Logical Equivalence Checking of Asynchronous Circuits Using Commercial Tools

Arash Saifhashemi
Intel Corporation
Santa Clara, CA
Email: saifhash@usc.edu

Hsin-Ho Huang
Electrical Engineering
University of Southern California
Los Angeles, CA
Email: hsinhohu@usc.edu

Priyanka Bhalerao
Yahoo Corporation
Sunnyvale, CA
Email: pbhalera@usc.edu

Peter A. Beerel*
Electrical Engineering
University of Southern California
Los Angeles, CA
Email: pabeerel@usc.edu

***Abstract*—We propose a method for logical equivalence check (LEC) of asynchronous circuits using commercial synchronous tools. In particular, we verify the equivalence of asynchronous circuits which are modeled at the CSP-level in SystemVerilog as well as circuits modeled at the micro-architectural level using conditional communication library primitives. Our approach is based on a novel three-valued logic model that abstracts the detailed handshaking protocol and is thus agnostic to different gate-level implementations, making it applicable to a variety of different design styles. Our experimental results with commercial LEC tools on a variety of computational blocks and an asynchronous microprocessor demonstrate the applicability and limitations of the proposed approach.**

I. Introduction and Related Work

Logical equivalence checking has become an accepted requirement of commercial synchronous design flows to quickly find bugs and add confidence in the taped-out netlist. However, asynchronous design flows still rely on extensive dynamic simulation with unique test-bench and coverage tools to show functional correctness, increasing risk and hampering the widespread adoption of this technology [1].

The design of asynchronous circuits often starts from a high-level specification based on Communicating Sequential Processes (CSP) [2] or one of its variants, such as [3]. The high-level description is *decomposed* into smaller communicating processes until each process is sufficiently small or regular to be easily mapped to an existing library of specialized asynchronous leaf cells and/or macros [4]–[7]. A netlist of such small CSP cells is called an *image netlist*. These leaf cells can then be implemented with a variety of asynchronous pipeline templates. Moreover, recently power optimization techniques have been proposed that involve adding conditional communication into the decomposition to reduce token flow [8] as well as optimizing the location of conditional communication cells within the design [9]. This paper provides a formal framework for the logical equivalent verification of such manual and automatic decompositions and optimizations.

Existing research efforts in formal verification of asynchronous circuits have been divided into three broad camps: *hazard-freedom or conformance checking*, *equivalence checking*, and *property verification*. In particular, many techniques try to verify hazard-freedom/conformance of a gate-level implementation of a leaf-level CSP cell, a unique requirement to asynchronous design [10], [11]. Others target notions of equivalence between the two designs [12]–[14]. These approaches generally cannot be used to compare CSP with decomposed versions because the decomposition often introduces pipelining that changes the allowed sequence of events at the external interface. Therefore, some researchers only check critical properties on the final decomposed design [15], [16].

Our proposed approach is different from the previous work in the following ways: first, since it is focused on CSP-level designs, it is implementation-agnostic and can be used for design flows that target various asynchronous templates. Secondly, compared to [11], we explicitly support modules that have channels with conditional communication. Thirdly, our flow is highly based on commercial synchronous LEC tools which can validate moderate-sized and complex circuits in a reasonable time and will likely continue to improve over time.

The remainder of this paper is organized as follows. In Section II, we introduce our abstract 3VL model of asynchronous circuits and our definition of equivalence. Section III describes how we model SVC designs and discusses the limitation and scope of our approach. Section IV describes how we convert this 3VL model into a 2VL model suitable for commercial equivalence checkers. Section V presents our experimental results and is followed by our conclusions in Section VI.

II. The Three-Valued Logic Model

An asynchronous system consists of a netlist of CSP processes which communicate with each other via asynchronous channels. We assume the behavior of each process can be described in terms of *iterations* during which it repeatedly performs the following three actions:

1) It receives from none, some, or all input channels.
2) It performs calculations.
3) It sends to none, some, or all output channels.

Unlike synchronous circuits where each primary input/output has a Boolean value of 0 or 1, asynchronous channels may either not communicate at all, or communicate a value of 0 or 1. We thus model the behavior of CSP netlists using three-valued (3VL) logic that is based on the more general notion of multi-valued [17]. In particular, 3V logic variables can take a value from the set $\mathcal{T} = \{0, 1, N\}$ where the value N models the condition of *no communication action* on a channel.[1]

[1]The value N should not be confused by the traditional concept of a *don't care*.

∧		A = 0	A = 1	A = N
B	0	0	0	N
	1	0	1	N
	N	N	N	N

(a) $A \wedge B$ (AND)

∨		A = 0	A = 1	A = N
B	0	0	1	N
	1	1	1	N
	N	N	N	N

(b) $A \vee B$ (OR)

≡		A = 0	A = 1	A = N
B	0	1	0	0
	1	0	1	0
	N	0	0	1

(c) $A \equiv B$ (EQUIVALENCE)

		¬ A
A	0	1
	1	0
	N	N

(d) $\neg A$ (NOT)

Ⓡ		E = 0	E = 1	E = N
L	0	0	0	N
	1	0	1	N
	N	0	N	N

(e) LⓇE (RECEIVE)

Ⓢ		E = 0	E = 1	E = N
L	0	N	0	N
	1	N	1	N
	N	N	N	N

(f) LⓈE (SEND)

Figure 1: Primitive functions in three-value logic

Basic 2VL functions and operators can be defined based on 3V variables as shown in Figure 1. The $\wedge$ and $\vee$ operators represent logical functions similar to Boolean logical AND and OR functions as long as none of the inputs is N, otherwise the output of these functions is N. The inverting operator $\neg$ is the same as the Boolean inverter when the input is *0* or *1*, but its output is N its input is N. The output of the equivalence operator $\equiv$ is always *0* or *1* and never *N*.The RECEIVE and SEND operators (denoted Ⓡ and Ⓢ) describe conditional communication behavior and are described more fully below.

We model the system iteration-based behavior using a 3VL finite state machine (FSM) as defined below.

Definition 1 (3VL Finite state machine): A 3VL finite state machine is a 6-tuple:

$$(\Sigma, S, \delta, S_0, \Gamma, \lambda),$$

where:

- Σ is a finite non-empty set of input minterms.
- S is a finite non-empty set of states.
- $\delta : S \times \Sigma \rightarrow S$ is the next state function.
- $S_0 \subseteq S$ is the set of initial states.
- Γ is a finite non-empty set of output minterms.
- $\lambda : S \times \Sigma \rightarrow \Gamma$ is the output function.

A unique aspect of our 3VL FSM is that state variables can be defined as *persistent*. If a persistent state variable starts from a non-N value, it will never update to *N*. Persistent state variables (P-states) are similar to flip-flops in synchronous circuits whose clock in a given cycle is *gated* and hence the flip-flop will not update its state.

Two FSMs are *equivalent* if starting from their respective initial states, they will produce the same output sequence when they are given the same input sequence [18]. In Boolean logic, if two sequential circuits share the same set of inputs, outputs, and state-holding elements (e.g., flip-flops), then it can be shown that it is sufficient to check their combinational portions for equivalence [19]. To extend this notion to 3V FSMs with our notion of persistent states, we include a multiplexer at the output of the combinational logic for each persistent state to update the state only if the input is not *N*, as shown in Figure 2 as *n* P-states. Therefore, if the initial value of a persistent state is non-*N*, it will never update to *N* because of the multiplexer.

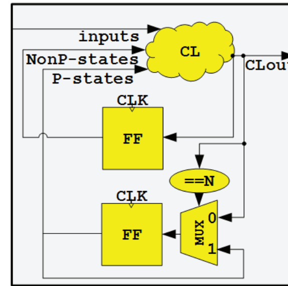

Figure 2: 3VL FSM

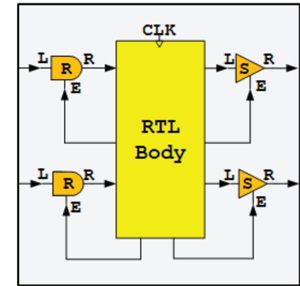

Figure 3: Decomposed SVC

III. Decomposed SVC-based Approach

The first step of our verification approach is to automatically decompose our SVC-based designs into a special form that explicitly exposes the conditional communication and state variables, referred to as an image RTL [7]. This form naturally defines the 3VL FSM by decomposing every SVC module into a clock-driven *RTL-Body* wrapped in SEND and RECEIVE primitives as shown in Figure 3.

The SVC description of the RTL-Body contains all the logic including state variables of the original SVC module with additional logic to control the enable signals of the RECEIVE/SEND modules. The SEND and RECEIVE are asynchronous modules modeled by the 3VL operators Ⓡ and Ⓢ defined in Figure 1. In particular, in each iteration, based on the value received on *E*, the RECEIVE primitive may or may not receive from *L*, but it always sends a value on *R*. The SEND primitive always receives from *L*, but based on the value received from *E*, it may or may not send on *R*. Since we moved all conditional communication actions into the surrounding RECEIVE/SEND modules, the RTL-Body becomes an unconditional module, i.e., at each iteration it unconditionally receives on all inputs and unconditionally sends on all outputs, therefore, the transformation of the RTL-Body into a synthesizable and synchronous FSM is straightforward [7].

Each iteration of the SVC description is thus conceptually mapped into one clock cycle of the RTL image. If there is no communication action on a channel *C* at iteration the value of the corresponding variable *C* in the image RTL at clock cycle *i* is *N*. This effectively translates our verification problem into the synchronous domain.

We do not verify the low-level handshaking controllers, but we only verify the equivalence at the CSP-level using the 3VL model. Verifying the handshaking controllers is not within the scope of this paper. In particular, by comparing the 3VL

model of SVC descriptions, we check the logical equivalence assuming the handshaking circuitry used to implement the asynchronous communication between modules is functioning correctly. In addition, our 3VL model cannot emulate the flow-control nature of asynchronous processes, therefore, it cannot capture behaviors in which tokens are stalled at the inputs of asynchronous blocks across iterations. This limitation is formalized as *iteration stall freedom (ISF)* in [20]. We refer to the systems that do not have this complication as Primary Input ISF (PIISF) to emphasize the fact that the stall-freedom requirement is limited to primary inputs; that is, tokens propagated via state-holding elements are allowed to stall because the state-holding elements are able to capture their values across iterations. In practice, a large class of asynchronous systems, including an asynchronous CPU and its decomposition described in our experimental results section, are PIISF.

IV. Binary Coded Three Valued Logic (BC3VL)

In order to use commercial Boolean equivalence checking tools, the next step of our verification procedure is to encode each variable of a 3VL FSM using two bits: *valid (v)* and *data (d)*.

Each DFF is duplicated: one for the *data* bit and one for *valid* the bit. The valid and data bit outputs of every other library cell are a function of valid and data bits of their inputs as defined below.

- A gate implementing an unconditional function $o = f(i_1, i_2, \ldots, i_n)$:

$$o.v = i_1.v \wedge i_2.v \wedge \cdots \wedge i_n.v$$
$$o.d = \boldsymbol{o.v} \wedge f(i_1.d, i_2.d, \ldots, i_n.d)$$

- A state variable function: $s = f(i_1, i_2, \ldots, i_n)$:

non-persistent:

$$s.v = i_1.v \wedge i_2.v \wedge \cdots \wedge i_n.v$$

persistent:

$$s.v = 1$$
$$s.d = \boldsymbol{s.v} \wedge f(i_1.d, i_2.d, \ldots, i_n.d)$$

- A RECEIVE cell:

$$r.v = (e.v \wedge \neg e.d) \vee (l.v \wedge e.v \wedge e.d)$$
$$r.d = \boldsymbol{r.v} \wedge l.d \wedge e.d$$

- A SEND cell:

$$r.v = l.v \wedge e.v \wedge e.d$$
$$r.d = \boldsymbol{r.v} \wedge l.d$$

We transform the RTL-Body into a BC3VL RTL using SystemVerilog preprocessor macros to add declarations for valid bits for each primary input, primary output, and state variables such that the resulting description can be used as the input of commercial Boolean equivalence checkers. In particular, for each primary input (output) i (o), a new one-bit primary input i_v (o_v) is added. Further, for each state variable s, a new one-bit state variable s_v is added. The value of each added state variable is set to *1* upon reset, since our SVC semantics requires that all state variables have non-N (i.e. valid) values upon reset [20]. The values of o_v and s_v, on the other hand, depend on the valid bit of primary inputs and other state variables. If the state variable is persistent, however, the value of s_v will always be *1*. In particular, since the RTL-Body is unconditional, the valid bit of each primary output o is the AND of the valid bits of all inputs in the *support* (i.e., the set of variables on which o actually depends) of o. For example, for an ALU with primary inputs *I1*, *I2*, and *OP*, the valid bit of the primary output *O* should be added to the RTL-Body description as:

$$O.v = I_1.v \wedge I_2.v \wedge OP.v$$

Our tool, called *SVC23VLRTL* estimates the support functions for each primary output automatically. When it parses the SVC code, it records all inputs of the statements or assignments as support functions of their outputs. After parsing, it flattens all support functions, leaving only primary inputs and state variables. This support set may overestimate that of some possible implementations which can cause a false failure during equivalence checking. For this reason SVC23VLRTL also enables the support set estimate to be overridden by the user.

V. Experimental Results

To evaluate the effectiveness of the proposed approach, we first examine its application on small computational blocks and then provide a case study of verifying the top-level decomposition of an asynchronous microprocessor whose design is inspired by the seminal work of Martin [3].

We tested our proposed BC3VL method on several computational modules comparing different levels of their design:

1) **ALU**: A four-mode hierarchical arithmetic unit that implements AND, ADD, SUB, and MULT operations.
2) **ADDMULT**: An adder/multiplier dual-mode arithmetic unit. Using conditional communication, in each mode, data is only sent to the sub-unit that is performing useful calculation.
3) **CONDMULT**: A multiplier that conditionally receives data and computes controlled by the value received on an unconditional enable input channel.
4) **LCM**: A block that calculates the least common multiple of two inputs using Euclid's iterative algorithm. This block has state variables.

Table I shows more detailed information of these circuits.

Example	Data Width	# of Inputs	# of Outputs	# of Gates
ALU	32	2	1	2100
ADDMULT	32	5	2	5500
CONDMULT	32	3	1	2500
LCM	16	2	1	8400

Table I: LEC example statistics

For the ALU, we first compared a 32-bit ALU described at the SVC level with a decomposed implementation. The

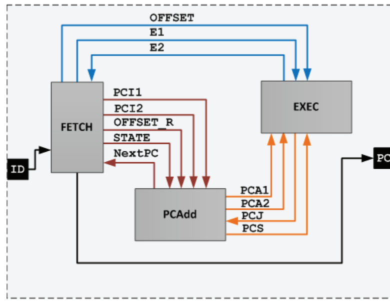

Figure 4: Top-level decomposition of the CPU

two ALU designs were given to the Cadence Conformal verification tool and were declared to be equivalent. Secondly, we compared the equivalence of these designs after each was fully synthesized into image library cells. Further, we perform automatic decomposition based on operand isolation [8] on the ALU design for power optimization and confirmed the result with the original ALU and the decomposed ALU image netlist.

For the ADDMULT, CONDMULT and LCM examples, we also verified the results at image netlist level before and after *reconditioning*, an automatic power optimization technique which moves logic through conditional communication primitives to save power [20]. Table II shows the resulting CPU run times for all these comparisons. Note that we choose a 16 bit for the LCM circuit because verification of the 32 bit LCM exceeded our 12 hour timeout limit.

Golden	Revised	Level	Run-Time (s)
ALU	Decomposed	SVC	1.64
ALU	Decomposed	Image Netlist	77.76
ALU	Operand Isolation	Image Netlist	3.42
ALU Operand Isolation	Decomposed	Image Netlist	77.62
ADDMULT	Reconditioning	Image Netlist	14.47
CONDMULT	Reconditioning	Image Netlist	5.82
LCM	Reconditioning	Image Netlist	386

Table II: LEC run times

As a final case study, we re-implemented Caltech's first asynchronous microprocessor [3] in SVC. Guided by the known limitations of the formal verification process, we ensured our decomposition and top-level implementation have one-to-one correspondence of state variables. This means that instead of having additional local state variables, we choose to synchronize computation through message passing on additional channels, as illustrated in Figure 4. Interestingly, our initial decomposition falsely failed to conform to the top-level implementation because of some unreachable states of the design. However, by adding constraints to the source Verilog to tell the verification tool to not consider these unreachable states, a common practice in synchronous LEC flows [21], the tool proved equivalence in 31 seconds.

VI. Conclusions

We presented a new formal approach for verifying the correctness of three key steps in asynchronous design flows: architectural decomposition, synthesis, and micro-architectural optimizations. We used three-valued-logic to model conditional communication and translate the design to a binary representation to employ the use of commercial synchronous equivalence checkers. Our experimental results demonstrate that while some limitations exist, the approach is powerful enough to enable push-button verification of moderate-size computational blocks and flexible enough to verify more complex decompositions with reasonable manual intervention.

References

[1] A. Yakovlev, P. Vivet, and M. Renaudin, "Advances in Asynchronous Logic: From Principles to GALS & NoC, Recent Industry Applications, and Commercial CAD Tools," in *DATE*, 2013, pp. 1715–1724.

[2] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[3] A. Martin, "Synthesis of Asynchronous VLSI Circuits," California Institute of Technology, Department of Computer Science, Tech. Rep. CS-TR-93-28, March 1991.

[4] D. Edwards and A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[5] C. G. Wong and A. J. Martin, "High-level Synthesis of Asynchronous Systems by Data-driven Decomposition," in *DAC*, 2003, pp. 508–513.

[6] Y. Thonnart, E. Beigne, and P. Vivet, "A Pseudo-Synchronous Implementation Flow for WCHB QDI Asynchronous Circuits," in *ASYNC*, 2012, pp. 73–80.

[7] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design and test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[8] "Observability Conditions and Automatic Operand-Isolation in High-Throughput Asynchronous Pipelines," in *Integrated Circuit and System Design, PATMOS*, ser. Lecture Notes in Computer Science, 2013, vol. 7606.

[9] A. Saifhashemi, H.-H. Huang, and P. Beerel, "Reconditioning: Automatic power optimization of QDI circuits," in *ASYNC*, May 2014.

[10] C. Nelson, C. Myers, and T. Yoneda, "Efficient Verification of Hazard-Freedom in Gate-Level Timed Asynchronous Circuits," *IEEE Transactions on CAD*, vol. 26, no. 3, pp. 592–605, 2007.

[11] S. Longfield and R. Manohar, "Inverting Martin Synthesis for Verification," in *ASYNC*, 2013, pp. 150–157.

[12] R. Negulescu, "Process Spaces and Formal Verification of Asynchronous Circuits," Ph.D. dissertation, University of Waterloo, 1998.

[13] P. A. Cunningham, "Verification of Asynchronous Circuits," Univeristy of Cambridge, Tech. Rep. UCAM-CL-TR-587, 2004.

[14] H. K. Kapoor, "Delay-Insensitive Processes: A Formal Approach to the Design of Asynchronous Circuits," Ph.D. dissertation, London South Bank University, 2004.

[15] T. Bui, T. Nguyen, and A.-V. Dinh-Duc, "Experiences with representations and verification for asynchronous circuits," in *ICCE*, 2012, pp. 459–464.

[16] D. Borrione, M. Boubekeur, E. Dumitrescu, M. Renaudin, J.-B. Rigaud, and S. Sirianni, "An approach to the Introduction of Formal Validation in an Asynchronous Circuit Design Flow," in *HICSS*, 2003.

[17] R. K. Brayton and S. P. Khatri, "Multi-valued Logic Synthesis," in *12th International Conference On VLSI Design*, 1999, pp. 196–205.

[18] K. Gupta, *Discrete Mathematics*, 10th ed. Krishna Prakashan, 2009.

[19] D. K. Pradhan and I. G. Harris, *Practical Design Verification*. Cambridge University Press, 2009.

[20] A. Saifhashemi, "Power Optimization of Asynchronous Pipelines Using Conditioning and Re-Conditioning Based on A Three-Valued Logic Model," Ph.D. dissertation, University of Southern California, 2012.

[21] E. Seligman and I. Yarom, "Best Known Methods for Using Cadence Conformal LEC at Intel," in *Cadence User Conference*, 2006.