Performance Aspects of Correctness-oriented Synthesis Flows

Fritjof Bornebusch[1], Christoph Lüth[1,2], Robert Wille[1,3,4], Rolf Drechsler[1,2]
[1] *Cyber-Physical Systems, DFKI GmbH, Bremen, Germany*
[2] *Mathematics and Computer Science, University of Bremen, Germany*
[3] *Integrated Circuit and System Design, Johannes Kepler University Linz, Austria*
[4] *Software Competence Center Hagenberg GmbH (SCCH), Hagenberg, Austria*
{*fritjof.bornebusch,christoph.lueth*}*@dfki.de, robert.wille@jku.at, drechsler@uni-bremen.de*

Keywords:
Hardware Designs, Proof Assistants, Functional HDLs, Hardware Synthesis, MIPS processor

Abstract:
When designing electronic circuits, available synthesis flows either focus on accelerating the synthesized circuit or correctness. In the quest for ever-faster hardware designs, the correctness of these designs is often neglected. Thus, designers need to trade-off between correctness and performance. The question is how large the trade-off is? This work presents a systematic comparison of two representative synthesis flows, the LegUp HLS framework as a representative for flows focusing on hardware acceleration, and a flow based on the proof assistant Coq focusing on correctness. For evaluation purposes, a 32-bit MIPS processor synthesized using the two flows, and the final HDL implementations are compared regarding their performance. Our evaluation allows a quantitative analysis of the trade-off, showing that correctness-oriented synthesis flows are competitive concerning performance.

1 Introduction

Electronic circuits have become more and more complex over time. The goal of synthesis flows is either to synthesize accelerated circuits which have a high performance or correct ones which guarantee correctness properties. As synthesis flows with an emphasis on acceleration often do not provide the ability to formulate correctness proofs, these design flows are a severe issue when applied in safety-critical systems such as cars, airplanes, or medical devices. This comparison leads to the question of whether both flows can be combined to get the best of both worlds.

To address this question, we first take a look at synthesis flows with an emphasis on acceleration. To tackle the synthesis of faster hardware designs, synthesis flows like Bambu [29], DWARV [27], or LegUp [8, 9] evolved (in the following called *acceleration-oriented synthesis flows*). These flows start with a model written in a Domain-specific language (DSL) to describe hardware designs that are embedded into the C programming language. After the model is implemented, it is synthesized into a low-level implementation in a hardware description language (HDL) at the Register-Transfer-Level (RTL), e.g., Verilog. During the automatic synthesis process, different optimizations like loop or functional pipelining [8, 22] are performed to accelerate the final implementation.

One problem with these synthesis flows is the missing definition of a synthesis scheme [15, 2] and the resulting lack of property verification. In general, it is unclear (1) how the implementation is generated from the model in detail; (2) whether the semantics of the model are correctly represented by the semantics of the implementation; and (3) how to track and verify properties stated at the specification level in the implementation.

In contrast, synthesis flows with an emphasis on verification like Kami [11] or the one based on Coq [4] and CλaSH [3] as introduced in [5] start with a specification in a formal language that allows the verification of functional properties about the hardware design (in the following called *correctness-oriented synthesis flows*). After the specified behavior was verified, an RTL implementation is synthesized automatically. This way, these flows guarantee a correct transformation of the semantics of the specification to the final implementation and, hence, ensure the verified properties hold on all levels.

However, while these approaches can guarantee correctness, it remains unclear how the performance of the resulting designs compares to the performance of designs obtained by the acceleration-oriented synthesis flows reviewed above. In fact, it is intuitive to assume that a focus on verification may harm this performance. But unfortunately, this possible trade-off has not been addressed in detail yet. While anecdotal evidence suggests correctness-oriented flows can be competitive with respect to performance, we present a systematic analysis by comparing the design of a non-trivial circuit with two representative flows from each camp.

The missing trade-off leaves designers with the question of whether they should focus on correctness (motivating the utilization of a correctness-oriented synthesis flow) or on performance (motivating the utilization of an acceleration-oriented synthesis flow).

This paper addresses this question. To this end, we investigate both design flow paradigms – using the LegUp high-level synthesis (HLS) framework [8, 9] and the synthesis flow from [5] as a representative for acceleration-oriented and correctness-oriented synthesis, respectively. We chose those flows as they represent the most efficient (cf. [26]) and most recent flows available thus far, implementing the respective concepts. The foundation of the investigation is a 32-bit MIPS processor [20], which is synthesizable by LegUp. The functional behavior of this processor is specified, verified, and synthesized using the correctness-oriented flow, described above.

Our quantitative analysis of the processor implementations gives a first impression to gauge the trade-off between performance and correctness. Even though there will be cases that justify the application of the acceleration-oriented flows, our analysis shows the potential of further research of applying correctness-oriented flows in an industrial setting, even in cases where performance is a critical issue. Moreover, it is easier to increase the performance of circuits synthesized by correctness-oriented flows than to make hardware designs following acceleration-oriented flows correct.

We do not discuss whether an acceleration-oriented model or a correctness-oriented specification is more user-friendly as this question is too subjective to answer.

This work is structured as follows: first, we motivate our work by describing and discussing the LegUp synthesis flow and the considered problem we address in this work. Section 3 describes the correctness-oriented synthesis flow in detail and how it addresses the considered problem. Section 4 describes our specification of the processor and how properties are verified. Section 5 evaluates and discusses the RTL implementations. Finally, Section 6 summarizes this work.

2 Motivation

This section analyzes the LegUp HLS framework synthesis flow [8] as a representative of a contemporary, state-of-the-art acceleration-oriented synthesis flow. On average, LegUp synthesizes the fastest hardware designs, which is the reason for picking it as a representative [26]. This flow analysis shows the missing ability to verify correctness properties of models implemented for these flows.

An available model of a 32-bit MIPS processor is used as a running example to analyze the synthesis flow implemented by the LegUp HLS framework [20, 8]. The MIPS architecture describes an instruction set architecture (ISA) for a reduced instruction set computer (RISC) [25].

2.1 The LegUp Synthesis Flow

The foundation of the LegUp framework is the LLVM (Low Level Virtual Machine) compiler infrastructure [24]. LLVM is a modular compiler infrastructure for optimized code generation. A model is transformed into LLVMs internal intermediate representation, which is a machine-independent assembly language using the Clang compiler front-end and later optimized by a series of built-in compiler optimizations. LegUp extends the LLVM backend by generating Verilog code instead of Machine code [8]. In order to accelerate hardware designs, different additional optimizations are performed during the optimization process by LegUp, e.g., loop or functional pipelining [8, 22].

These optimizations aim to identify behavior that can be accelerated. Whether an opti-

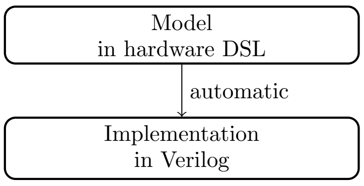

Figure 1: Sketched LegUp synthesis flow for pure hardware designs. A model in a hardware DSL is synthesized into an accelerated low-level implementation in Verilog automatically.

mization can be performed and , therefore, result in an implementation that satisfies the required performance properties depends on the used model.

For modeling designs, LegUp defines a Domain-specific language embedded into the C programming language. Two modes are provided to synthesize a model. The first mode is the generation of a hybrid processor/accelerator architecture. The described behavior of a model is compiled and executed on a dedicated processor that profiles its execution. After profiling, segments of the model are selected that are accelerated by hardware implementations. The final part is re-compiling the model into a hybrid hardware/software system (Hardware/Software Codesign [18]).

The second mode is the automatic synthesis of a model in a pure and accelerated RTL implementation, sketched in Figure 1. After the implementation is generated, it can be synthesized on an FPGA using commercial synthesis tools. In contrast to the first mode, constructs like dynamic memory management, recursion, and floating-point arithmetic are not supported [8].

In this paper, we focus on the second mode. Since the running example used in this work describes a 32-bit MIPS processor, the model is synthesized to pure hardware.

Example 1. *In order to analyze the LegUp synthesis flow regarding the verification of properties, we consider a 32-bit MIPS processor implementation. This implementation is already the subject of current research [20, 26] and is sketched in Listing 1. The model implements a subset of the 32-bit MIPS standard instruction set, roughly 40 instructions. It also provides an implementation of a program that is a set of bitvectors and follows the bit order for instructions stated by the 32-bit MIPS instruction specification [25].*

According to the program counter, each instruction is processed in one iteration and is read

```
ins = imem[IADDR (pc)];
op = ins >> 26;
switch (op) {
 case R:
  funct = ins & 0x3f;
  shamt = (ins >> 6) & 0x1f;
  rd = (ins >> 11) & 0x1f;
  rt = (ins >> 16) & 0x1f;
  rs = (ins >> 21) & 0x1f;
  switch (funct) {
   case ADDU:
    reg[rd] = reg[rs] + reg[rt];
    break;
   case SLL:
    reg[rd] = reg[rt] << shamt;
    break;
[...]
  address = ins & 0xffff;
  rt = (ins >> 16) & 0x1f;
  rs = (ins >> 21) & 0x1f;
  case ADDIU:
   reg[rt] = reg[rs] + address;
   break;
[...]
  case J:
   tgtadr = ins & 0x3ffffff;
   pc = tgtadr << 2;
   break;
```

Listing 1: Extract from the 32-bit MIPS processor model that contains the ADDU, SLL, ADDIU, and J instruction [20]. The model is implemented as a state machine that iterates over the instructions. The current instruction is separated into its parts using logical *shift* and logical *and* operations.

from the instruction array. It is separated into its parts, e.g., the operation code*,* function code*, or* operands*. After separation, the instruction is processed according to its* operation code *or* function code*. The program counter is changed after instruction execution so that the next instruction is read from the instruction array. The model also contains a register file storing 32 entries and a data memory storing 64 entries. The execution of the iterations is stopped by a dedicated instruction (syscall 10), which means* exit *and is part of the program.*

2.2 Considered Problem

LegUp implements a new LLVM backend to synthesize hardware designs to Verilog implementations [8]. LegUp's input language defines a sequential execution scheme, but hardware designs define a parallel one. To formally describe the transformation of a sequential scheme into a parallel one, *synthesis schemes* [15, 2] can be used. According to the LegUp authors [8], the transformation from LLVM's internal representation language to Verilog does not follow such a synthesis scheme. The same goes for synthesis flows implemented by Bambu [29] and DWARV [27]. As a result, it is unclear how properties formulated at the model level relate to the implementation and how one could verify them. Moreover, it is unclear how to formulate properties in the hardware DSL because of its embedding into C. While there are tools to state and verify properties of C programs, such as Frama-C [13] or Astrée [12], these tools assume a compiler behaving according to the semantics defined by the C standard [30]. These assumptions are not the case for hardware designs as just described.

The missing ability to verify properties of models using the LegUp synthesis flow leads to the following questions. First, can we synthesize the 32-bit MIPS processor with a different synthesis flow that allows us to prove its correctness? Second, what would be the performance of the final implementation compared to the implementation synthesized by LegUp?

3 Correctness-oriented Synthesis Flows

In contrast to the acceleration-oriented synthesis flows just introduced, there are other synthesis flows like the correctness-oriented flows implemented by Kami [11] or [5]. In this section, we discuss and evaluate both flows to address the considered problem.

The idea of formally describing hardware using higher-order logic to prove correctness properties (formal synthesis) is not new [23, 17, 19]. Higher-order logic was used to avoid the combinatorial explosion of test vectors to ensure correctness and use symbolic reasoning instead. One of the first frameworks using this methodology were LAMBDA/DIALOG [16] and VERITAS [19]. Elaborating this methodology further description languages using higher-order logic, such as Hardware ML (HML) [28] and Bluespec [1, 6], were invented. The invention of Bluespec resulted in a hardware description language embedded into the proof assistant Coq to provide an automatic synthesis process that extracts a low-level implementation from a verified specification [11].

Kami and the Coq/CλaSH flow rely on the proof assistant Coq [4, 10] to specify and verify hardware designs and synthesize them afterwards in an implementation automatically. Coq specifies a functional behavior using the *Calculus of Inductive Constructions* (CiC). This formal language combines higher-order logic and a richly-typed functional programming language,

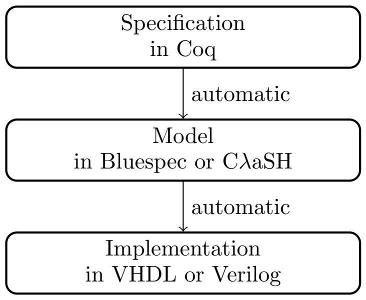

Figure 2: The correctness-oriented synthesis flows start with a specification using the proof assistant Coq. From the specification, a model in Bluespec (Kami) or CλaSH ([5]) is extracted. The model is finally synthesized to an RTL implementation, e.g., in VHDL or Verilog.

called Gallina. As higher-order logic is too expressive for automatic reasoning, a separated tactic language [14], called L_{tac}, is provided to let the engineer guide Coq's reasoning engine through the proof. Properties about the specified behavior are proven in this tactic language. As the engineer guides the reasoning process, proof assistants are also called interactive theorem provers. The synthesis flow of both Kami and the one proposed in [5] is sketched in Figure 2.

To our knowledge, Kami was the first project that proposes a formal processor specification extracted to a low-level implementation. Kami embeds a Domain-specific language (hardware DSL) into Gallina to describe hardware designs functional [11]. This language is based on the Bluespec hardware description language [1]. An executable Bluespec Verilog model is extracted from the specification, which is the input language for the Bluespec compiler. This compiler synthesizes a model to a hardware implementation in Verilog [1].

The hardware design synthesis flow introduced in [5] adds the hardware DSL CλaSH [3] to Coq's extraction backend. It uses Coq's specification language Gallina to describe the functional behavior of hardware designs. After the verification process, an executable CλaSH model is extracted from the specification. CλaSH is a functional hardware description language that borrows both its syntax and semantics from Haskell. The CλaSH model is finally compiled into a low-level implementation at the Register-Transfer Level (RTL). The supported HDLs are SystemVerilog, Verilog, and VHDL.

In contrast to the acceleration-oriented synthesis flows such as implemented by LegUp, both of these flows formulate a synthesis scheme describing how the semantics of the specification propagates to the final implementation. This synthesis scheme ensures that the proven properties at the specification level also hold for the implementation.

Both flows seem capable of addressing the problem discussed in Section 2.2. They allow the specification and verification of the 32-bit MIPS processor and subsequently synthesize the design on an FPGA. For example, Kami has been used to implement a RISC-V multi-core processor as a case study [11]. However, the flow [5], which we call Coq/CλaSH in the rest of the paper, is more light-weight and flexible, as CλaSH allows the synthesis of arbitrary combinational and synchronous sequential hardware designs [3, 5]. Because of its flexibility, we chose this one as a representative of correctness-oriented hardware synthesis flows.

However, the question remains whether such correctness-oriented synthesis flows will result in less efficient designs concerning the performance of the synthesized circuit? After all, correctness-oriented flows emphasized property verification and not so much on the acceleration of implementations. For this reason, one would not be surprised if the implementation synthesized by the Coq/CλaSH flow would be slower than the one synthesized by LegUp. Even then, the question would remain by *how much* the design would be slower.

4 Specification and Verification of the MIPS Processor

In this section, we describe the specification of the 32-bit MIPS processor in Gallina, using the Coq/CλaSH hardware design synthesis flow, and how properties about it are stated and verified. By this, we provide an analysis of the correctness-oriented design flow and a benchmark that, afterward, is used to compare to the acceleration-oriented design flow. The foundation regarding the implemented instructions, register file, and memory is the 32-bit MIPS processor, described in Section 2.1.

4.1 Specification of Sequential Hardware Designs

To represent sequential circuits functionally in the Coq/CλaSH synthesis flow, Mealy or Moore machines are used [5]. These machines abstract the clock by defining state transitions, which allow a time-controlled execution. The advantage of such a description is that we can prove properties such as liveness [7] about the hardware design. The type of the Mealy machine specified in Gallina is shown in Listing 2. In this case, a Mealy machine is used, as we need access to the program counter in the current state for calculating the output, as we see later.

```
Fixpoint mealy {S I O:Type}
    (f: S -> I -> (S*O))
    (s: S)
    (l: list(I))
  : list(O)
```

Listing 2: Function type of the *Mealy* machine specified in Gallina [5]. The machine takes a function as its first argument. This function maps a state (S) and an input (I) to a tuple of a new state and an output (S*O). An initial state and a list of inputs is also required by the function type. The result is a list of outputs. The types S, I, and O are inferred at compile time.

The recursive specification of the Mealy machine calls the function *f* with the current state and input, and returns a new state and an output, until every input is processed.

In our case, the program counter, the register file, and the memory define the state (S). The input (I) is ignored by our specification of function *f*, as the benchmark (the program mentioned in Section 2.1) is a fixed set of instructions. Since an output (O) is required, the result of an instruction is returned. Listing 3 shows the instantiated function type of function *f* required by the Mealy machine definition. The *registerFileType* and the *memoryType* are fixed-sized vectors of the length 32 and 64, respectively.

```
Definition mips
    (data : registerFileType*memoryType*
            Unsigned32.int)
    (dummy : bool)
  : (registerFileType*memoryType*
    Unsigned32.int)*Unsigned32.int
```

Listing 3: Function type of the *mips* function specified in Gallina.

The first argument is called *data*. It is a tuple of the *RegisterFileType*, the *memoryType* and the *Unsigned32.int* type. The first two types are vectors of a fixed size that represent the arrays of the

LegUp model. The third type represents the program counter. The second argument to the *mips* function is of the type boolean. The Coq/CλaSH synthesis flow used in this work extracts a CλaSH model from a specification. Since the MIPS processor model defines a constant set of executed instructions, there is no actual input, so we call that argument *dummy.* The return type is a tuple of the same tuple as the first argument and a 32-bit unsigned value (Unsigned32.int). This value defines the output of the Mealy machine, e.g., the result of an instruction.

After an instruction was executed, the changed register file, the changed memory, and the new program counter are returned (the new state). How the register file or memory is changed depends on the executed instruction.

4.2 Construction of Instructions

The instructions, together with their operands, are encoded as 32-bit unsigned integer values. These instructions are specified in three different formats. In addition to the operation code (op), which they all have in common, they differ in interpreting their bits. The operation code always consists of the highest six bits. The first format is the R-Format that specifies three registers, one shift amount, and one function code and has the following layout:

$$\underbrace{op(6)\ rs(5)\ rt(5)\ rd(5)\ shamt(5)\ funct(6)}_{31\ \ldots\ 0\ bits}$$

The three registers state the first register operand (rs - 5 bits), the second register operand (rt - 5 bits), and the register destination (rd - 5 bits). The shift amount (shamt) also has 5 bits, while the function code (funct) has 6 bits. The operation code in the R-Format is always zero. The second format is the I-Format. In addition to the operation code, this format specifies two registers and one immediate value and has the following layout:

$$\underbrace{op(6)\ rs(5)\ rt(5)\ immediate(16)}_{31\ \ldots\ 0\ bits}$$

The operation code and the two registers have the same bit sizes as in the R-Format. The immediate value is 16 bit in size. The third format is the J-Format and states one address value, which results in the following format:

$$\underbrace{op(6)\ address(26)}_{31\ \ldots\ 0\ bits}$$

```
let instr := nth instructionMemory pc in
let op := getOpCode instr in
match toFormat op with
 | RFormat =>
  let funct := getFunct instr in
  let shamt := getShamt instr in
  let rd := getRD instr in
  let rt := getRT instr in
  let rs := getRS instr in
  match toFunctionCode funct with
   | ADDU =>
    let value := add (nth registerFile rs)
                     (nth registerFile rt) in
    let registerFile' :=
    replaceAt rd value registerFile in
    ((registerFile',memory,newPC pc),value)
   | SLL =>
    let value := sll (nth registerFile rt)
                     (toZ shamt) in
    let registerFile' := replaceAt
     rd value registerFile in
    let pc' := newPC pc in
     ((registerFile',memory,pc'),value)
    [...]
 | IFormat =>
   let address := getAddress instr in
   let rt := getRT instr in
   let rs := getRS instr in
   match toOperationCode op with
    | ADDIU =>
     let value := add (nth registerFile rs)
      address in
     let registerFile' := replaceAt rt value
      registerFile in
     ((registerFile',memory,newPC pc),value)
     [...]
  | JFormat => match toOperationCode op with
   | J =>
    let tgtadr := getTargetAddress instr in
    let pc' := sll tgtadr z2 in
    ((registerFile,memory,pc'),pc')
    [...]
  end
```

Listing 4: Extract of the 32-bit MIPS processor specified using the Coq/CλaSH synthesis flow. It pattern matches over the instructions to access the parts of an instruction as defined by the format.

The address value is 26 bits in size.

These formats enable a unique interpretation of the instruction bits. Our specification of the 32-bit MIPS processor, sketched in Listing 1, is shown in Listing 4.

To separate an instruction into its parts, we implemented a couple of functions. We illustrate the implementations of these functions by reference to the *ADDU* instruction (R-Format), shown in Listing 4. This instruction adds two unsigned 32-bit values stored in the register file under the addresses *rs* and *rt* and stores the result in the register file at the address *rd.*

- **getOpCode**: The operation code is selected by applying right logical shift by the value 26 to the instruction.
- **getFunct**: The function code is determined by logical conjunction, which is applied to the instruction with the hexadecimal value *0x3f.* This value represents a bit vector of 26 0s followed by six 1s from the most significant

bit (MSB) to the least significant (LSB) (big-endian).

- **getShamt**: The shift amount is selected by first applying right logical shift by the value 6 to the instruction. Afterward, logical conjunction is applied to that value and the hexadecimal value *0x1f*. This value represents a bit vector of 27 0s followed by five 1s, from MSB to LSB (big-endian).
- **getRD**: The destination register is selected by first applying right logical shift by the value 11 to the instruction. Afterward, logical conjunction is applied to that value and the hexadecimal value *0x1f*.
- **getRT**: The first register is selected by first applying right logical shift operation by the value 16 to the instruction. Afterward, logical conjunction is applied, as for the destination register.
- **getRS**: The second register is selected by a right logical shift of the instruction with a shift amount of 21 first. Afterward, logical conjunction is applied, as for the destination register.

After separating the instruction into its parts as defined by the format, the actual operation can be performed. The *ADDU* instruction defines two register file addresses (rs and rt). The values for these addresses are selected first; *nth registerFile rs* and *nth registerFile rt*. The function *nth* returns a fixed-size vector (registerFile) for a given index (rt). The addition of these two values is stored in the register file at the index *rd* (replaceAt rd value registerFile). The *replaceAt* function replaces a value (value) at an index (rd) of a fixed size vector (registerFile). The final step is to replace the old register file (registerFile) with the changed one (registerFile') and increment the program counter for the next instruction.

```
Theorem mips_addu:
  forall registerFile : registerFileType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in
  let op := getOpCode instr in
  let funct := getFunct instr in
  let rd := getRD instr in
  let rt := getRT instr in
  let rs := getRS instr in
  let value := add (nth registerFile rs)
                   (nth registerFile rt) in

  toFormat op = RFormat /\
  toFunctionCode funct = ADDU ->
  mips (registerFile, memory, pc) dummy =
    ((replaceAt rd value registerFile,
     memory,newPC pc),value).
Proof.
  proveRFormat.
Qed.
```

Listing 5: Theorem specified in Gallina to verify that the ADDU instruction adds the two values of the register addresses (rt and rs) and stores the result at the register file address (rd).

```
Ltac proveRFormat :=
  intros;
  match goal with
   | [ H : _ |- _] =>
     destruct H as [H1 H2 ];
     unfold mips;
     match goal with
      | [op: _ |- _] =>
        unfold op in H1;
        match goal with
         | [instr: _ |- _] =>
           unfold instr in H1;
           rewrite H1;
           match goal with
            | [funct: _ |- _] =>
              unfold funct in H2;
              unfold instr in H2;
              rewrite H2;
              auto
           end
        end
     end
  end.
```

Listing 6: *proveRFormat* tactic in L_{tac}. The tactic allows the proving of properties that have the format shown in Listing 5. This format requires splitting the instruction in *op*, *funct*, etc., the instruction format to be *RFormat*, and the specification of the *function code*, e.g., ADDU.

4.3 Proving Properties

After the MIPS processor was specified in Gallina, properties can be proven about this specification. Listing 5 shows such a property about the specified *ADDU* instruction, which is defined as a theorem in Coq.

The theorem states that if the operation code (op) indicates the *RFormat* and the function code (funct) indicates the ADDU instruction, then the values of the register file addresses *rs* and *rt* are added together. The result is stored in the register file at address *rd*. To verify the final result is calculated correctly, a few statements are defined, starting with the *let* keyword.

Coq's tactic language L_{tac} allows the specification of user-defined proof methods [14]. We specified a proof method called *proveRFormat* that allows proving properties about instructions, which implement the *R-Format* and follow the theorem structure described above. The tactic is seen in Listing 6.

The *proveRFormat* tactic is built from tactics already provided by Coq. Coq splits a proof into a context that contains introduced variables, hypotheses, and a goal that is to be proven by the context. The proof is finished if there are no subgoals to prove. The first tactic that is used is the *intros* tactic. This tactic introduces variables, such as *registerFile* or *op* and the hypothesis of the implication. The next step is to match the hypothesis – Coq names the hypothesis with *H* by default. Since the hypothesis is an *and* expression, we destruct it into two hypotheses (H1 and H2) and replace the name *mips* with its specification in the subgoal by calling the *unfold* tactic. The *op* and *instr* variables are unfolded in the hypothesis H1 to rewrite it in the context. This rewriting reduces the subgoal to the second *match* statement (*toFunctionCode funct* seen in Listing 4). The final steps are to unfold the *funct* and *instr* variables in the second hypothesis (H2), applying H2 to the context by the *rewrite* tactic, and finish the proof applying the *auto* tactic. The *auto* tactic tries to automatically solve a goal by introducing new variables and hypotheses to the context and applying built-in tactics to the resulting subgoals. If the *auto* tactic fails, the subgoal remains unchanged. Similarly, proof methods for instructions implementing either the I-Format or the J-Format were specified.

These proof methods simplify the verification of properties about instructions that have already been implemented and those that might be added in the future. The specification and verification of the theorems for the rest of the instructions work analogously to the one above. Due to size constraints, we cannot show them and the specification of the proof methods here in detail[1].

After verifying that the specified instructions are correct using the theorem formats and tactics described above, other properties have to be shown that the processor specification is correct and functionally behaves as expected. One of those properties is that the NOP (no operation) instruction is interpreted as specified by the MIPS architecture standard [25]. This instruction does not change any state but only increments the program counter. For the 32-bit MIPS processor NOP is not specified as an extra operation code but maps to the shift logical left operation (SLL), described in Listing 4. The theorem shown in Listing 7 verifies this behavior.

```
Theorem mips_nop:
  forall registerFile : registerFileType,
  forall memory : memoryType,
  forall pc output : Types.Unsigned32.int,
  forall dummy : bool,

  let nop := Ox00000000 in

  let registerFile' := replaceAt
   (and (srl nop z11) Ox1f)
   (sll (nth registerFile
   (and (srl nop z16) Ox1f))
   (toInt (and (srl nop z6) Ox1f)))
   registerFile in

  let output := sll
    (nth registerFile
    (and (srl nop z16) Ox1f))
    (toInt (and (srl nop z6) Ox1f)) in

  nth instructionMemory pc = nop ->
  mips (registerFile,memory,pc) dummy =
  ((registerFile',memory,newPC pc),output).
```

Listing 7: The NOP instruction is implemented for the MIPS processor as: sll r0 r0 0. The value of register r0 is logically shifted left by 0, and the result is stored in r0. The theorem *mips_nop* ensures this behavior. Note that the register r0 returns the constant zero [25].

The NOP instruction is a 32-bit vector containing only zeros (Ox00000000). This instruction does not change the content of the initial register file, but the SLL operation returns a new register file, as seen in Listing 4. For this reason, the expression *registerFile'* is specified. As the processor interprets the NOP instruction as the *SLL* operation, the *output* specifies the result of this operation. Note that this theorem cannot be proven using the *proveRFormat* as it does not contain a conjunction in the hypothesis, i.e., it does not follow the required format.

In this section, we have specified and verified a 32-bit MIPS processor using the Coq/CλaSH synthesis flow. The verification of properties successfully addresses the deficiencies of the LegUp synthesis flow, as described in Section 2.2. The specification was automatically synthesized to a Verilog implementation using CλaSH, to answer the question of how the performance of the two implementations compares.

5 Evaluation

In this section, we evaluate and discuss the performance of both the acceleration-oriented and correctness-oriented synthesis flow. The foundation is the RTL implementation of the 32-bit MIPS processor synthesized by LegUp and

[1]The specification of the 32-bit MIPS processor can be found under: https://gitlab.informatik.uni-bremen.de/fritjof/mips-processor

Coq/CλaSH. Both implementations implement the same instructions and execute the program, described in Section 2.1. In the following, the results obtained by both implementations are summarized first. Afterwards, we discuss what conclusions can be drawn from that.

5.1 Results

Table 1 shows the performance results of both implementations. The values in this table should be considered an approximation as they highly depend on the FPGA the hardware design is synthesized for; they indicate rather than quantify exactly the relation between the two synthesized designs.

We now explain the individual rows of Table 1 in detail. The first row contains the maximum clock frequency F_{MAX} at which the final circuit can be operated. As we see, the circuit synthesized by the LegUp HLS framework can be operated at a higher frequency than the one synthesized by Coq/CλaSH.

The second row contains the clock cycles needed by the processor implementations to execute the example program (clock latency). These values are evaluated by simulation using Intel® Quartus® ModelSim. For simulation, a clock cycle of 20 ns was used. Together with the maximum frequency, this results in the time it takes in μs to execute the program (Wall-Clock) provided by the model, described in Section 2.1. The implementation synthesized by LegUp takes 79.5 μs for execution, while the implementation synthesized by Coq/CλaSH takes 218.76 μs.

The fourth row contains the Adaptive Logic Modules (ALMs) called Lookup Tables (LUTs) in the Xilinx Vivado synthesis tool. These are the basic building blocks for hardware designs on an FPGA. As seen, the circuit synthesized by LegUp consumes 2% of the available ALMs, while the one synthesized by Coq/CλaSH consumes 3% of the available ALMs. To better classify these values, we take a look at the last row of the table. This row contains the total block memory in bits, which is essentially the block RAM of the FPGA. The memory of an FPGA is separated into distributed RAM (ALMs) and block RAM. LegUp stores each local and global memory in the separated block RAM by default. For larger memories, the block RAM is much faster than distributed RAM. The implementation synthesized by Coq/CλaSH uses no block Ram but stores the entire design in distributed RAM.

The fifth row of Table 1 contains the consumed registers. To classify these values, we consider the design of the 32-bit MIPS processor. The LegUp model of the processor changes the values of an array in place, so only one array, e.g. for the register file, is needed. The functional foundation of the Coq specification and thus the CλaSH model requires *single assignment* of variables. For this reason, the underlying Mealy machine needs the changed register file as part of the new state, as seen in Listing 4. The synthesis of this behavior results in the consumption of more registers.

The sixth row contains the amount of used Digital Signal Processing (DSP) blocks. These blocks describe a dedicated functionality, e.g. multipliers, which are provided by the synthesis tool. The usage of those DSP blocks is automatically inferred by analyzing the RTL code.

5.2 Discussion

In this section, we discuss the results of our evaluation described above. Acceleration-oriented synthesis flows such as LegUp define a model in a hardware DSL embedded into C. The low-level nature of this language allows a more acceleration-oriented implementation of hardware designs, but lacks the verification of properties, as described in Section 2.2.

On the other hand, correctness-oriented synthesis flows such as the Coq/CλaSH flow define a behavior functionally at a higher level of abstraction, making them easier to understand, and hence less error-prone [21], and susceptible to verification in the first place. It does, however, have an impact on performance, resulting in lower clock frequency or a higher amount of clock cycles.

Our evaluation shows that although the implementation using the correctness-oriented flow was in general slower than the one using LegUp, we were able to synthesize a 32-bit MIPS processor which is in the *same ball-park* concerning performance indicators like clock frequency or execution time using the standard tool chain of the Coq/CλaSH flow. This systematic comparison shows the huge potential of correctness-oriented synthesis flows, showing that these flows result in circuits with competitive performance.

Research projects like Kami show that the synthesis of verified specifications is a subject of current research. The successful synthesis

Table 1: Evaluation of the two 32-bit MIPS processor implementations. The *LegUp* column contains the values based on the implementation synthesized by the LegUp HLS framework. The *Coq/CλaSH* column contains the values of the synthesized design based on the Coq/CλaSH synthesis flow used in this work, which is discussed in Section 3.

	LegUp	**Coq/CλaSH**
F_{MAX} in [MHz]	63.36	55.86
Cycles	5035	12220
Wall-Clock in μs	79.5	218.76
ALMs	1045 / 56480 (2%)	1772 / 56480 (3%)
Registers	939	1644
DSP Block	6 / 156 (4%)	2 / 156 (1%)
Total Block Memory Bits	3072 / 7024640 (< 1%)	0 / 7024640 (0%)

The RTL implementations in Verilog were synthesized for the Cyclone V family using the commercial synthesis tool Intel® Quartus® Prime.

of a RISC-V processor shows the potential of correctness-oriented flows [11]. When hardware is used in safety-critical systems, verifying the correct functional behavior becomes essential; our evaluation demonstrates that correctness-oriented flows can achieve this without sacrificing too much performance. Moreover, there is still a huge unexplored potential for performance gains in correctness-oriented flows, whereas adding verification to an acceleration-oriented flow seems, at first sight, far more challenging.

For these reasons, our analysis suggests that correctness-oriented synthesis flows can be employed when the need for verification arises in a performance-oriented environment.

6 Conclusion

In this work, we analyzed the acceleration-oriented hardware design synthesis flows as implemented by Bambu [29], DWARV [27], and LegUp [8] and showed their missing ability of property verification. In contrast, we considered correctness-oriented hardware design synthesis flows, as implemented by Kami [11] or the Coq/CλaSH synthesis flow [5]. We address the question of a quantitative analysis of the trade-off concerning the performance between both flows by comparing a non-trivial circuit designed by two representative flows. The designed circuit was a synthesized RTL implementation of a 32-bit MIPS processor [20]. LegUp was chosen as a representative of the acceleration-oriented synthesis flows, while the Coq/CλaSH flow was chosen as a representative of the correctness-oriented flows.

Our evaluation, seen in Table 1, allows a quantitative analysis of the trade-off between performance and correctness. This paper indicates that using a hardware design flow allowing correctness proofs does not require sacrificing much performance in the implemented system. However, if more performance is needed we argue that it is easier to increase the performance of circuits synthesized by correctness-oriented flows than to add correctness to acceleration-oriented flows. For this reason, we suggest further research to enhance the performance of correctness-oriented flows.

Besides the MIPS instruction set architecture the open RISC-V instruction architecture set [31] has got a lot of attention over the last decade. For example, Kami provides a verified 32-bit RISC-V processor that implements the integer instruction set. It would be interesting how the Coq/CλaSH approach compares to the low-level implementation synthesized by Kami concerning performance. This comparision, however, would be future work as it is outside the scope of this work.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grantno. 01IW16001, the LIT Secure and Correct Systems Lab funded by the State of Upper Austria, as well as by the BMK, BMDW, and the State of Upper Austria in the frame of the COMET program (managed by the FFG).

References

[1] Arvind. “Bluespec: A language for hardware design, simulation, synthesis and verification Invited Talk”. In: IEEE Computer Society, 2003, p. 249.

[2] Christiaan Baaij and Jan Kuper. “Using Rewriting to Synthesize Functional Languages to Digital Circuits”. In: *Trends in Functional Programming (TFP).* Vol. 8322. Lecture Notes in Computer Science. Springer, 2013, pp. 17–33.

[3] Christiaan Baaij et al. “CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell”. In: *Euromicro Conference on Digital System Design (DSD).* 2010, pp. 714–721.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq’Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An (EATCS) Series. Springer, 2004.

[5] Fritjof Bornebusch et al. “Towards Automatic Hardware Synthesis from Formal Specification to Implementation”. In: *Asia and South Pacific Design Automation Conference (ASP-DAC).* 2020.

[6] Thomas Bourgeat et al. “The essence of Bluespec: a core language for rule-based hardware design”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.* Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 243–257. DOI: 10.1145/3385412.3385965. URL: https://doi.org/10.1145/3385412.3385965.

[7] Manfred Broy. “Verifying of interface assertions for infinite state Mealy machines”. In: *Journal of Computer and System Sciences* 80.7 (2014), pp. 1298–1322.

[8] Andrew Canis et al. “LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems”. In: *ACM Trans. on Embedded Computing Systems* 13.2 (2013), 24:1–24:27.

[9] Andrew Canis et al. “LegUp High-Level Synthesis”. In: *FPGAs for Software Programmers.* Ed. by Dirk Koch, Frank Hannig, and Daniel Ziener. Springer, 2016, pp. 175–190.

[10] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[11] Joonwon Choi et al. “Kami: a platform for high-level parametric hardware specification and its modular verification”. In: *Proceedings of the ACM on Programming Languages (PACMPL)* 1.ICFP (2017), 24:1–24:30.

[12] Patrick Cousot et al. “The ASTREÉ Analyzer”. In: *European Symposium on Programming.* 2005, pp. 21–30.

[13] Pascal Cuoq et al. “Frama-C - A Software Analysis Perspective”. In: *Software Engineering and Formal Methods.* 2012, pp. 233–247.

[14] David Delahaye. “A Tactic Language for the System Coq”. In: *International Conference on Logic for Programming and Automated Reasoning (LPAR).* Vol. 1955. Lecture Notes in Computer Science (LNCS). Springer, 2000, pp. 85–95.

[15] Dirk Eisenbiegler and Ramayya Kumar. “Formally embedding existing high level synthesis algorithms”. In: *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, (CHARME).* Vol. 987. Lecture Notes in Computer Science (LNCS). Springer, 1995, pp. 71–83.

[16] Simon Finn et al. “Formal system design — interactive synthesis based on computer-assisted formal reasoning”. In: *Applied Formal Methods for Correct VLSI Design.* IMEC-IFIP International Workshop. Elsevier, Nov. 1989, pp. 97–110.

[17] Mike Gordon et al. “Automatic Formal Synthesis of Hardware from Higher Order Logic”. In: *Electron. Notes Theor. Comput. Sci.* 145 (2006), pp. 27–43.

[18] Soonhoi Ha and Jürgen Teich, eds. *Handbook of Hardware/Software Codesign.* Springer, 2017.

[19] Keith Hanna, N. Daeche, and M. Longley. “Formal Synthesis of Digital Systems”. In: *International Workshop on Applied Formal Methods For Correct VLSI Design.* 1989, pp. 532–548.

[20] Yuko Hara et al. “Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis”. In: *Journal of Information Processing (JIP)* 17 (2009), pp. 242–254.

[21] J. Hughes. “Why Functional Programming Matters”. In: *Computer Journal* 32.2 (1989), pp. 98–107.

[22] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. “Scheduling for Functional Pipelining and Loop Winding”. In: *Design Automation Conference (DAC)*. ACM, 1991, pp. 764–769.

[23] Ramayya Kumar et al. “Formal Synthesis in Circuit Design - A Classification and Survey”. In: *International Conf. on Formal Methods in CAD*. Vol. 1166. Lecture Notes in Computer Science (LNCS). Springer, 1996, pp. 294–309.

[24] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2004, pp. 75–88.

[25] MIPS. *MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual*. Revision 6.06. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf. 2016.

[26] Razvan Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604.

[27] Razvan Nane et al. “DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler”. In: *International Conference on Field programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 619–622.

[28] John W. O’Leary et al. “HML: A Hardware Description Language Based on Standard ML”. In: *Computer Hardware Description Languages and their Applications, Proceedings of the International Conference on Computer Hardware Description Languages and their Applications CHDL*. Vol. A-32. IFIP Transactions. North-Holland, 1993, pp. 327–334.

[29] Christian Pilato and Fabrizio Ferrandi. “Bambu: A modular framework for the high level synthesis of memory-intensive applications”. In: *International Conference on Field programmable Logic and Applications (FPL)*. IEEE, 2013, pp. 1–4.

[30] *Programming languages — C*. ISO/IEC Standard 9899:1999(E). Second Edition. 1999.

[31] RISC-V. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. Version 1.12-draft. https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-privileged.pdf. 2020.