Formal Verification of n-bit ALU using Theorem Proving

Sumayya Shiraz and Osman Hasan

School of Electrical Engineering and Computer Science
National University of Sciences and Technology (NUST) Islamabad, Pakistan
{sumayya.shiraz,osman.hasan}@seecs.nust.edu.pk

Abstract. Automatic verification techniques, like automated theorem proving and model checking, cannot analyze large circuits due to the heavy requirements of memory and computational power. On the other hand, we can verify generic circuits, with universally quantified variables, using interactive theorem provers and thus overcome the above-mentioned limitations but at the cost of significant user guidance in the proof process. To facilitate this process and thus reduce the user involvement in the proofs, we recently proposed a higher-order-logic formalization of all the commonly used combinational circuits, like basic gates, adders, multiplier, multiplexers, demultiplexers, decoders and encoders, using the HOL4 theorem prover. In this project's paper, we describe this formally verified library and illustrate its utilization by verifying an n-bit arithmetic logic unit (ALU).

1 Introduction

Verification of digital designs is of utmost importance due to their extensive usage in safety-critical domains, such as health and transportation, where the cost of an undetected system bug is quite high. Traditionally, digital designs are verified using simulation, which ascertains the correctness of the design by observing the behavior of the circuit under a subset of all possible inputs only. Formal verification [15] is an accurate alternative to simulation that overcomes its limitations by proving or disproving the correctness of the given design against its desired properties mathematically. The main principle behind formal analysis of a digital circuit is to construct a computer-based mathematical model of the given circuit and formally verify, within a computer, that this model meets rigorous specifications of intended behavior. Thus, the engineer working with a formal methods-based verification tool has to develop a formal model of the given circuit and the formal specification of the desired properties. Moreover, she may be involved in the verification task as well.

There are some formal verification tools, mainly based on model checking [10] and automated theorem proving techniques [14], that accept Verilog models [2] and automatically translate them to the corresponding formal models and also automatically verify the relationship between the formal model and its corresponding specification. Thus, the verification engineer has to be involved in the

formal specification of the properties only. These kind of tools, such as FormalPro by Mentor Graphics, Conformal by Cadence, Synopsys Hector, Calypto's SLEC and Formality by Synopsys, are quite well-suited for the industrial setting and are thus widely accepted by the industry as well. However, model checking is generally limited to sequential circuits and also suffers from the well-known state-space explosion problem. Similarly, automated theorem provers cannot cope with the verification problems of large designs as well, due to an exponential increase in computations with an increase in the number of variables and intermediate nodes. Interactive theorem provers [14], using the expressive higher-order logic, can overcome these shortcomings but at the cost of explicit user involvement. The verification engineer needs to manually construct a logical model of the system and then verify the desired properties while guiding the theorem proving tool. This could be a very rigorous process and the user needs to be an expert in both system design and theorem proving skills. This drawback limits the usage of higher-order-logic theorem proving in the mainstream hardware industry where the engineers prefer to have push-button type tools.

To minimize the user involvement in using an interactive theorem prover for the verification of combinational circuits, we recently proposed a library of combinational circuits [27], consisting of formally verified generic circuits of commonly used components, such as various implementations of n-bit Adders, n-bit multiplier, n:1 Multiplexers, 1:n Demultiplexers, n:2^n Decoders, 2^n:n Encoders and n-bit logic gates. The verification of these generic components was done interactively but the availability of this library greatly facilitates the verification of more complex designs. The user of the proposed approach has to just provide the structure of the combinational circuit to be verified in terms of its sub-components, based on the existing components in the proposed library, and its desired behavior in the language supported by the HOL4 theorem prover. The relationship between the structural view and the behavior of the given circuit can then be verified using the library of formally verified generic circuits in a very straightforward manner.

In this project's paper, we describe all the main components of this library [27] and illustrate its effectiveness in formally verifying generic circuits by formally verifying an n-bit arithmetic logic unit (ALU) with very minimal user interaction. The main motivation of this paper is to illustrate the utilization of our formally verified library [27] in verifying more complex combination circuits. We have used the HOL4 theorem prover for this work, mainly because the existing library of formal combinational components [27] has been developed in HOL4.

2 Related Work

The first-order-logic theorem prover ACL2 has been used to verify different hardware designs, including register-transfer level (RTL) models of floating-point hardware [8] and pipeline machines using first-order quantification [24]. Similarly, a framework is proposed for the mechanized certification of secure hard-

ware systems using ACL2 [23]. However, these verifications are done for specific operand widths of the components. In order to alleviate this problem, ACL2 has been used in conjunction with symbolic simulation for verifying hardware [11] and VIA nano microprocessor components [32]. However, using symbolic simulation compromises the completeness of the analysis and thus accuracy. Similarly, ACL2 has also been used with IBM's SixthSense model checker [16,17] to develop a hybrid verification framework for digital hardware. But the scalability of this technique is a major concern since the state transition checks grow exponentially for large circuits and thus the automatic verification capability is compromised.

Interactive theorem provers, using higher-order logic, can overcome the limited expressiveness problem of ACL2. Thus, PVS has been used for the verification of some large designs, including some FPGA designs [9] and the floating point unit used in the VAMP processor [3], which supports addition, subtraction, multiplication, division, comparison, and conversions. Similarly, a hardware verification tool, called PROVERIFIC [28], allows Property Specification Language (PSL) assertions to be used with PVS. All the above-mentioned works require detailed user guidance in the proof process. Moreover, these formalizations are dedicated towards a particular circuit and are thus not generic.

The PVS theorem prover has been used along with decision procedures and BDD-based propositional simplification to automatically verify combinational circuits [7]. However, this proof strategy tackles each circuit verification from scratch whereas our approach is modular as we utilize the formally verified models of commonly used combinational circuits to verify more complex circuits. This kind of modularity makes the verification approach more scalable. A library of basic circuits is also implemented and verified in PVS [4] that is quite similar to the one presented in this paper. However, this work is just focused towards the verification of microprocessor designs. Secondly, the formalization and verification details of the components, reported in this work, are not openly available. Thus, our idea is mainly inspired from this work but we have developed recursive definitions for all the commonly used combinational circuits and have formally verified them using the HOL4 theorem prover [27]. To the best of our knowledge, these kinds of generic recursive definitions of combinational circuits have never been presented and used to verify more complex combinational circuits in the literature before.

The Coq theorem prover is based on the Calculus of (Co)Inductive Constructions (CiC) and features dependent types, which are quite helpful in creating reliable circuit models as errors can be caught earlier by type checking [5]. Braibant [5] created a library in Coq to facilitate modeling and verifying hardware circuits. Although dependent types, available in this library, are helpful in creating reliable definitions, the library still requires the user to guide the proof tools, which somewhat limits the scope of this work for industrial usage. A step-by-step procedure for the formal verification of a multiplier in CiC is given in [22]. But this work also requires extensive user interaction for verifying new designs and is specific for one example only.

The HOL theorem prover has been used for the verification of the SPW Data-strobe (DS) encoding [21] and multiway decision graphs (MDG) components library [6]. Both of these works are application specific. A hardware platform for a gate level electronic control unit has been implemented and interactively verified in HOL [31]. Similarly, the HOL Light theorem prover has been used for the verification of floating-point algorithms for division, square root and transcendental functions [13]. However, the verification does not involve the gate level implementations and requires significant user interaction.

Many hybrid techniques, based on the idea of exploiting the strengths of interactive theorem proving and automatic verification tools, have been developed as well. The HOL theorem prover has been integrated with MDG for hardware verification [19]. Similarly, the Pipelined Double-Precision IEEE Floating-Point Multiplier is verified by the Voss hardware verification system using a combination of theorem proving and model checking [1]. The Floating point divider unit of an Intel IA-32 microprocessor [18] and large-scale industrial trials on data-path-dominated hardware [25] are formally verified using the Forte framework, which uses the ThmTac theorem-prover and the symbolic trajectory model checker. However, the verification in the Forte framework requires significant user interaction and thus is not very easy to work with. The Isabelle/HOL theorem prover has been used along with the nuSMV model checker and SAT solvers for verifying some basic combinational circuits and the simple sequential DLX processor at the gate level [30]. However, all these works are focused on one or a subset of combinational circuits. Similarly, due to their hybrid nature, they also suffer from the state-space explosion problem.

Based on the above-mentioned review, it is observed that all of the interactive theorem proving-based verification approaches for combinational logic circuits require the formalization of the circuit to be verified from scratch and require considerable user guidance during the proof process. In order to alleviate these issues, a library of formally verified commonly used combinational circuits is created [27]. The definitions of these formally verified combinational circuits can be readily built upon to formalize almost any other combinational circuit. Moreover, the formally verified expressions of the combinational circuits in this library allow us to verify proof goals of any other combinational circuit in a very straightforward manner involving simple rewriting steps. It is important to note that the development of the library involved human guidance and interactive reasoning but, once developed, this library greatly facilitates the formalization and verification process for more complex combinational circuits. The effectiveness of the library can be estimated with the help of formal verification of generic n-bit ALU, verified in this paper, which is done with minimal user involvement.

3 Formal Verification of Generic Combinational Circuits

This section gives a brief introduction to the formally verified generic library [27] of the commonly used combinational components. This formalization, mainly inspired by the seminal work on digital circuit verification done at the University of

Cambridge, UK [12], is the core component for verifying generic combinational circuits. The main idea is to model generic (arbitrary-input) circuit diagrams (implementations) of combinational circuits in a recursive manner, as shown in Fig. 1, by using the logical sub-components of the given circuit and their interconnections. The inputs and outputs of these circuits are modelled as lists of booleans to allow generic definitions. We have used the big endian format for representing these lists of booleans, i.e., 4 bit input data list a is represented as [a4;a3;a2;a1], with a4 being the most-significant bit. The primary inputs and outputs of these definitions are universally quantified while the internal connection points, hidden from the external world, are introduced using existential quantifiers. The behavior, or specifications, of these combinational circuits is represented in terms of their desired input-output relationships. The relationships between the implementations and their corresponding specifications are then verified using induction on the input variables within the sound core of a theorem prover. Any combinational circuit using the formally verified components of this library can then be verified by simply using the above-mentioned formally verified relationships with a very minimal user interaction.

Now, we explain the formally verified components of our library one by one.

3.1 Logic Gates

All of the primitive logic gates i.e., NOT, AND, NAND, OR, XOR, NOR and XNOR, are formally defined in the library [27]. All of these definitions, except the inverter, are generic and thus can be used to model the respective gate with any number of inputs.

3.2 Multiplexer

The n:1 Multiplexer (Mux) [20] passes the signal of any one of the n input data lines to the one bit output line depending upon the $\log_2 n$ input select lines. Fig. 1(a) provides the recursive implementation of a generic n:1 Mux, where n is the width of data input lines a, k is the width of select input lines s and b is a boolean output signal. The relation between the width of select and data input lines can be specified by the equation $k = \log_2 n$, or in other words $n = 2^k$. The primitive 2:1 Mux can be implemented using basic logic gates [26]. The n:1 mux is formally verified in Theorem 1, where implementation and specification for n:1 mux is formally defined [27] as `mux_imp_n a s b` and `mux_spec_n a s b` respectively.

Theorem 1: $\vdash \forall$ `a s b. (¬(s = []) ∧ (LENGTH a = 2 EXP LENGTH s))`
`⇒ (mux_imp_n a s b = mux_spec_n a s b)`

where assumptions ensure that at least one select line is required to ensure a valid MUX, and define the relationship between the input data and select lines. Verification of Theorem 1 is primarily based on induction on variable s. The proof script for the formal reasoning about Theorem 1 consists of about 400 lines of HOL code [26].

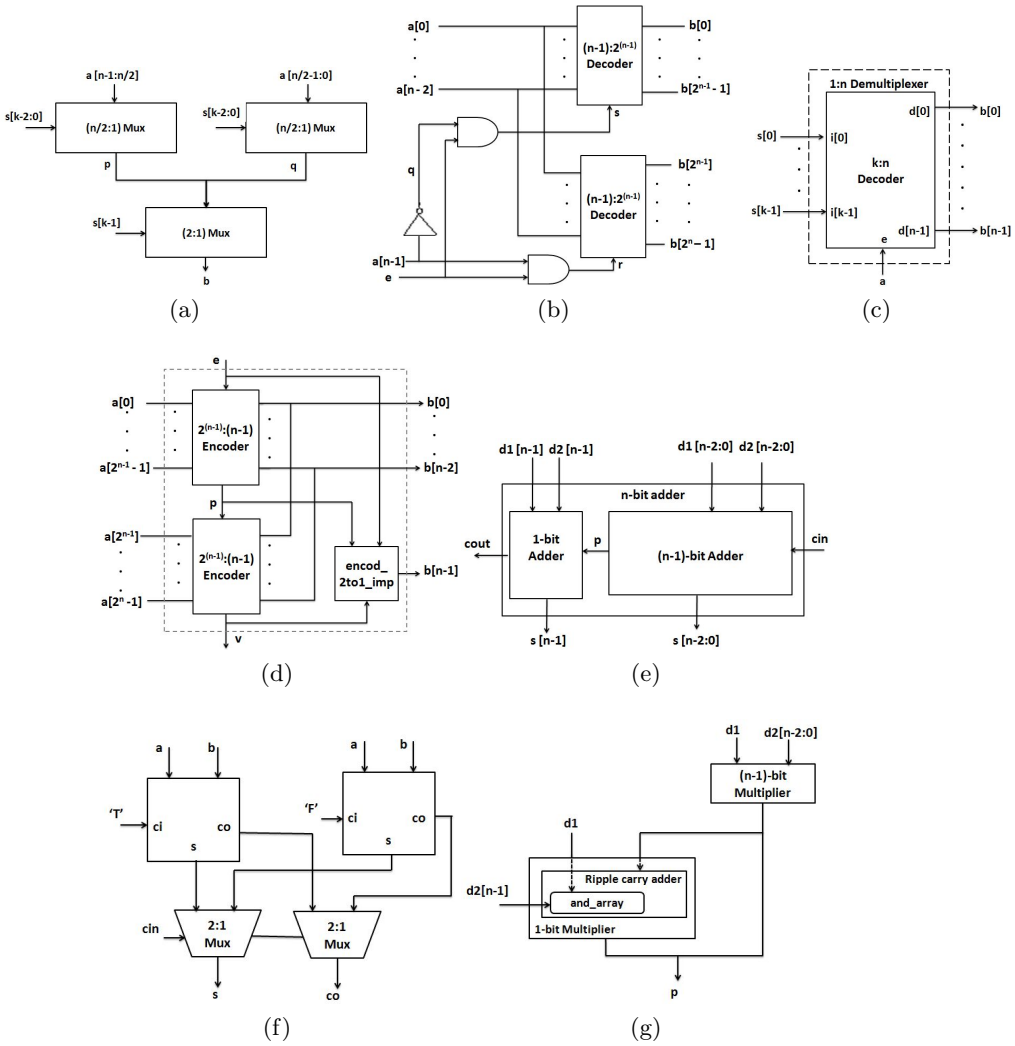


Fig. 1: Recursive Implementations (a) n:1 Mux (b) n:2^n Decoder (c) 1:n Demultiplexer (d) 2^n:n Encoder (e) n-bit Adder (f) 1-bit Carry Select Adder (g) n-bit Multiplier

3.3 Decoder

The recursive implementation of a $n : 2^n$ Decoder [20], shown in Fig. 1(b), is implemented using using two $(n-1) : 2^{(n-1)}$ Decoders having input of tail of the data input line, i.e., *a[n-2:0]*. Head of the data input line, i.e., *a[n-1]*, in conjunction with a global enable input e enables either of the two Decoders, which then sets the bits of the output signal depending upon the binary number represented by the input data vector. Here n is the width of the output data line and is used for the recursive implementation of the circuit. The relationship between the specification and implementation of the Decoder defined formally as `decod_imp_n n e a b` and `decod_spec_n n e a b` [27] is verified as:

Theorem 2: `⊢ ∀ n e a b. ((LENGTH b = n) ∧ (LENGTH b = 2 EXP LENGTH a)) ⇒ (decod_imp_n n e a b = decod_spec_n n e a b)`

where the assumptions ensure that the length of output data signal is equal to width of the Decoder and the relationship between the data input and the data output vectors. The proof script for Theorem 2 consists of about 1000 lines [26].

3.4 Demultiplexer

The functionality of Demultiplexer [20] is quite similar to that of the Decoder with the difference that Decoder sets one of the output lines depending upon the input signal while the Demultiplexer transmits the input data to one of the output lines depending upon the input select lines. Fig. 1(c) shows an implementation of the Demultiplexer using a Decoder, where the data input signal of the Demultiplexer, a, is connected to the enable signal of the Decoder, the select input signal of the Demultiplexer, s, is connected to the data input signal of Decoder and the data output signal of the Demultiplexer, b, is connected to the data output signal of Decoder. The relation between the width of select line k, and the width of the data output lines n is $k = \log_2 n$, or $n = 2^k$.

Theorem 3: `⊢ ∀ n a s b. ((LENGTH b = n) ∧ (LENGTH b = 2 EXP LENGTH s)) ⇒ (dmux_imp_n n a s b = dmux_spec_n n a s b)`

where the assumptions ensure that the length of output data vector is equal to the width of the Demultiplexer and relationship between the output data and the input select vectors. The proof of Theorem 3 is based on Theorem 2 and consists of only 20 lines of HOL code [26]. The less number of lines clearly show that existing formal components of the library greatly facilitate the formalization of new components.

3.5 Encoder

The Encoder [20] generates a binary output code for one bit of input True at a time. There are two discrepancies that may happen with the Encoders, i.e., the output behavior is non-deterministic in the case when more than one input bits

are True at a time or all input bits are zero. Priority Encoder [20] resolves these issues, by encoding output on the basis of priority and by using a valid output bit, respectively. Fig. 1(d) presents a recursive implementation of a $2^n : n$ Priority Encoder, using two $2^{n-1} : (n-1)$ Encoders, which encodes on the basis of the highest priority of the input signal, i.e., all other bits of the input data signal are ignored if the most significant bit of the data input signal is True. Where n specifies the width of the output data signal b, e is the enable input signal of the Encoder, p is connected with the valid output signal of the first Encoder and is used to enable the second Encoder, when the top half of the input data vector contains all False elements, v is the valid output signal, which indicates the validity of the encoded output data signal, and the function `encod_2to1_imp` computes the head of the output data signal using NOT, AND gates and a 2:1 Mux [26]. The relationship between the specification and implementation of the Encoder formally defined as `encod_spec_n n e a b v` and `encod_imp_n n e a b v` [27], is formally verified in HOL as following theorem.

Theorem 4: ⊢∀ `n e a b v.(LENGTH a = 2 EXP LENGTH b) ∧ (LENGTH b = n)`
`⇒ (encod_imp_n n e a b v = encod_spec_n n e a b v)`

where the assumptions ensure that the length of output data vector is equal to the width of the Encoder and relationship between the input data and the output data vectors. The proof script of Theorem 4 consists of about 900 lines of HOL code [26].

3.6 Ripple Carry Adder

A recursive implementation of n-bit Ripple Carry Adder [20] is shown in Fig. 1(e), where *d1* and *d2* are the two data input vectors which are required to be added, *cin* is the boolean carry input, *cout* is the boolean carry output and s is the sum output vector of the adder. One bit adder is implemented using the basic logic gates, i.e., XOR, AND and OR gates. [26]. Using 1-bit adder, the structure of the n-bit adder can be formalized as `adder_imp_n n d1 d2` [27]. The variable of recursion n specifies the width of the adder. The behavior can be formalized as `adder_spec_n n d1 d2 cin` [27]. The relationship between the implementation and specification is proved as a theorem, where the assumptions ensure that the lengths of both input vectors is equal to width of the adder. The proof script consists of about 2000 lines [26].

Theorem 5: ⊢ ∀ `n d1 d2 cin. (LENGTH d1 = n) ∧ (LENGTH d2 = n) ⇒`
`(adder_imp n d1 d2 cin = adder_spec_n n d1 d2 cin)`

3.7 Carry Select Adder

The formalization of the Carry Select Adder [20] is quite similar to that of the Ripple Carry Adder since both share the same recursive implementation, shown in Fig. 1(e). The main difference is the implementation of the 1-bit adder, which

is implemented using a Mux and full adder as shown in Fig. 1(f). The idea is to obtain the addition for 1-bit data using two full adders working in parallel for both cases of the carry input, i.e., 'T' and 'F'. The final values for sum and carry-out are chosen based on the input value of carry using a Mux. The formal definitions and theorems of the implementation and specification of the n-bit Carry Select Adder are almost same as for the Ripple Carry Adder. The proof script for the formal reasoning consists of about 200 lines of HOL code [26].

3.8 Multiplier

The recursive implementation of a n-bit Multiplier [20] is shown in Fig. 1(g), where each bit of the multiplicand, *d2*, is multiplied one-by-one with the multiplier *d1*, making partial products, which are then added using a Ripple Carry Adder. The 1-bit Multiplier is implemented using a Ripple Carry Adder and arrays of AND gates `and_array` [26]. Implementation and specification of n-bit multiplier formalized as `mult_imp d1 d2` and `mult_spec d1 d2` [27] are formally verified as following theorem

Theorem 6: $\vdash$ $\forall$ `d1 d2. mult_imp d1 d2 = mult_spec_n d1 d2`

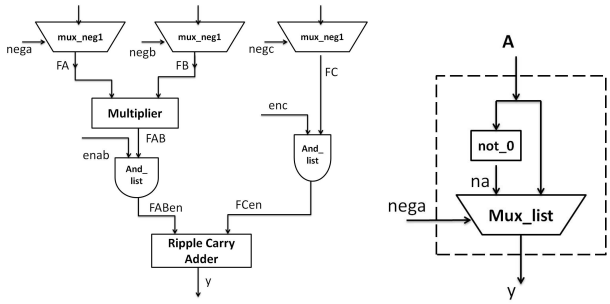
The proof script for Theorem 6 consists of about 2000 lines of HOL code [26].

The main advantage of the results presented in this section, i.e., the formal verification of the universally quantified theorems for the correctness of generic combinational circuits with arbitrary inputs, is the ability to use them for verifying a wide range of combinational circuits in a very straightforward manner. This benefit is attained at the cost of extensive user-effort spent in guiding the HOL theorem prover for verifying these theorems. The formalization, presented in this section, took around 7000 lines of HOL code and approximately 12 man-months [27]. These lines and effort include a number of general list and arithmetic theory proofs that are built upon to reason about Theorems 1 to 6. A significant amount of time was also spent on identifying the generic implementations of the common combinational circuits that can be expressed recursively as well.

4 Formal Verification of n-bit ALU

In this section, we use the library of formally verified combinational circuits, described in the previous section, to formally verify an n-bit arithmetic logic unit (ALU), shown in Fig. 2(a). It takes three n-bit inputs, a, b and c, which can be optionally inverted depending upon the signals *nega*, *negb* and *negc*. These signals along with other enable signals, *enab* and *enc*, generate different outputs of the ALU: $a.b$, $-a.b$, $a.b+c$, $a.b-c$, $-a.b+c$, $-a.b-c$, etc. This ALU has been recently formally verified for operand widths ranging form 4 to 256 bits taking 0.01 to 34.66 seconds [33]. We extend this work by formally verifying this ALU design for n-bit operands.

The first step is the formalization of the implementation of the given circuit, which can be defined using the pre-verified components of the library as follows:



(a) (b)

Fig. 2: Implementations (a) n-bit ALU (b) `mux_neg1_imp`

```
∀ n a b c nega negb negc enab enc y co.
alu n a b c nega negb negc enab enc y co =  ∃  FA FB FC FAB FABen FCen.
mux_neg1_imp a nega FA  ∧
mux_neg1_imp b negb FB   ∧
mux_neg1_imp c negc FC   ∧
mult_imp_n FA FB FAB   ∧
and_list_imp FAB enab FABen  ∧
and_list_imp FC enc FCen  ∧
Adder_imp_n (n + n) FABen (make_list_F n  ++ FCen) F y co
```

Where, the variable n represents the operand widths for variables a, b and c and co denote the carry out signal. The function `Adder_imp_n` and `mult_imp_n` are the formally verified ripple carry adder and multiplier of the generic library, respectively. The function `mux_neg1_imp` represents a combination of a multiplexer and a not gate, shown in Fig. 2(b), such that it allows to select between a given arbitrary-width input and its inverted signal depending upon the select signal *nega*. The implementation and specification of mux_neg1 is formally defined below:

Definition 1: *Implementation of mux_neg1*

```
⊢ ∀ a sel y. mux_neg1_imp a sel y <=>
          mux_list_imp (not_list a) a sel y
```

Definition 1a: *Implementation of mux_list*

```
⊢ ∀ a b sel.(mux_list_0 [] b sel= []) ∧ (mux_list_0 (h::t) b sel =
         mux_0_imp h (HD b) sel::mux_list_0 t (TL b) sel)
⊢ ∀ a b sel y. mux_list_imp a b sel y = (y = mux_list_0 a b sel)
```

where `not_list` returns the list by inverting all of its input data elements, the expression `mux_0_imp` is 2:1 mux, defined as `nand (nand in2 (not sel)) (nand`

sel in1) and the HOL function HD and TL returns the head and tail of the input list respectively. The behavior of the mux_list and mux_neg1 is formally defined as:

Definition 1b: *Specification of mux_list*

```
⊢ ∀ a b sel. mux_list_spec a b sel = if sel then a else b
```

Definition 2: *Specification of mux_neg1*

```
⊢ ∀ a sel y. mux_neg1_imp a sel y <=>
    if sel then y = (not_list a) else y = a
```

The relationship between the specification and implementation of the mux_neg1 is formally verified in HOL as the following theorem:

Theorem 7: *Formal Verification of mux_neg1*

```
⊢ ∀ a b sel y. (mux_neg1_imp a b sel y <=> mux_neg1_spec a b sel y)
```

Similarly, the component and_list is used for either transferring the input data list or list of all false elements. The implementation and specification of this component is formally defined below:

Definition 3: *Implementation of and_list*

```
⊢ ∀ a b sel. (and_list_0 [] en = []) ∧
(and_list_0 (h::t) en = (and [h;en]::(and_list_0 t en)))
⊢ ∀ a en out. and_list_imp a en out = (out = and_list_0 a en)
```

where the function and recursively performs the logical and between all the elements of a boolean list [27] and the function and_list_0 models a series of AND gates for performing the logical conjunction of a single bit signal *en* with all elements of input list *a* individually. The function and_list_imp represents an and_list component in the predicate form.

Definition 4: *Specification of and_list*

```
⊢ ∀ a en. and_list_spec a en = if (en) then a
          else make_list_F (LENGTH a)‘;
```

where the expression (make_list_F n) returns a list of n false elements [26]. The relationship between the specification and implementation of the and_list is formally verified as:

Theorem 8: *Formal Verification of and_list*

```
⊢ ∀ a en. (and_list_0 a en <=> and_list_spec a en)
```

The behaviour of the ALU is formally defined by carrying the binary subtraction using the 1's complement of the desired input, i.e., *BV_n (not_list a)*, where BV_n converts its argument boolean list into a number [27].

```
(if (enab) then
  if (enc) then
   if (~nega /\ ~negb /\ ~negc) then
    (co::y) = (num_BV_f(SUC (n + n))((BV_n a* BV_n b)+BV_n c))
    else if (~nega /\ ~negb /\ negc) then
    (co::y)=(num_BV_f(SUC (n + n))((BV_n a*BV_n b)+(BV_n (not_list c))))
    else if (~nega /\ negb /\ ~negc) then
    (co::y)=(num_BV_f(SUC (n + n))((BV_n a*(BV_n (not_list b)))+BV_n c))
    else if (~nega /\ negb /\ negc) then
    (co::y) = (num_BV_f(SUC (n + n))
     ((BV_n a*(BV_n (not_list b)))+(BV_n (not_list c))))
    else if (nega /\ ~negb /\ ~negc) then
    (co::y)=(num_BV_f(SUC (n + n))(((BV_n (not_list a))*BV_n b)+BV_n c))
    else if (nega /\ ~negb /\ negc) then
    (co::y) = (num_BV_f(SUC (n + n))
     (((BV_n (not_list a))*BV_n b)+(BV_n (not_list c))))
    else if (nega /\ negb /\ ~negc) then
    (co::y) =  (num_BV_f(SUC (n + n))
     (((BV_n (not_list a))*(BV_n (not_list b)))+BV_n c))
    else
    (co::y) = (num_BV_f (SUC (n + n))
     (((BV_n (not_list a))*(BV_n (not_list b))) + (BV_n (not_list c))))
  else
    if (~nega /\ ~negb) then ((co::y)  = (num_BV_f (SUC (n + n))
     (BV_n a * BV_n b)))
    else if (~nega /\ negb) then ((co::y)  = (num_BV_f (SUC (n + n))
         (BV_n a * (BV_n (not_list b)))))
    else if (nega /\ ~negb) then ((co::y)  = (num_BV_f (SUC (n + n))
        ((BV_n (not_list a)) * BV_n b)))
    else ((co::y)  = (num_BV_f (SUC (n + n))
     ((BV_n (not_list a))*(BV_n (not_list b)))))
else
if (~enc /\ ~negc) then ((co::y)  = (F::make_list_F (n+n)))
else if (~enc /\ negc) then ((co::y)  = (F::make_list_F (n+n)))
else if (enc /\ ~negc) then ((co::y)  =
     (num_BV_f (SUC (n + n)) (BV_n c)))
else ((co::y)  = (num_BV_f (SUC (n + n)) (BV_n (not_list c))))
```

where num_BV_f converts a number into a list with n booleans [27] and the expression SUC n represents the successor of the variable n. The equivalence between the formal implementation and specification of the given circuit is verified as the following theorem.

Theorem 9: *Formal Verification of n-bit ALU*

$\vdash \forall$ n a b c nega negb negc enab enc y co.
(LENGTH a = n) $\wedge$ (LENGTH b = n) $\wedge$ (LENGTH c = n) $\wedge$ n > 0 $\Rightarrow$

```
(ALU_n_imp n a b c nega negb negc enab enc y co <=>
ALU_n_spec n a b c nega negb negc enab enc y co)
```

where the assumptions ensure that the length of all input data vectors is equal to the width of the ALU and that should be greater than zero. The proof script of Theorem 9 is very straightforward and its first part is given below:

```
e (REPEAT STRIP_TAC THEN RW_TAC bool_ss [ALU_n_spec] THEN
RW_TAC std_ss[ALU_n_imp,AND_LIST_THM,and_list_spec,
and_list_imp,mult_imp_n,MULT_N_THM,mult_spec_n,
mux_neg1_thm,mux_neg1_spec,make_list_F,not_0,
LENGTH,Adder_imp_n,ADDER_RIPPLE_N,Adder_spec_n,
BV_n_make_list_F_a,LENGTH_make_list_F,
LENGTH_APPEND,BV,LENGTH_num_BV_f,
LENGTH_not_0,BV_n_make_list_F]);
```

The verification process mainly involves rewriting with the already verified theorems in a very straightforward manner involving very little user interaction. The first step is the removal of universal quantifiers using `STRIP_TAC`. This is followed by rewriting with the specification definition using `RW_TAC bool_ss`, which produces 16 subgoals depending upon the conditional statements used in the specification of the circuit. The verification of these 16 subgoals is not shown above due to space limitations but it involves simple rewriting with all definitions and theorems for the components of library used in the given subgoal using (RW_TAC std_ss). So we merely had to plug-in the definitions of the specifications and the names of the definitions and theorems for the components used in the subgoal to be verified in the rewriting tactics. The proof script is around 800 lines long and required about a couple of hours of development tme. Hence, use of the library of formally verified components made the verification process almost automatic, i.e., with very minimal user interaction. Moreover, it is important to note that based on this formally verified equivalence theorem with universally quantified input variables, we are able to verify corresponding equivalence relationships for any width size by appropriately instantiating Theorem 9, which clearly indicates the strength of the proposed methodology in verifying combinational circuits.

5 Conclusions

In this paper, we have presented our efforts in developing a framework for the formal verification of generic combinational circuits using a higher-order-logic theorem prover HOL4 while minimizing the user interactive efforts. The main idea is to develop a higher-order-logic library of all commonly used combinational circuits that includes their generic circuit implementations, their generic specifications and the proof of their equivalences. Since, these formalizations are done for arbitrary n-bit circuits so they can be used in turn to formalize and verify other n-bit combinational circuits. In this paper, we used this methodology to verify an n-bit ALU and the user effort in proof guidance was found to

be very little. Moreover, the user did not need to have an extensive knowledge of theorem proving for using this library. This ALU is now part of our library and can be further used to verify more complex blocks.

The proposed work opens the door to many interesting future directions of research. The formally verified library of circuits needs to be enhanced and advanced components like, Wallace Tree, Booth multipliers and components of floating-point arithmetic units may be added. More case studies for evaluation purposes are also underway. As long term goals, we plan to integrate a model checker with the proposed methodology to verify both combinational and sequential circuits within the same framework. Our work can also be combined with the recently proposed theorem proving-based analog circuit verification approach [29] to form a theorem proving-based Analog and Mixed Signal (AMS) circuit analysis framework.

Acknowledgements

This work was supported by the National Research Program for Universities grant (number 1543) of Higher Education Commission (HEC), Pakistan.

References

1. M.D. Aagaard and C.-J.H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *IEEE/ACM International Conference on Computer-aided Design*, pages 7–10. IEEE Computer Society, 1995.
2. Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Design Automation Conference*, pages 218–223. ACM, 2004.
3. C. Berg and C. Jacobi. Formal verification of the VAMP floating point unit. In *Correct Hardware Design and Verification Methods*, volume 2144, pages 325–339. Springer, 2001.
4. Christoph Berg, Christian Jacobi, and Daniel Kroening. Formal verification of a basic circuits library. In *IASTED International Conference on Applied Informatics. ACTA*. Press, 2001.
5. T. Braibant. Coquet: A Coq library for verifying hardware. In *Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 330–345. Springer, 2011.
6. P. Curzon, S. Tahar, and O. Ait-Mohamed. Verification of the MDG components library in HOL. In *Theorem Proving in Higher-Order Logics: Emerging Trends*, pages 31–46, 1998.
7. David Cyrluk, S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design - Theory, Practice and Experience, Second International Conference, TPCD '94, Bad Herrenalb, Germany*, pages 203–222, 1994.
8. D. Russinoff, M. Kaufmann, E. Smith, R. Sumners. Formal Verification of Floating-Point RTL at AMD using the ACL2 Theorem Prover. In *IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, 2005.
9. H. Deng. Formal verification of fpga based systems, 2011. MS Thesis, McMaster University, Canada.
10. E.M. Clarke and O. Grumberg and D. Peled. *Model Checking*. MIT press, 1999.

11. G.Al. Sammane, J. Schmaltz, D. Toma, P. Ostier and D. Borrione. TheoSim: Combining Symbolic Simulation and Theorem Proving for Hardware Verification. In *Integrated Circuits and System Design*, pages 60–65. ACM, 2004.
12. A. Camilleri, M. Gordon and T.F. Melham. Hardware verification using Higher-order Logic. Technical Report IUCAM-CL-TR-91, Computer Laboratory, University of Cambridge, Cambridge, UK, 1986. www.cl.cam.ac.uk/techreports/UCAM-CL-TR-91.pdf.
13. J. Harrison. Formal verification at Intel. In *Logic in Computer Science*, pages 45–54. IEEE Computer Society, 2003.
14. J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
15. O. Hasan and S. Tahar. Formal Verification Methods. *Encyclopedia of Information Science and Technology, IGI Global*, pages 7162–7170, 2015.
16. J. Sawada and E. Reeber. ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool. In *Formal Methods in Computer Aided Design*, pages 161–170. IEEE, 2006.
17. J. Sawada, P. Sandon, V. Paruthi, J. Baumgartner, M. Case and H. Mony. Hybrid Verification of a Hardware Modular Reduction Engine. In *Formal Methods in Computer-Aided Design*, pages 207–214, 2011.
18. R. Kaivola and M.D. Aagaard. Divider circuit verification with model checking and theorem proving. In *Theorem Proving in Higher Order Logics*, volume 1869, pages 338–355. Springer, 2000.
19. S. Kort, S. Tahar, and P. Curzon. Hierarchical formal verification using a hybrid tool. *Software Tools for Technology Transfer*, 4(3):313–322, 2003.
20. P.K. Lala. *Principles of Modern Digital Design*. Wiley, 2007.
21. L. Li, L. Liu, Y. Guan, Y. Zhang, J. Zhang, and L. Tao. A formal method for verifying the implementation of SPW data-strobe-encoding by applying theorem proving. In *SpaceWire Test and Verification*, pages 247–253, 2010.
22. C. Paulin-Mohring. Circuits as streams in coq: Verification of a sequential multiplier. In *Types for Proofs and Programs*, volume 1158, pages 216–230. Springer, 1996.
23. S. Ray and W. A. Hunt. Mechanized certification of secure hardware designs. In *Microprocessor Test and Verification, Common Challenges and Solutions*. IEEE Computer Society, 2007.
24. S. Ray and W. A. Hunt Jr. Deductive Verification of Pipelined Machines Using First-order Quantification. In *Computer-Aided Verification*, volume 3117 of LNCS, pages 31–43. Springer-Verlag, 2004.
25. C-JH. Seger, R.B. Jones, J.W. O'Leary, T. Melham, M.D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.
26. S. Shiraz. Automatic formal verification of generic combinational circuits, http://save.seecs.nust.edu.pk/projects/HLHV, 2018.
27. S. Shiraz and O. Hasan. A HOL library for hardware verification using theorem proving. *Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE*, 37(2):512–516, 2018.
28. P. Sule, Y. Kim, and N. Mansouri. PROVERIFIC: Experiments in employing (psl) standard assertions in theorem-proving-based verification. In *Midwest Symposium on Circuits and Systems*, pages 112–115, 2005.

29. S.H. Taqdees and O. Hasan. Formalization of laplace transform using the multivariable calculus theory of HOL-Light. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312, pages 744–758. Springer, 2013.
30. S. Tverdyshev. *Combination of Isabelle/HOL with Automatic Tools*, pages 302–309. Springer, 2005.
31. S. Tverdyshev. A verified platform for a gate-level electronic control unit. In *Formal Methods in Computer-Aided Design*, pages 164–171. IEEE, 2009.
32. W.A. Hunt, Jr. Verifying via Nano Microprocessor Components. In *Formal Methods in Computer-Aided Design*, pages 3–10, 2010.
33. C. Yu, W. Brown, and M. Ciesielski. Verification of arithmetic datapath designs using word-level approach: A case study. In *Circuits and Systems*, pages 1862–1865, 2015.