A Library for Combinational Circuit Verification using the HOL Theorem Prover

Sumayya Shiraz and Osman Hasan *Senior Member, IEEE*

***Abstract*—Interactive theorem provers can overcome the scalability limitations of model checking and automated theorem provers by verifying generic circuits and universally quantified properties but they require explicit user guidance, which makes them quite uninteresting for industry usage. As a first step to overcome these issues, this paper presents a formally verified library of commonly used combinational circuits using the higher-order logic theorem prover HOL4. This library can in turn be used to verify the structural view of any arbitrary combinational circuit against its behavior with very minimal user-guidance. For illustration, we verified several combinational circuits, including a 24-bit adder/subtractor, the 8-bit shifter module of the c3540 benchmark, the 17-bit EqualZ_W module of the c2670 benchmark, a 16:1 Multiplexer and a 512-bit Multiplier.**

I. Introduction

FORMAL verification [1] overcomes the limitations of simulation by mathematically proving the equivalence of a model against its desired properties. Many formal verification tools, based on model checking [1] and automated theorem proving techniques [1], are available that accept Verilog codes [2] and automatically translate them to the corresponding formal models and also automatically verify the relationship between the formal model and its corresponding specification given by the user. Thus, the verification engineer has to be involved in the formal specification of the properties only. This 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 but they have scalability issues. For example, 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.

Higher-order-logic theorem proving [1] can overcome these shortcomings. For example, PROVERIFIC [3] allows verifying Property Specification Language (PSL) assertions for hardware designs PVS. HOL has been used for the verification of the SPW Data-strobe (DS) encoding [4] and a gate level electronic control unit [5]. Similarly, Braibant [6] has created a library in Coq to facilitate modeling and verifying hardware circuits. However, in all these works, 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 interactive theorem proving in the mainstream hardware industry as the engineers prefer to have push-button type analysis tools.

The main scope of this paper is to facilitate the usage of interactive theorem proving for the verification of combinational circuits by minimizing the user involvement. In this regard, we present a library of formally verified generic (n-bit) circuits of commonly used components, like Multipliers, Adders, Multiplexers, Demultiplexers, Decoders, Encoders and logic gates. The main challenges in this library development include the identification of recursive implementations of the commonly used combinations circuits, formalization of their implementation and specification definitions and the interactive verification of the theorems that describe the equivalence between these generic implementations (i.e., with arbitrary length inputs) and specifications. Our formalization is primarily inspired by the formal hardware verification approach initially proposed by Camilleri, et. al [7].

The user of the proposed approach is required to 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 used interactive 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, which is just composed of some rewriting steps. Thus, the user of our methodology can leverage upon the strengths of interactive theorem proving without being involved in the extensive verification tasks. The effectiveness and utilization of the proposed methodology is illustrated by verifying a number of real-world combinational circuits automatically, like a 24-bit adder/subtractor, the 8-bit shifter of the c3540 benchmark, the 17-bit EqualZ_W module of the c2670 benchmark, a 16:1 Multiplexer using a 4:16 Decoder, a 512-bit Multiplier. We have used HOL4 [8] as our proof assistant due to its long term relationship with hardware verification [7] and the support that it provides to reason about Laplace [9] transform method, which is frequently used for analyzing analog circuits. Thus, the proposed library may also lead to the development of formal reasoning support for analog and mixed signal (AMS) circuits in future.

Shiraz and O. Hasan are with the School of Electrical Engineering and Computer Science (SEECS), National University of Sciences and Technology (NUST), Islamabad, Pakistan.
E-mail: {sumayya.shiraz, osman.hasan}@seecs.nust.edu.pk

II. Proposed Methodology

The proposed methodology, shown in Fig. 1, requires two inputs: (i) A higher-order-logic model of the structural connections of various components of the circuit that is needed to be verified, i.e., implementation, and (ii) the formal specification or the required behavior of the given circuit. The final output of our methodology is a formally verified equivalence relationship between these two models of the given circuit. It is important to note here that the main objective of our methodology is to obtain this result with very minimal user interaction. This has been achieved by developing a library of formal definitions and formally verified theorems corresponding to most of the commonly used combinational logic blocks, i.e., primitive circuits, including all logic gates, n-bit Ripple Carry Adder, n-bit Carry Select Adder, n-bit Multiplier, n:2^n Decoder, 2^n:n Encoder, n:1 Multiplexer and 1:n Demultiplexer by building upon the Boolean, Arithmetic and List theories of the HOL theorem prover. The unique feature of these primitive circuits is that they do not exhibit a behavior that can be modelled in terms of other circuits. The behavior (specification) of these primitive circuits is usually verified against their recursively defined implementation models. Moreover, all these definitions and theorems of the primitive circuits are generic and hence can be used for the formal verification of any type of the circuit irrespective of its size and complexity. The implementation models of the circuits to be verified are defined based on their structure and the implementation models of the primitive circuits and then the formally verified theorems, relating the implementations and specifications to the primitive circuits, can be used to verify the behavior of the given circuit almost automatically. It is important to note here that the verification of the primitive circuits required explicit user guidance. But once verified, the corresponding formally verified theorems facilitate the user in almost automatically verifying most of the combinational circuits that can be constructed in terms of these formally verified modules.

The first step of the proposed methodology is to develop a formal model of the structure of the combinational circuit, which is usually specified in a hardware description language (HDL), such as Verilog or VHDL, to be verified in terms of the formally verified generic components. Besides the formal model, we also require a formal specification, i.e., the behavioral description of the given circuit, which can be obtained from the given set of circuit requirements.

The second step is to formally verify that the given circuit is equivalent to the specification given by the user. In most of the cases, the verification is almost automatically done as the reasoning is based on simple rewriting with the definitions and theorems of the basic components involved in the given circuit under verification. The exceptions may happen when the specification is given as a complex arithmetic expression, which does not have a very straightforward relationship with the structure. In these cases, the user can provide proof guidance while interacting with the HOL user interface in the traditional interactive theorem proving style. However, these cases seldom arise and the approach concludes with very minimal user interaction in most of the cases, as will be illustrated by the case studies presented in Section IV of this paper. Finally, upon the successful verification, the user gets a formal proof of system properties, which specify that the given circuit's structure is equivalent to the given specification.

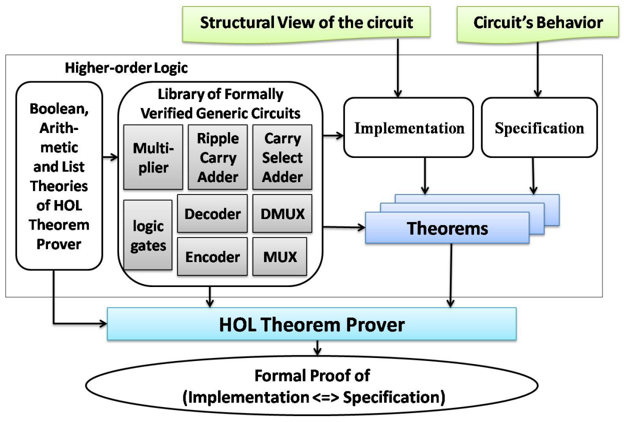

Structural View of the circuit
Circuit's Behavior
Higher-order Logic
Boolean, Arith-metic and List Theories of HOL Theorem Prover
Library of Formally Verified Generic Circuits
Multi-plier
Ripple Carry Adder
Carry Select Adder
logic gates
Decoder
DMUX
Encoder
MUX
Implementation
Specification
Theorems
HOL Theorem Prover
Formal Proof of (Implementation <=> Specification)

Fig. 1: Proposed Methodology

It is worth mentioning here that the structural and behavioral models are manually developed in the proposed approach and thus are susceptible to errors. But the chances of verifying wrong circuits are very low since the chances of making the corresponding mistakes in both the specification and implementation and thus verifying their equivalence are very less. What can practically happen is that either the specification or the implementation may have a translation bug but this can be caught during the formal verification phase and thus rectified.

III. Library of Generic Combinational Circuits

This section explains the formal definitions and verification of equivalent behaviours of the commonly used combinational components, mentioned in Fig. 1. This formalization, mainly inspired by the seminal work on digital circuit verification done at the University of Cambridge, UK [7], is the core component that allows the straightforward formal verification of generic combinational circuits. The main idea is to model generic (arbitrary-input) circuit diagrams of combinational circuits. i.e., circuit implementations, by using their logical components and their interconnections. The inputs and outputs of these circuits are modelled as lists of booleans to allow generic definitions. 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 manually verified using induction on the input variables. To the best of our knowledge, these generic relationships verified as theorems are not available in the literature.

A. *Logic Gates*

The inverter can be modeled as the following predicate [7]:

Definition 1: `⊢ ∀ a out. not a out = (out = ¬a)`

Similarly, the function `and` recursively performs the conjunction between all the elements of a boolean list. The function `and_n` represents an n-bit AND gate in the predicate form:

Definition 2:
```
⊢ and [] = T ∧
∀ h t. and (h::t) = (h ∧ (and t))
⊢ ∀ a out. and_n a out = (out = and a)
```

where *::* represents the list operation cons. The NAND gate can be formalized as `nand_n a out = (out = ¬(and a))`. Other logical gates have been similarly defined [10].

B. Multiplexer

The n:1 Multiplexer (Mux) [11] passes the signal of any one of the *n* input data lines to the one bit output line depending upon the k input select lines. Fig. 2 depicts the recursive implementation of a generic n:1 Mux, where *n* is the width of data input lines *a[n-1],..a[0]*, *k* is the width of select input lines *s[k-1],..s[0]* and *b* is a boolean output signal. The relation between the width of select and data input lines is $k = \log_2 n$.

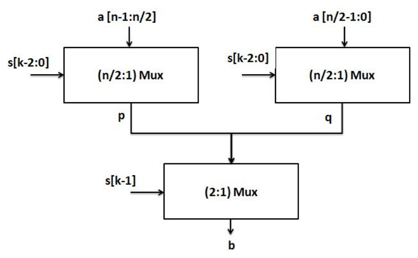


Fig. 2: Recursive Implementation of n:1 Mux

The 2:1 Mux can be formalized using basic logic gates:

Definition 3:
```
⊢∀ a b s y.mux_imp a b s y = ∃ p q r.
   nand_n [a;p] q ∧ nand_n [s;b] r ∧
   nand_n [q;r] y ∧ not s p
```

Definition 4:
```
⊢ ∀ a b s y. mux_spec a b s y =
    if s then (y = b) else (y = a)
```

and the relationship between Definitions 3 and 4 can be verified automatically. Next, we formalize the n:1 Mux as:

Definition 5:
```
⊢ ∀ a b.mux_imp_n a [] b = (b=HD a)∧
∀ a h t b. mux_imp_n a (h::t) b = ∃ p q.
    mux_imp q p h b ∧
    mux_imp_n (DROP (HALF a) a) t q ∧
    mux_imp_n (TAKE (HALF a) a) t p
```

where the function `HD` returns the head of the input list, the HOL expression `HALF a` returns half of the length of the given list a, i.e., `(LENGTH a) DIV 2`, and the expressions (`TAKE n a`) and (`DROP n a`) pick and drop the first *n* elements from the list *a*, starting from the first element, i.e., `HD a`, respectively. We define the behavior of the n:1 Mux as

Definition 6:
```
⊢ ∀ a s b. mux_spec_n a s b =
    (b = (EL (LENGTH a - 1 - BV_n s) a))
```

where `EL n a` returns the n^{th} element of its argument list and `BV_n` converts its argument boolean list into a number:

Definition 7:
```
⊢ BV_n [] = 0 ∧
∀h t.BV_n(h::t)=(2 EXP(LENGTH t))*BV h+BV_n t
```

The function `BV` converts a boolean variable to its corresponding number [7], [10], while considering the first element, i.e, head of the list, as the most significant bit. Next, we verify that the Mux implementation and specification are equivalent.

Theorem 1:
```
⊢ ∀ a s b. ¬(s = []) ∧
(LENGTH a = 2 EXP LENGTH s) ⇒
    (mux_imp_n a s b <=> mux_spec_n a s b)
```

The assumptions ensure that there is at least one select line and the relationship between the input and select lines. We proceed to verify this theorem by performing induction on *s* since this is the variable of recursion in our definition. The base case can be verified by applying induction on the variable *a* along with Definitions 5 and 6 and and some list theory functions. The subgoal, corresponding to the inductive case, after rewriting with the functions `mux_imp_n` and `mux_imp` becomes:

Subgoal 2.1:
```
A1:(∀ a b. ¬(h'::t = []) ∧
(LENGTH a = 2 EXP LENGTH (h'::t))⇒
mux_imp_n a(h'::t)b<=>mux_spec_n a(h'::t)b)
A2:(LENGTH a=2 EXP LENGTH(h::h'::t))
C: ∃ p q. mux_spec p q h b ∧
   mux_imp_n (DROP (HALF a) a) (h'::t) q ∧
   mux_imp_n (TAKE (HALF a) a) (h'::t) p
    <=> mux_spec_n a (h::h'::t) b
```

Now, the inductive hypothesis, i.e., Assumption A1 in the above subgoal, can be used to replace the implementation definitions of n:1 Mux in the conclusion (C) with their corresponding specification behaviors, which will result in a subgoal with specification definitions of Muxes only. Thereafter, rewriting with Definitions 4 and 6, we get two subgoals for the cases when the head of the select lines `h` is `F` and `T`.

Subgoal 2.2:
```
A1: (∀ a b.
(LENGTH a = 2 EXP LENGTH (h'::t))⇒
mux_imp_n a(h'::t)b<=>mux_spec_n a(h'::t) b)
A2: LENGTH a = 2 EXP LENGTH (h::h'::t)
A3: ∃p. mux_imp_n(TAKE(HALF a) a) (h'::t) p =
   mux_spec_n (TAKE (HALF a) a) (h'::t) p
A4: ∃q. mux_imp_n (DROP (HALF a) a) (h'::t) q =
   mux_spec_n (DROP (HALF a) a) (h'::t) q
C1: h ⇒(b=EL(BV_n(h'::t))(TAKE(HALF a) a))
    <=> (b = EL (BV_n (h::h'::t)) a)
C2: ¬h ⇒ (b=EL(BV_n(h'::t))(DROP(HALF a) a))
    <=> (b = EL (BV_n (h::h'::t)) a)
```

C1 of Subgoal 2.2 is verified using the following lemma:

Lemma 1:
```
⊢ ∀ a m n.((m < LENGTH a) ∧(n < m) ⇒
    (EL n a = EL n (TAKE m a))
```

Whereas the proof of C2 is based on the following lemmas:

Lemma 2:
```
⊢ ∀ a n m. (m < LENGTH a) ⇒
    (EL (n + m) a = EL n (DROP m a))
```

Lemma 3:
```
⊢ ∀ a. BV_n a < 2 EXP LENGTH a
```

The formal implementations, specifications and theorems for the other circuits of our library are given in Table I, where `num_BV_f` converts a number into a list with *n* booleans:

Definition 18:
```
⊢∀n a.num_BV_f n a = REV(num_BV n a)
⊢∀ a.num_BV 0 a = [] ∧ ∀n a.num_BV (SUC n) a=
  (num2bool (a MOD 2)::num_BV n (a DIV 2))
```

TABLE I: Formal Verification of Combinational Circuits

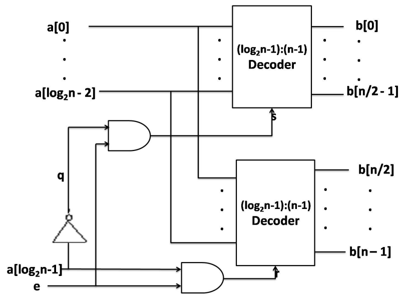
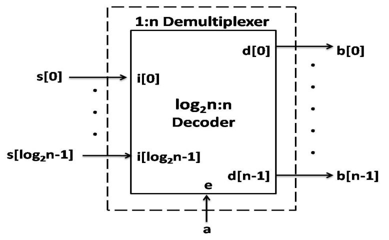
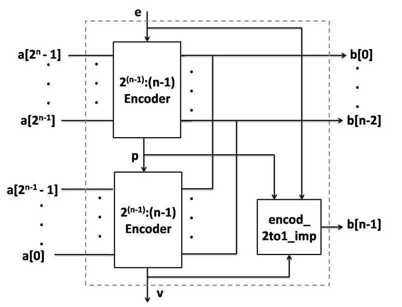
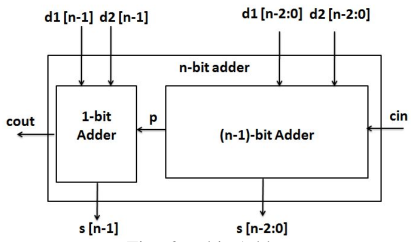
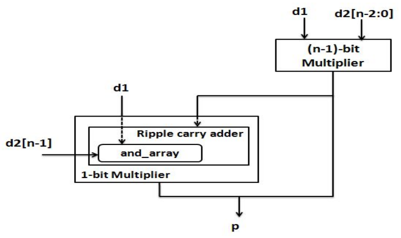
Recursive implementation of Combinational Circuits	Formal Definitions	Theorems	Proof Script
 a[0] a[log₂n - 2] q a[log₂n-1] e (log₂n-1):(n-1) Decoder s (log₂n-1):(n-1) Decoder r b[0] b[n/2 - 1] b[n/2] b[n – 1] Fig. 3: $\log_2$n:n Decoder	**Definition 8:** *Implementation of log_2n:n Decoder* `⊢ ∀ n e b. decod_imp_n n e [] b = if e then (HD b = T) else (BV_n b = 0) ∧ ∀ n e h t b. decod_imp_n n e (h::t) b = ∃ q r s. not h q ∧ and_n [e;q] s ∧ and_n [h;e] r ∧ decod_imp_n n s t (DROP (HALF b) b) ∧ decod_imp_n n r t (TAKE (HALF b) b)` **Definition 9:** *Specification of log_2n:n Decoder* `⊢ ∀ n e a b. decod_spec_n n e a b = if e then (b = num_BV_f n (2 EXP BV_n a)) else (b = num_BV_f n 0)`	**Theorem 2:** *Formal Verification of log_2n:n Decoder* `⊢ ∀ 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)`	1000 lines
 1:n Demultiplexer s[0] s[log₂n-1] i[0] i[log₂n-1] log₂n:n Decoder d[0] d[n-1] e a b[0] b[n-1] Fig. 4: 1:n Demultiplexer	**Definition 10:** *Implementation of 1:n Demux* `⊢ ∀ n a s b. dmux_imp_n n a s b = decod_imp_n n a s b` **Definition 11:** *Specification of 1:n Demux* `⊢ ∀ n a s b. dmux_spec_n n a s b = if a then (b = num_BV_f n (2 EXP BV_n s)) else (b = num_BV_f n 0)`	**Theorem 3:** *Formal Verification of 1:n Demux* `⊢ ∀ 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)`	20 lines
 e a[2ⁿ - 1] a[2ⁿ⁻¹] a[2ⁿ⁻¹ - 1] a[0] 2⁽ⁿ⁻¹⁾:(n-1) Encoder p 2⁽ⁿ⁻¹⁾:(n-1) Encoder encod_ 2to1_imp v b[0] b[n-2] b[n-1] Fig. 5: 2^n:n Encoder	**Definition 12:** *Implementation of 2^n:n Encoder* `⊢ ∀ n e a v.encod_imp_n n e a [] v = (if e then (if (HD a) then (v = F) else (v = T)) else (v = F)) ∧ ∀ n e a h t v.encod_imp_n n e a (h::t) v= ∃ p. encod_2to1_imp e p v h ∧ encod_imp_n n e (TAKE (HALF a) a) t p ∧ encod_imp_n n p (DROP (HALF a) a) t v` **Definition 13:** *Specification of 2^n:n Encoder* `⊢ ∀ n e b v. encod_spec_n n e [] b v = if e then (v = T) else (v = F) ∧ ∀ n e h t b v. encod_spec_n n e (h::t) b v = if e then (if h then ((b=num_BV_f n (LENGTH t))∧ (v=F)) else encod_spec_n n e t b v) else (v=F)`	**Theorem 4:** *Formal Verification of 2^n:n Encoder* `⊢ ∀ 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)`	900 lines
 d1 [n-1] d2 [n-1] d1 [n-2:0] d2 [n-2:0] n-bit adder cout 1-bit Adder p (n-1)-bit Adder cin s [n-1] s [n-2:0] Fig. 6: n-bit Adder	**Definition 14:** *Implementation of n-bit Ripple Carry Adder* `⊢∀d1 d2 cin.adder_imp 0 d1 d2 cin=[cin]∧ ∀n d1 d2 cin.adder_imp (SUC n) d1 d2 cin= (adder_imp_1 (HD d1) (HD d2) HD (adder_imp n (TL d1) (TL d2) cin) ++ (TL (adder_imp n (TL d1) (TL d2) cin)))` `⊢ ∀ n d1 d2 cin s cout. adder_imp_n n d1 d2 cin s cout = ((cout::s) = adder_imp n d1 d2 cin)` **Definition 15:** *Specification of n-bit Ripple Carry Adder* `⊢∀n d1 d2 cin. adder_spec_n n d1 d2 cin = num_BV_f (SUC n) (BV_n d1+BV_n d2+BV cin)`	**Theorem 5:** *Formal Verification of n-bit Ripple Carry Adder* `⊢ ∀ n d1 d2 cin. (LENGTH d1 = n) ∧ (LENGTH d2 = n) ⇒ (adder_imp n d1 d2 cin <=> adder_spec_n n d1 d2 cin)`	2000 lines
 d1 d2[n-2:0] (n-1)-bit Multiplier d1 d2[n-1] Ripple carry adder and_array 1-bit Multiplier p Fig. 7: n-bit Multiplier	**Definition 16:** *Implementation of n-bit Multiplier* `⊢ ∀ d1. mult_imp d1 [] = make_list_F (LENGTH d1) ∧ ∀d1 h t.mult_imp d1 (h::t)=(mult_imp_1 d1 (TAKE (LENGTH d1) (mult_imp d1 t)) h)++ (DROP (LENGTH d1) (mult_imp d1 t))` `⊢ ∀ a b p. mult_imp_n a b p = (p = mult_imp a b)` **Definition 17:** *Specification of n-bit Multiplier* `⊢ ∀ d1 d2. mult_spec_n d1 d2 = (num_BV_f (LENGTH d1 + LENGTH d2) (BV_n d1 * BV_n d2))`	**Theorem 6:** *Formal Verification of n-bit Multiplier* `⊢ ∀ d1 d2. mult_imp d1 d2 <=> mult_spec_n d1 d2`	2000 lines

TABLE II: Results of Case Studies

Circuit	**Bench-mark**	**Inputs**	**Gates**	**Verification Time (s)**
24-bit Adder/sub	-	49	144	01
8-bit Shifter	c3540	29	201	60
16:1 Mux	-	20	62	03
17-bit EqualZ_W	c2670	35	25	0.3
512-bit Multiplier	-	1024	1572864	126

where `REV` reverses the order of the given list, the expression `SUC n` represents the successor of the variable *n* and `num2bool` converts a number to its corresponding boolean value [10]. The function `encod_2to1_imp`, in the implementation definition of Encoder, evaluates the head of the output data signal. The expression (`make_list_F n`) returns a list of *n* false elements [10] and the function `mult_imp_1` implements a 1-bit Multiplier using a Ripple Carry Adder [10].

It can be observed from the lines of code, given in Table I, that the verification effort for some of the circuits, like the decoder, encoder, adder and multiplier, involved very complex reasoning compared to the one used to verify the n:1 Mux above. These efforts include the verification of 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 recursive implementations of the common combinational circuits. Moreover, the proof sketches of the theorems could not be obtained in any text and were developed as part of the reported work as well. 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 automatically verifying a wide range of combinational circuits, as depicted in Fig. 1.

IV. Case Studies

We used the proposed methodology to formally verify a number of circuits, given in Table II, using a Ubuntu workstation with Corei5-2320 processor operating at 3GHz with 4GB memory. The verification was almost automatically done in less than a couple of minutes using the library of the building blocks. The accuracy and automation in verification are the main benefits provided by our proposed methodology. For illustration, we explain the verification of a 24-bit adder/subtractor, which acts as a binary adder or subtractor depending upon the signal *sel*. We formalize its structure as:

```
∀ a23..a0 b23..b0 sel y23..y0 co.
Add_Sub_24_imp a23..a0 b23..b0 sel y23..y0 co
= ∃ xb23...xb0. xor_n [b23;sel] xb23 ∧
xor_n [b22;sel] xb22∧ ..... xor_n [b0;sel] xb0 ∧
Adder_imp_n (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC
(SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC
(SUC (SUC (SUC (SUC 0))))))))))))))))))))))))
[a23;a22;....a1;a0] [xb23;xb22;...xb1;xb0]
sel [y23;y22...y1;y0] co
```

The specification is modeled using subtraction by 2's complement:

```
∀ a23...a0 b23...b0 sel y23...y0 co.
Add_Sub_24_spec a23...a0 b23...b0 sel y23...y0 co =
if sel then ([co;y23;y22..y1;y0]=num_BV_f (SUC (SUC (SUC (SUC
(SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC
(SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC 0
)))))))))))))))))))))))))
(BV_n [a23;a22;..a1;a0] + (BV_n [¬b23;¬b22;...¬b1;¬b0] + 1))
else ([co;y23;y22..y1;y0]=num_BV_f (SUC (SUC (SUC (SUC (SUC
(SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC
(SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC (SUC 0
)))))))))))))))))))))))))
(BV_n [a23;a22;..a1;a0] + (BV_n [b23;b22;..b1;b0]))))
```

Their equivalence is then verified using the following script:

```
REPEAT STRIP_TAC THEN RW_TAC bool_ss [Add_Sub_24_spec] THEN
RW_TAC std_ss[Add_Sub_24_imp,Adder_imp_n,LENGTH,
  ADDER_RIPPLE_N,Adder_spec_n,xor_n,xor,BV,ADD_ASSOC]
```

`REPEAT STRIP_TAC` removes all the universal quantifiers. Then, the specification is rewritten using (`RW_TAC bool_ss`), which creates 2 subgoals corresponding to `(sel= T)` and `(sel = F)`. Next, we rewrite and simplify (`RW_TAC std_ss`) both the subgoals with all the definitions and theorems of the components used in the circuit. The same steps can be followed to verify any given circuit and the user has to just use the appropriate definitions and theorems for the given circuit and its components in the tactics.

V. Conclusions

The paper presents a methodology to minimize user intervention in the formal equivalence verification between the behavior and implementation of a combinational circuit using a higher-order-logic theorem prover. We use the formal structural and behavioral descriptions of commonly used n-bit circuits and their equivalence verification for this purpose, which is the foremost contribution of this paper.

Acknowledgement

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

References

[1] O. Hasan and S. Tahar, "Formal verification methods," *Encyclopedia of Information Science and Technology, IGI Global*, pp. 7162–7170, 2015.

[2] Z. S. Andraus and K. A. Sakallah, "Automatic abstraction and verification of verilog models," in *Design Automation Conference*. ACM, 2004, pp. 218–223.

[3] 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*, 2005, pp. 112–115.

[4] 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*, 2010.

[5] S. Tverdyshev, "A verified platform for a gate-level electronic control unit," in *Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 164–171.

[6] T. Braibant, "Coquet: A Coq library for verifying hardware," in *Certified Programs and Proofs*, ser. LNCS, vol. 7086. Springer, 2011, pp. 330–345.

[7] A. C. M. Gordon and T. Melham, "Hardware Verification using Higher-order Logic," Computer Laboratory, University of Cambridge, Cambridge, UK, Tech. Rep. IUCAM-CL-TR-91, 1986, www.cl.cam.ac.uk/techreports/UCAM-CL-TR-91.pdf.

[8] K. Slind and M. Norrish, "A brief overview of HOL4," in *Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 5170. Springer, 2008, pp. 28–32.

[9] S. Taqdees and O. Hasan, "Formalization of laplace transform using the multivariable calculus theory of HOL-Light," in *Logic for Programming, Artificial Intelligence, and Reasoning*, vol. 8312. Springer, 2013, pp. 744–758.

[10] S. Shiraz, "Automatic Formal Verification of Generic Combinational Circuits, http://save.seecs.nust.edu.pk/projects/HLHV," 2017.

[11] P. Lala, *Principles of Modern Digital Design*. Wiley, 2007.