-statement contain the code for each HTL state on the right hand side, and the state number on the left hand side. In addition, register declarations are copied over as-is from the HTL module. Finally, in the control logic `always`-block we insert code to reset the module state if the reset signal is asserted.

Extrapolating from this to the case including a non-empty `externctrl` map requires one main change: recursively include the Verilog translation of all modules referenced in the map. Because of the renumbering described in Section 4.3 there is no change required to "apply" `externctrl` in any way. Simply placing the Verilog translations of all referenced HTL modules in the same Verilog module will have the effect prescribed by the `externctrl` map.

There is one complication to this translation scheme: register declarations. Since they are part of HTL, and are directly transferred to Verilog, they will be duplicated

```
add(a, b) {
  externctrl { }
  controllogic {
    2:  state <= 1;
    1:  state <= 3;
    3:  ;
  }
  datapath {
    2:  reg_10 <= {{a + b} + 0};
    1:  finish = 1; return = reg_10;
    3:  finish <= 0;
  }
}
```

```
module add(clk, reset, start, a, b, return_val, finish);
  input [0:0] clk, reset, start;
  input [31:0] a, b;
  output reg [0:0] finish;
  output reg [31:0] return_val;

  reg [0:0] state;
  reg [31:0] reg_10;

  always @(posedge clk)
    if ({reset == 1}) begin
      state <= 2;
      finish <= 0;
    end
    else begin
      case (state)
        2: state <= 1;
        1: state <= 3;
        3: ;
        default: ;
      endcase
    end

  always @(posedge clk)
    case (state)
      2: reg_10 <= {{a + b} + 0};
      1: finish = 1; return_val = reg_10;
      3: finish <= 0;
      default: ;
    endcase
endmodule
```

Register declarations. Hidden in HTL listings, but part of HTL

Handling of reset signal

Figure 4.3.: **Verilog translation for a simple HTL module.** Code added by verilog generation in green

for registers present in `externctrl` maps. This is because they are declared both in the module of which they are control registers (as `input` or `output`) and also in the module whose `externctrl` map they appear in (as internal registers). Furthermore, control registers of each module are marked as input or output. These declarations need to be removed for all registers but those belonging to the top-level module. Both these problems can be resolved by removing declarations marked as input or output from included modules. After this step, we have the final Verilog which can be saved to disk and handed off to a synthesis or simulation tool.

Register declarations for all state machines

main Control logic

main Datapath

add Control logic

add Datapath

```
1  module main(start, reset, clk, finish, return_val);
2    input [0:0] clk, reset, start;
3    output reg [0:0] finish;
4    output reg [31:0] return_val;
5
6    reg [0:0] reg_1, add_reset, add_fin;
7    reg [31:0] main_state, add_state, add_a, add_b, add_return_val,
8               reg_10, reg_20, reg_21, reg_25, reg_26, reg_28,
9               reg_30;
10   reg [31:0] add_stack [-1:0];
11   reg [31:0] main_stack [-1:0];
12
13
14    always @(posedge clk)
15     if ({reset == 1}) begin state <= 9; finish <= 0; end
16     else begin
17       case (state)
18         9:  state <= 8; 8: state <= 7; 7: state <= 12;
19         12: if ({add_fin == 1}) state <= 6;
20         6:  state <= 5; 5:  state <= 4; 4:  state <= 10;
21         3:  state <= 2;
22         10: if ({add_fin == 1}) state <= 3;
23         2:  state <= 1; 1:  state <= 11;
24         11: ;
25         default: ;
26       endcase
27    end
28    always @(posedge clk)
29     case (state)
30       9:  reg_21 <= 0;
31       8:  reg_26 <= 1;
32       7:  add_reset <= 1; add_a <= reg_21; add_b <= reg_26;
33       12: add_reset <= 0; reg_28 <= add_return_val;
34       6:  reg_21 <= reg_28;
35       5:  reg_25 <= 2;
36       4:  add_reset <= 1; add_a <= reg_21; add_b <= reg_25;
37       10: add_reset <= 0; reg_30 <= add_return_val;
38       3:  reg_21 <= reg_30;
39       2:  reg_20 <= reg_21;
40       1:  finish = 1; return_val = reg_20;
41       11: finish <= 0;
42       default: ;
43      endcase
44
45    always @(posedge clk)
46     if ({add_reset == 1}) begin add_state <= 2; add_fin <= 0; end
47     else begin
48       case (add_state)
49         2: add_state <= 1; 1: add_state <= 3; 3: ;
50         default: ;
51       endcase
52      end
53    always @(posedge clk)
54     case (add_state)
55       2: reg_10 <= {{add_a + add_b} + 0};
56       1: add_fin = 1; add_return_val = reg_10;
57       3: add_fin <= 0;
58       default: ;
59      endcase
60 endmodule
```

Figure 4.4.: **Verilog translation of Listing 3.** Edited for readability.

4.5. Performance Evaluation

We compare the performance of Vericert-generated code with resource-sharing implemented as described in this report versus without, i.e. in the state on which we began developing our changes. We used a selection of benchmarks from the polybench [17] benchmark suite. We used the benchmark selection originally used to evaluate Vericert in Herklotz et al. in 2020 [8], with the exception of the `adi` and `ludcmp` benchmarks. The former caused a compiler error in both Vericert master and our Vericert, while the latter caused an error only in our compiler[1]. We used Icarus Verilog [19] to simulate the generated designs and get a cycle count measurement. The designs were then synthesised using Xilinx Vivado 2017.1, targeting a Xilinx 7-series FPGA (XC7K70T) with a target frequency of 50MHz. The area usage and f_{max} results reported here are based on the reports generated by Vivado. The detailed data is presented in Appendix A.

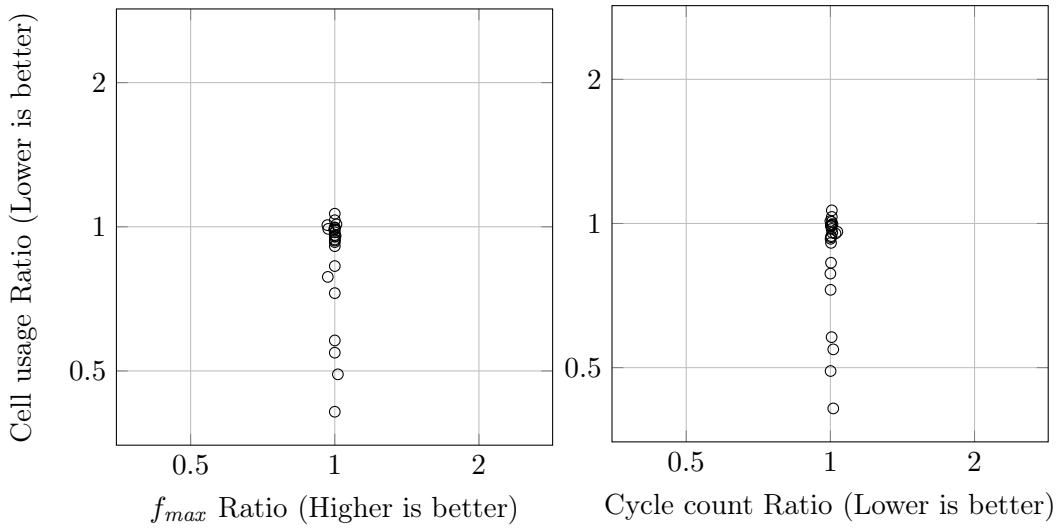

Figure 4.5.: Comparison of change in f_{max} and cycle count against change in cell usage

Figure 4.5 shows a visual summary of the benchmarking data. The two plots compare cell usage, f_{max} and cycle count using the ratio of the value in Vericert with our changes over in Vericert without our changes. We found an improvement in cell usage across almost all benchmarks. The change in cell usage ranged from using 106% of the logic cells in the worst case to just **41%** in the best case averaging out to **87.9%** the area usage overall. Cycle count appears to be mostly unaffected, using 0.7% more clock cycles on average. A modest increase was expected, as the translation involves introducing an extra state into the state machine for every call in the source program. Clock frequency (f_{max}) appears similarly unaffected, ranging from a 1.5% improvement (increase) to a 3.1% worsening (decrease), averaging a 0.2% decrease over all benchmarks.

[1]The reason for this is to be investigated and resolved. This does not affect compiler *correctness*, as no miscompilation was identified

Overall, we consider the project to be a success in terms of the performance-related goals we set in Section 1.1. We have achieved an measurable improvement in area usage, while cycle count and frequency have seen minimal impact.

4.6. Previous attempts

The design presented above is not the one originally planned. There were other paths explored and abandoned early on in the project which are presented briefly here.

It was identified early on in the proof that the main problem to be solved by the project was the lack of a semantics for function calls in any of the "hardware" languages in Vericert, those being HTL, and Vericert's subset of Verilog. As "true" Verilog has no concept of calls, we attempted to augment HTL, adding calls to the language and its semantics. This took the form of three new instructions: `fork`, `join`, and `wait`. `fork` would behave as the datapath of the state implementing the first part of the call process in the current design, while `wait` and `join` would behave as the control- and datapath of the second state in the process, respectively.

`fork` would take as arguments a module identifier to be called, and a list of registers where the arguments to the call are located. It would set the arguments for the identified module and set its reset signal. `wait` would appear in the controlpath and take a module identifier and state number as arguments, waiting for the module to assert its done signal before progressing the state.

We abandoned this approach as it increased complexity in the HTL proof which assumed that HTL statements were "just" Verilog statements. Adapting the proof to support these new instructions was quickly found to be too costly and abandoned.

CHAPTER 5

Correctness proof

Formal proof of the correctness of this project formed the bulk of its workload. This proof follows the general structure of proofs in CompCert, outlined in Section 2.2. As we have made changes throughout all of Vericert's passes, all their associated proofs required adjustments as well. We have *partially* completed the correctness proof for HTL generation, with certain lemmas assumed correct. We give an outline of the proof, including changes made to the semantics of the language, and list our assumptions. The renaming passes and Verilog generation have not been proven correct in any way, but we nonetheless give a possible outline of their possible correctness proofs.

5.1. HTL Semantics

We first present the original semantics for HTL, as used in Vericert and outline the changes we made to support our implementation.

5.1.1. Original

In general, semantics for intermediate languages in CompCert's framework are given in terms of a "state", which is defined separately for each language. A set of *inference rules* are then added, defining how this state evolves as a program in the language executes. For the HTL semantics, this state has three variants:

1. State stk m pc Γ_r Γ_α

   Active while in the middle of a module's execution. m is the currently executing module. *Not its identifier*, but the full datastructure representing the module. Γ_r is the map containing scalar registers' values, while Γ_α serves that role for array registers. pc is the value in the module's state register, denoted σ_m Finally stk is the *callstack*, represented as a list of *stackframes*, meant to contain the states of

$$\frac{\Gamma_r\,!\,fin_m = 1 \qquad \Gamma_r\,!\,ret_m = v}{\text{State } stk \;\; m \; pc \; \Gamma_r \; \Gamma_\alpha \rightarrow \text{Returnstate } stk \;\; v} \text{ Finish}$$

$$\frac{}{\begin{array}{c}\text{Callstate } stk \;\; m \; \vec{r} \rightarrow \\ \text{State } stk \;\; m \; entry_m \; (\text{init\_params } \vec{r} \; a)[rst_m \mapsto 0, \, fin_m \mapsto 0, \, \sigma_m \mapsto entry_m] \; \epsilon\end{array}} \text{ Call}$$

Figure 5.1.: State transition rules in original HTL semantics

modules further up the callstack. This was empty in all proofs originally, as there is only a single module called in any HTL program.

2. Returnstate *stk* *v*

   Entered when a module finishes execution. Holds the stack, which has is the same as for the normal State, and the return value of the module, i.e. the value of the module's return register when it finished execution.

3. Callstate *stk* *m* $\vec{r}$

   State used to initiate a call to a module. In the original HTL semantics this is only used for the initial state, which calls the main function. $\vec{r}$ denotes the values of the arguments to the call which are given to the parameter registers, denoted $\vec{p}$, with $\vec{p}_n$ being the register of the nth parameter.

The inference rules for the original state transitions are given in Figure 5.1. The "Call" and "Finish" rules are only ever used to call the main function and return from it, respectively. This is why there are no Callstate → State or Returnstate → State rules given. The State → State rule reuses the Verilog semantics for execution of statements, and executes the control logic statement at the program counter, followed by the datapath statement at the program counter, correctly handling blocking and non-blocking assignments, and ensuring that the control registers have the appropriate values to allow execution.

5.1.2. Augmented

Our augmented semantics for HTL to support calls required the addition of one parameter to all three state variants: the identifier of the current module. We also implement two new state transitions: "InitCall", transitioning from State to Callstate, and "Return", allowing us to step from Returnstate back to State. Finally, we augment the step relation itself with the "global environment" g: A map from module identifiers to modules. Such a map is keeping with CompCert practice, as it is also used for the semantics of RTL. These new rules have also required the use of the stack, the former creating a new stackframe and the latter consuming it. The parameters of Stackframe are identical to those of State, excluding only *stk* (since a stackframe does not have its own stack).

$$\frac{\begin{array}{c}\texttt{externctrl}_m\,!\,rst = (id_w, \textbf{reset}) \rightarrow \Gamma_r\,!\,rst = 0 \\ \forall n < |\vec{p}|.\, \texttt{externctrl}_m\,!\,\vec{p}_n = (id_w, \textbf{param}_n) \quad \forall n < |\vec{p}|.\, \vec{r}_n = \Gamma_r\,!\,\vec{p}_n \\ g\,!\,id_w = w\end{array}}{\begin{array}{c}\text{State } stk\ id_m\ m\ pc\ \Gamma_r\ \Gamma_\alpha \xrightarrow{g} \\ \text{Callstate } ((\text{Stackframe } id_m\ m\ pc\ \Gamma_r\ \Gamma_\alpha) :: stk)\ id_w\ w\ \vec{r}\end{array}}\ \text{InitCall}$$

$$\frac{\begin{array}{c}\texttt{externctrl}_m\,!\,rtrn = (id_w, \textbf{return}) \\ \texttt{externctrl}_m\,!\,fin = (id_w, \textbf{finish})\end{array}}{\begin{array}{c}\text{Returnstate } (\text{Stackframe } id_m\ m_m\ pc\ \Gamma_r\ \Gamma_\alpha :: stk)\ id_w\ v \xrightarrow{g} \\ \text{State } stk\ id_m\ m_m\ pc\ \Gamma_r[\sigma_m \mapsto pc, fin \mapsto 1, rtrn \mapsto v]\ \Gamma_\alpha\end{array}}\ \text{Return}$$

Figure 5.2.: Added/altered state transitions for HTL

InitCall requires that a register rst, which is mapped to the reset signal of some "callee" module through the `externctrl` map, is set to 0. If that condition is met, the rule can be applied. We find a list of registers $\vec{p}$ that are mapped to parameters of the callee and use their values to form $\vec{r}$, the arguments that are used in the call. Finally, we look up the name of the callee in the global environment to find the data of the callee module (m_{callee}). After creating a new Stackframe using the parameters of the calling State we have all the necessary data for the new Callstate.

Return pops the top stackframe off the callstack and uses the data from it to re-create the State at which the state was created (i.e. the state at which the last call took place). In addition, it requires that the "caller" has registers mapped to the "return" and "finish" registers of the "callee" in its `externctrl` map. Those registers are set to 1 and the return value, respectively.

A "necessary evil" had to be allowed into the formulation of these semantics: non-determinism. The InitCall rule *can* be triggered when an `externctrl`-mapped reset register is set to 0, but it does not *have* to be. The Step rule would be equally valid at that state. We explain how this non-determinism was "tamed" for the purposes of the HTL generation correctness proof, as well as how it would affect the Verilog generation correctness proof.

5.2. RTL to HTL translation

The goal of this proof is to prove that the output of the function has semantics which match those of the input. This can be broken down as follows:

1. Find a specification of the translation function which describes its proof-relevant aspects.

2. Prove that the translation fulfils its specification.

3. Describe what it means for a state in the source language (RTL) to match a state in the target language (HTL). This includes the fact that the programs the states relate to are related by the translation spec.

4. Prove that if some RTL state S steps to some RTL state S' in the RTL semantics, and S matches some HTL state R, then there exists an HTL state R', such that R steps to R' in one or more steps and R' matches S'.

$$\forall R, S, S'.\, (S \underset{RTL}{\Longrightarrow} S') \rightarrow S \; match \; R \rightarrow \exists R'.\, (R \underset{HTL}{\Longrightarrow}^{*} R') \wedge (S' \; match \; R')$$

   This is called a *semantic preservation* proof using *forward simulation*

Here, we describe the translation specification and state matching statements, and outline the semantic preservation proof.

5.2.1. Translation specification

The specification, at the level of instructions, in base vericert has the form

$$\text{tr\_code}\ c_{RTL}\ pc\ data\ ctrl\ fin\ rtrn\ \sigma\ stk$$

This predicate says that the RTL statement at pc in the RTL code c_{RTL} is translated into the datapath and control statements at pc in $data$ and $ctrl$, respectively. In Vericert, this just wraps the predicate spec_instr (detailed in Herklotz et al. 2020 [8] page 8), which relates an RTL instruction to a control and a datapath HTL statement.

We extend tr_code by adding two special cases, for the translation of the `call` and `return` RTL instructions. The translation of these two instructions differs from that of all other instructions: They each create a new state, as well as new registers, i.e. ones not present in the RTL code. We solve both these problems by require that there exist *some* HTL state in the control- and datapaths matching the specification of the added state with *some* registers

5.2.2. Matching states

The match relation is extended from Vericert master. We extend the matching of call-stacks, adding a rule matching an RTL to an HTL stack frame, where Vericert master only allows two empty lists of stack frames to match. This rule duplicates the requirements for the matching of states, reflecting how stack frames duplicate the data contained in a State. Furthermore, we also require that every module on the stack contains appropriate registers in its `externctrl` map for the module in the stackframe above it, as well as the one at the top of the stack containing them for the currently executing module. For example, if f calls g then we require that module f have `externctrl` registers for g's $reset$, $finish$, etc. signals. Finally, we require that the control- and datapath code at the return σ value (the "return address") of every stackframe contain a specific code fragment, (that of the "join" state) which makes sure that the result is copied from the `externctrl` register to the destination register used in the RTL code.

5.2.3. Semantic preservation proof

The proof is by induction on the RTL step relation. Each case of this proof is packaged in it own lemma, as some are quite lengthy. Each lemma essentially replicates the top-level semantic preservation statement, specialised to a specific instance of the RTL step relation.

Existing lemmas, for the most part, required minor adjustments to cover the new requirements imposed on the state matching relation. Most could be proven by reusing the proof from the starting state in the resulting state, as the code in question did not affect the property. In other cases, however, changes were more significant. For the proofs relating to the `load` and `store` instructions, certain assumptions had to be taken. The original proof used the fact that the stack in HTL was always empty, i.e. there were no stackframes. This was valid because the only function ever called was the main. This is no longer the case as a function call structure now exists in HTL. This change could indeed be the cause of bugs, were it not for the fact that our implementation handles this. The *inlining pass* makes this scenario impossible; functions performing loads or stores are inlined, meaning that the `load` and `store` instructions should only ever appear in the main function. This fact, however, has not been proven correct and is therefore taken as an assumption. The reasons for this are outlined in Subsection 5.2.4.

There were three new lemmas required to match transitions in RTL which were previously impossible in HTL.

1. State → Callstate
2. Callstate → State
3. Returnstate → State

In addition, the lemma relating to the State → Returnstate transition had to be proven from scratch, even though its statement remained the same.

We give the statement of each lemma and an outline of each proof, stating where assumptions have been made or proofs are still incomplete.

Lemma 5.1 (`transl_icall_correct`). *The semantics of the State → Callstate transition in RTL are preserved by the HTL translation.*

Proof sketch. This step corresponds to the first part of the execution of the RTL `call` instruction. The translation specification dictates that the HTL state matching an RTL call instruction is a "fork" state, set to transition into a "join" state. The initial RTL state of this transition, then, matches to an HTL the "fork" state, which sets the arguments' values and asserts the module's reset signal (as always using registers in `externctrl`). An HTL State → State transition is then applied, moving execution to the next state, a "join", which de-asserts the reset signal. At this point, the HTL State → Callstate transition can be applied. The callstate that we transition to can be shown to match the RTL Callstate of the lemma. The execution has been proven correct, but the matching between the RTL and HTL states has only *partially* been proven correct. □

Lemma 5.2 (`transl_callstate_correct`). *The semantics of the Callstate → State transition in RTL are preserved by the HTL translation.*

Proof sketch. The initial RTL Callstate directly matches an HTL Callstate, which single-steps to an HTL State. This state is then shown to match the target RTL State.

The execution has been proven correct, but the matching between the RTL and HTL states has only *partially* been proven correct. For the part of the state matching which we proved correct we made used of the `no_stack_calls` admitted lemma. □

Lemma 5.3 (`transl_returnstate_correct`). *The semantics of the Returnstate → State transition in RTL are preserved by the HTL translation.*

Proof sketch. The initial state of this transition matches an HTL Returnstate. We need to perform two steps to find an HTL state matching the final RTL state. First, from the Returnstate, we enter the calling module, using the state register (σ) value specified at the top stackframe. From the stackframe matching relation we know that at this σ value there is a "join" state, set to transition to a state matching the return address of the RTL stackframe. From here, we can apply the State → State transition to execute this "join" state. This copies the called function's result from the `externctrl` register to the register which appears in the final RTL state, and as we established, transitions to the state matching the RTL return address.

This lemma is fully proven correct, but makes use of the `no_pointer_return` and `no_stack_functions` assumptions which are detailed below. □

Lemma 5.4. *The semantics of the State → Returnstate transition in RTL are preserved by the HTL translation.*

Proof sketch. This proof corresponds to the execution of the `return` RTL instruction, which maps to two states in HTL: One setting the module result and `finish` signal (the "return" state) and the "idle" state. The RTL transition is matched by two HTL transitions: one State → State transition, executing the "return" state, and a State → Returnstate transition, allowed because of the "return" state setting the module `finish` signal. The "idle state" is not invoked in this proof, as it does not match any part of the RTL semantics. It is required strictly for the resulting Verilog.

This lemma is fully proven correct and did not require the use of any assumptions. □

5.2.4. Assumptions

For the purposes of the correctness proof of the RTL to HTL pass, we had to take certain assumptions. These are listed here, along with the reason for which they are required, their *consequences* in terms of correctness (i.e. could this assumption be violated? If so what could go wrong?), and what steps could be taken to remove it. This section contains an exhaustive list of the assumptions in correctness proof of the RTL to HTL pass. These assumptions are in addition to the lemmata stated as not entirely proven above.

`only_main_stores`

We assume that, for all RTL states matching an HTL state, if the current instruction is a `store`, then the callstack is empty, i.e. the currently executing function is at the top of the callstack. This assumption is founded on the inlining pass. Since we inline all functions performing loads or stores *before* the RTL to HTL translation, the only place where a `store` instruction could be encountered is in the RTL main function. This assumption is used for the correctness proof of the `store` instruction translation, in order to prove that the translation of that instruction preserves the matching of stackframes.

We believe this assumption to be valid, and that it would be provable given more time.

`no_stack_functions`

`no_stack_calls`

These two assumptions are closely related to each other and `only_main_stores`. Together, they state that if an RTL state matches an HTL state, then the function currently executing (in the case of `no_stack_functions`) or being called (in the case of `no_stack_calls`) must either have a stack size of zero or be at the top of the callstack, i.e. be the main function. These two are used in the proofs relating to the `call` and `return` RTL instructions. They are used in conjunction with the following two assumptions.

`mem_free_zero`

`mem_alloc_zero`

These are to do with Memory in RTL semantics. They state that allocating or freeing an empty block of memory leaves the representation of memory unchanged. These lemmas are *not* provable, due to the way in which the free and allocate functions work for RTL. They are however, functionally correct. While allocating or freeing a block of memory does not leave the memory unchanged, it does leave it in a functionally identical state, which should be provable and could be used to draw the same conclusions as these lemmas have been used for in our proof.

`no_pointer_return` and `no_pointer_call`

This assumption is required due to a fact used throughout HTL, which departs from the rest of CompCert. In CompCert semantics, in general, pointers are represented as a base and offset. The base can be the stack pointer for pointers to function-local addresses, or the beginning of an allocated block for dynamically-allocated memory. In HTL semantics, however, pointers are represented *only* by the offset. This is because in HTL there is only a single memory block, and all pointers can be seen as being based at its start.

The HTL generation proof relies on all pointers in the input RTL having the stack pointer of the currently executing function as their base address. This is possible because calls were previously never translated. There were, therefore, no pointers returned from functions, and so all pointers were ones based on the stack pointer of the main function.

This has now changed. Functions can be called and so pointers can originate outside the currently executing function, having a base pointer other than the current stack pointer. This conflicts with the previously held assumption.

Our work-around to this problem was to take the `no_pointer_call` and `no_pointer_return` assumptions. These essentially state that no function is called with a pointer argument, or returns a pointer, respectively. This eliminates the problem, since pointers are once again assumed to never cross function boundaries, but is obviously wrong: Functions *can* return pointers. This conflict means that the formal proof does not guarantee that accesses on pointers which have been returned from functions will be valid. For example, if a function were used to perform arithmetic on a pointer, and the resulting pointer were dereferenced, our formal proof does not guarantee that the semantics of that access will be preserved past HTL generation, and therefore in the resulting hardware. Note, however, that a function returning a pointer to a local variable, which is then dereferenced, is *undefined behaviour*. CompCert, much like any other C compiler, makes no guarantees about programs exhibiting undefined behaviour, and so this scenario need not be considered.

Because of the way in which function returns are translated to HTL, we believe this assumption to be highly unlikely to result in a bug. The HTL translation of an RTL return simply copies the returned register to the result register of the module, without performing any alterations. Indeed, basic testing on simple functions performing pointer arithmetic, showed that Vericert-generated Verilog behaved correctly. Nevertheless, this is a gap in the proof and should be eliminated in order to consider this pass (and Vericert as a whole) verified.

Eliminating these two assumptions could be done in a few different ways. The most direct, easiest route, would be to change the assumption in the proof. Instead of requiring that all pointers are based on the current function's stack pointer, we could require that they are all based on the stack pointer of the function at the bottom of the callstack, i.e. the main. This is equivalent to the assumption in Vericert master, since it only ever had one function on the callstack, however, it would generalise to our extended Vericert. The more principled fix would be to get rid of the need to inline functions performing memory accesses altogether. The semantics of memory accesses in HTL would then be closer to those of RTL and this assumption could then be removed completely. This is a much more radical change, which we discuss in Section 6.2.

5.3. HTL renaming

The two HTL renaming passes described in Chapter 3 are, as of the writing of this report, unverified. We lay out a roadmap for their correctness proof.

For the two passes we will need to prove two theorems: The first is the regular semantic preservation proof that is required of any pass in CompCert or Vericert. This is to ensure that neither pass alters the behaviour of a program with respect to HTL semantics. The second is a theorem stating that the two passes, together, perform their intended function. That is, that they ensure that the only register names shared across HTL modules are

those intended to be shared through the `externctrl` map. This will be necessary for the subsequent correctness proof of the HTL to Verilog translation.

For the semantic preservation proof, we need to establish how we match the state of the original program to that of its renamed version. This would essentially amount to applying the renaming mapping to the execution state. For example, say we know the renaming applied the mapping $[1 \mapsto 10; 2 \mapsto 20]$, meaning that register 1 has been renamed to register 10 and register 2 has been renamed to register 20. Using this mapping, we would then say that the state $\{1 : \texttt{`a'};\ 2 : \texttt{`b'}\}$ in the original program matches the state $\{10 : \texttt{`a'};\ 2 : \texttt{`b'}\}$ in the renamed program. That is, we expect register 10 to have the value of register 1 in the old program and register 20 to have the value of register 2 in the old program. We would then proceed to prove that this matching of states is preserved through all transitions in the HTL semantics by forward simulation. This planned proof would also require that we augment both passes to output not only the transformed program, but also the mapping they applied.

The second proof would be split between the two renaming passes. First we establish that the global renaming pass does indeed make register names globally unique. Then, using that fact, and establishing that the `externctrl` application pass only changes the names of registers appearing in the `externctrl` map, we could show that, after both passes, we have a program were all register names are `externctrl` map.

5.4. HTL to Verilog translation

The HTL to Verilog translation was previously proven correct. This was a simple proof, compared to most others in CompCert or Vericert. This is because the semantics of HTL and Verilog are almost a one-to-one match. It was further simplified by the fact that the Verilog generation uses the entire HTL program as-is and simply adds structure around it (see Section 4.4). The proof therefore, needed only to address the parts of HTL which were implicit in its semantics but were made syntactically explicit in Verilog, for example, the code added to handle the reset signal.

We expect a large section of this proof to still be usable, however a significant amount of work will need to be added. We expect most of the complexity to be due to the addition of function call semantics in HTL which have no analogue in Verilog. It will, be necessary to be precise in how the state matching relation is defined, so that a calling state in HTL can match an appropriate state in Verilog. As discussed in the previous section, we also expect to explicitly have to make use of the correctness of the renaming passes in proving the semantics of calls correct.

CHAPTER 6

Conclusion and Future Work

6.1. Completing the proof

The project has achieved its implementation goals, providing an improvement in the area usage of generated hardware, for only a consistently small worsening of cycle count and clock frequency. The main limitation of the project is, however, in its proof. It would have to be expanded to cover all aspects of the compiler before the project can be considered fully verified.

The first step to this would be completing the HTL generation proof. This would mean removing or proving correct any assumptions and completing the proofs of lemmata left incomplete. The two renaming passes and Verilog generation would also have to be proven correct as outlined in Sections 5.3 and 5.4.

6.2. Expanding resource sharing to array-based functions

The inclusion of the inlining pass could also be considered a limitation. Vericert is currently unable to share the hardware of functions which perform any kind of memory access, which is what made the inlining pass necessary. Removing this limitation would allow for further improvements in area usage, however the impact on cycle count and clock frequency would depend on the specific implementation and remains to be seen.

6.3. Enabling concurrent execution of functions

The way in which resource sharing has been implemented should enable concurrent execution of functions. Currently, calls in C map to transfer of control between state machines in the resulting hardware, where only one state machine is ever running at a time. This

could be altered. Currently calls happen using two states, where one sets the call up, setting the arguments and resetting the called module, and the next blocks while the called module executes. These states are currently always created to be sequential. Instead, the initiating state could be transferred "up" to where all the arguments are available, and another call to this function is not also active, while the waiting state could be transferred "down", to right before the first point where its result is needed. This way the execution of the called function would be interleaved with that of the caller, and the waiting state could find the result immediately available, cutting down on overall cycle count.

Bibliography

[1] Matthew Aubury et al. "Handel-C language reference guide". In: *Computing Laboratory. Oxford University, UK* 12 (1996).

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science An EATCS Series. Springer Berlin Heidelberg, 2004, nil. DOI: 10.1007/978-3-662-07964-5. URL: https://doi.org/10.1007/978-3-662-07964-5.

[3] Andrew Canis et al. "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. Monterey, CA, USA: Association for Computing Machinery, 2011, 33–36. ISBN: 9781450305549. DOI: 10.1145/1950413.1950423. URL: https://doi.org/10.1145/1950413.1950423.

[4] R. Chapman, G. Brown, and M. Leeser. "Verified high-level synthesis in BEDROC". In: *[1992] Proceedings The European Conference on Design Automation*. 1992, pp. 59–63. DOI: 10.1109/EDAC.1992.205894.

[5] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design Test of Computers* 26.4 (2009), pp. 8–17. ISSN: 1558-1918. DOI: 10.1109/MDT.2009.69.

[6] Zewei Du et al. "Fuzzing High-Level Synthesis Tools". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. Virtual Event, USA: Association for Computing Machinery, 2021, p. 148. ISBN: 9781450382182. DOI: 10.1145/3431920.3439466. URL: https://doi.org/10.1145/3431920.3439466.

[7] Dan Gajski, Todd Austin, and Steve Svoboda. "What Input-Language is the Best Choice for High Level Synthesis (HLS)?" In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: Association for Computing Machinery, 2010, 857–858. ISBN: 9781450300025. DOI: 10.1145/1837274.1837489. URL: https://doi.org/10.1145/1837274.1837489.

[8] Yann Herklotz et al. “Formal Verification of High-Level Synthesis”. 2020. URL: https://yannherklotz.com/docs/drafts/formal_hls.pdf (visited on 01/30/2021).

[9] William A Howard. “The formulae-as-types notion of construction”. In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[10] E. Hwang, F. Vahid, and Yu-Chin Hsu. “FSMD functional partitioning for low power”. In: *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*. 1999, pp. 22–28. DOI: 10.1109/DATE.1999.761092.

[11] “IEEE Standard for Verilog Hardware Description Language”. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[12] *Intel High Level Synthesis Compiler*. 2021. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[13] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Commun. ACM* 52.7 (July 2009), 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[14] A. Lööw and M. O. Myreen. “A Proof-Producing Translator for Verilog Development in HOL”. In: *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 2019, pp. 99–108. DOI: 10.1109/FormaliSE.2019.00020.

[15] Juan Perna and Jim Woodcock. “Mechanised wire-wise verification of Handel-C synthesis”. In: *Science of Computer Programming* 77.4 (2012). Brazilian Symposium on Formal Methods (SBMF 2008), pp. 424 –443. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2010.02.007. URL: http://www.sciencedirect.com/science/article/pii/S0167642310000341.

[16] Juan Perna et al. “Correct hardware synthesis”. In: *Acta informatica* 48.7-8 (2011), pp. 363–396.

[17] Louis-Noël Pouchet. *PolyBench/C. the Polyhedral Benchmark suite*. 2020. URL: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[18] *Vitis HLS*. 2021. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[19] Stephen Williams. *Icarus Verilog*. http://iverilog.icarus.com/. Version 11.0.

APPENDIX A

Benchmarking data

Benchmark	Ratio of ours over original		
	Cell usage	Cycle count	f_{max}
2mm	0.988	1.004	1.000
3mm	0.579	1.006	1.000
atas	0.961	1.034	1.000
bicg	0.546	1.014	1.000
cholesky	0.786	1.000	0.967
covariance	0.990	1.002	0.969
doitgen	0.828	1.003	1.000
durbin	0.929	1.001	1.000
fdtd-2d	0.727	1.001	1.000
floyd-warshall	0.492	1.001	1.015
gemm	0.994	1.007	1.000
gemver	0.957	1.009	1.004
gesummv	0.953	1.025	1.000
heat-3d	1.065	1.007	1.000
jacobi-1d	0.911	1.003	1.000
jacobi-2d	0.936	1.003	1.000
mvt	0.411	1.014	1.000
nussinov	1.012	1.000	1.008
seidel-2d	0.941	1.004	1.000
symm	0.987	1.005	0.997
syr2k	1.032	1.007	1.000
syrk	0.997	1.010	1.000
trisolv	0.974	1.006	1.000
trmm	1.008	1.006	0.965
Geometric mean	0.879	1.007	0.998

Table A.1.: Evaluation results for polybench