Chapter 15

Formal verification of NCL circuits

Ashiq Sakib[1], Son Le[1], Scott C. Smith[1] and
Sudarshan Srinivasan[1]

Validation is a critical component of any commercial design cycle. Testing-based approaches have been predominantly what has been used to ensure design correctness. AQ1 Formal verification is an alternate approach to design validation, where correctness is established using mathematical proofs. Since a proof can correspond to a very large number of test cases, formal verification has been found to be extremely useful in establishing design correctness and finding corner-case errors that often escape traditional testing. Since the now infamous FDIV bug (i.e., bug AQ2 found in the floating-point unit of the Intel Pentium processor in 1994 after shipping, which cost Intel $500 million to correct), the semiconductor industry has aggressively incorporated formal verification into its design cycle for validation.

One of the more popular formal verification approaches that have been found to be extremely scalable and useful in semiconductor design is equivalence checking. Typically, a lot of time, money, and effort are invested into ensuring the correctness of a design. However, the design itself is never static, as it is continuously tinkered with and optimized. Equivalence checking technology can, with a high degree of automation and efficiency, check that the golden model (i.e., the design that has been extensively validated) and its derivate are functionally equivalent. Scalability is harnessed by exploiting the structural similarity of the golden model and its derivate. Examples of commercial equivalence checkers include IBM Sixth Sense, Jasper Gold Sequential Equivalence Checker, Calypto SLEC, Mishchenko EBCCS13, and Cadence Encounter Conformal Equivalence Checker.

In this chapter, we describe an equivalence checking methodology for NCL AQ3 circuits. Note that currently, there are no commercial equivalence checkers for QDI circuits. For commercial applications, NCL circuits, and QDI circuits in general, are often synthesized from synchronous intellectual property designs. The resulting NCL design may then be further optimized and tinkered with. Therefore, we have designed an equivalence checker that can be used in two ways: (1) to verify the functional equivalence of two NCL designs and (2) to verify the equivalence between an NCL design and a synchronous design.

[1] Department of Electrical and Computer Engineering, North Dakota State University AQ4

15.1 Overview of approach

Vidura *et al.* [1] have previously developed an approach for verifying the equivalence of an NCL circuit against a synchronous circuit. They use the theory of Well-Founded Equivalence Bisimulation (WEB) refinement [2] as the notion of equivalence. In WEB refinement, both the circuit to be verified (here the NCL circuit) and the specification circuit (here the synchronous circuit) are modeled as transition systems (TSs), which capture the behavior of the circuit as a set of states and transitions between the states. WEB refinement essentially defines what it means for two TSs to be functionally equivalent. Their approach performs symbolic simulation on both the NCL circuit and the synchronous circuit to generate the TSs corresponding to both circuits. A decision procedure is then used to verify that the two TSs satisfy the WEB refinement property.

In working with the above approach, we found that because NCL circuits exhibit highly nondeterministic behaviors, the corresponding TSs are very complex, even for relatively simple circuits. This complexity leads to two issues. First is state space explosion. Second, it becomes very difficult to compute the reachable states of the resulting TS. Computing reachable states is important because unreachable states often flag numerous spurious counterexamples, which makes verification intractable.

We have therefore developed an alternate approach to circumvent having to deal with the NCL TS. The high-level idea is to perform structural transformation on the NCL circuit netlist to convert the NCL circuit into an equivalent synchronous circuit. The converted synchronous circuit is then compared against the specification synchronous circuit, using WEB refinement as the notion of correctness. The converted synchronous circuit, specification synchronous circuit, and the WEB refinement property are then automatically encoded in the Satisfiability Modulo Theory Library (SMT-LIB) language [3]. The resulting equivalence property is then checked using an SMT solver. Additional checks need to be performed to ensure that the NCL circuit is live (i.e., deadlock free). Thus, the overall verification has three high-level steps: (1) conversion from NCL to synchronous; (2) verification of converted synchronous against specification synchronous; and (3) additional checks on original NCL circuit to ensure liveness. The methodology can also be used to check the equivalence of two NCL circuits by applying the conversion technique to both NCL circuits to obtain two corresponding synchronous circuits, verifying these two synchronous circuits against each other, and performing the additional liveness checks on both NCL circuits.

15.2 Related verification works for asynchronous paradigms

Several formal verification techniques have been implemented to verify the two major asynchronous design paradigms: bounded-delay and QDI. The bounded-delay model is based on the assumption that the delay in all circuit components and wires is bounded—i.e., worse case delay can be calculated. Because of these timing constraints, most of the verification schemes for timed asynchronous models involve trace theory, Signal Transition Graph [4], and timed Petri nets. Reference [5] illustrates a gate-level verification method based on trace theory where the circuit, as well as the correctness properties, is modeled as Petri nets. An approach based on time-driven unfolding of Petri nets is used to verify freedom from hazards in asynchronous circuits consisting of logic gates and micropipelines [6]. However, timed-model-based verification methods are not applicable to QDI circuits, which are based on exactly the opposite assumption that circuit delays are unbounded and therefore indeterminate.

There exist several verification schemes specific to QDI circuits as well. Verbeek and Schmaltz [7] illustrate a deadlock-verification scheme for QDI circuits based on the Click Library [8]. Circuits based on this primitive library are structurally different from other QDI paradigms, such as NCL. Moreover, this method does not verify the functional correctness (safety) of the circuit. Refinement-based formal methods have been successful in verifying both bounded-delay and QDI asynchronous models. Desynchronized circuits, which are based on a bounded-delay structure, can be verified by a refinement-based approach, as discussed in [9]. As mentioned in the previous section, reference [1] presents a method to check the functional equivalence of NCL circuits against their synchronous counterparts using WEB refinement; and a model-checking-based method that checks for safety and liveness of PCHB circuits is presented in [10]. AQ5 However, both of these techniques suffer from state space explosion, since they model the QDI circuits as TSs, which become very complex for large circuits due to the nondeterministic behavior of QDI paradigms. Using a conversion technique along with WEB refinement, similar to that presented herein but applied to PCHB circuits, we were able to verify equivalence of combinational PCHB circuits with their Boolean specification, which proves to be highly scalable and much faster than previous techniques [11]. That method is currently being extended to sequential PCHB circuits.

Along with safety and liveness, input-completeness and observability are two critical properties of NCL circuits, which must be verified in order to ensure delay insensitivity, since a circuit may function correctly under normal operating conditions while not being input-complete or observable, but may then malfunction under extreme timing scenarios, such as those caused by process, voltage, or temperature variations. A manual approach to checking input-completeness is outlined in [12], which requires an analysis of each output term. For example, in order for output Z to be input-complete with respect to input A, every product term in all rails of Z (in SOP format) must contain any rail of A. This ensures that Z cannot be AQ6 DATA until A is DATA; and if Z is constructed solely out of NCL gates with hysteresis, the gate hysteresis ensures that Z cannot transition from DATA to NULL until A transitions from DATA to NULL. Hence, Z is input-complete with respect to A. However, this method cannot ensure input-completeness of relaxed NCL circuits [13], where not all gates contain hysteresis. Also, scalability is a problem with this approach, as the number of product terms that need to be verified grows exponentially as the number of inputs increase. Kondratyev *et al.* [14] provide a formal verification approach for observability verification, which entails determining all input combinations that assert $gate_i$, then forcing $gate_i$ to remain de-asserted while checking that none of those input combinations result in all circuit outputs becoming DATA. This check is performed for all gates to ensure circuit observability; and if also applied to each circuit input (i.e., replace $gate_i$ with $input_i$ in the observability check explanation), it will guarantee input-completeness. Our approach for observability checking, detailed in Section 15.3.5, is very similar to [14], while our approach checks input-completeness for all inputs simultaneously, as detailed in Section 15.3.4.

15.3 Equivalence verification for combinational NCL circuits

In industry, asynchronous NCL circuits are typically synthesized from their synchronous counterparts. Throughout the synthesis and optimization process, the synchronous specification undergoes several transformations, resulting in major structural differences between the implemented NCL circuit and its synchronous specification. For this kind of scenario, equivalence checking is a widely used formal verification method that checks for logical and functional equivalence between two different circuits.

NCL verification based on equivalence checking has proved to be a unified, fast, and scalable approach that eliminates most of the limiting factors of previous verification works in the field. The NCL equivalence verification method requires five steps, as described below and detailed in the following subsections: AQ7

Step 1: The netlist of an NCL circuit to be verified is converted into a corresponding Boolean/synchronous netlist, which is modeled in the SMT-LIB language using an automated script that we developed. The converted netlist is then checked against its corresponding Boolean/synchronous specification using an SMT solver to test for functional equivalence, as detailed in Section 15.3.1.

Step 2: Step 1 only checks the converted circuit's signals corresponding to the original NCL circuit's rail^1 signals, with their equivalent Boolean/synchronous specification external outputs or register outputs; hence, the original NCL circuit's rail^0 signals must also be ensured to be inverses of their respective rail^1 signals, through the invariant check detailed in Section 15.3.2, in order to guarantee safety after passing Step 1.

Step 3: The NCL netlist is then automatically converted into a graph structure, and information related to the handshaking control is gathered by traversing the graph. This information is utilized to analyze the handshaking correctness of the circuit in order to check for deadlock, as detailed in Section 15.3.3.

Steps 4 and 5: Once the NCL circuit passes Step 2, each combinational logic (C/L) block must be verified to be both input-complete (Step 4) and observable (Step 5) in order to guarantee liveness of the circuit under all timing scenarios, as detailed in Sections 15.3.4 and 15.3.5, respectively.

15.3.1 Functional equivalence check

A 3 × 3 NCL multiplier, shown in Figure 15.1(a), is used as an example to illustrate the equivalence verification procedure for combinational NCL circuits. NCL multipliers use input-incomplete NCL AND functions (denoted with an I inside the AND symbol), input-complete NCL AND functions (denoted with a C inside the AND symbol), NCL Half-Adders (HA), and NCL Full-Adders (FA), which all consist of a combination of NCL threshold gates, as shown in Figure 15.1(b), (c), (d), and (e), respectively. All signals in Figure 15.1(a) are dual-rail; and all registers are reset-to-NULL, denoted as REG_NULL. In addition to the I/O registers, the multiplier in Figure 15.1(a) includes one intermediate register stage to increase throughput.

The netlist of the NCL 3 × 3 multiplier is shown in Figure 15.2(a). The first two lines indicate all primary inputs and primary outputs, respectively. Lines 3–44 correspond to the NCL C/L threshold gates, where the first column is the type of gate, the second column lists the gate's inputs, in comma separated format starting with input A, and the last column is the gate's output. Lines 45–64 correspond to 1-bit NCL registers, where the first column is the reset type of the register (i.e., _NULL, _DATA0, or _DATA1, for reset to NULL, DATA0, or DATA1, respectively), the second column denotes the register's level (i.e., the depth of the path through registers without considering the C/L in-between. For the 3 × 3 multiplier example, there are three stages of registers, with levels 1, 2, and 3, starting from the input registers), the third and fourth columns are the register's rail^0 and rail^1 data inputs, respectively, the fifth and sixth columns are the register's *Ki* input and *Ko* output, respectively, and the seventh and eighth columns are the register's rail^0 and rail^1 data outputs, respectively. Lines 65–72 correspond to the C-elements (i.e., THnn gates) used in the handshaking control circuitry, where the first column is C*n*, with *n* indicating the number of inputs to the C-element, the second column lists the inputs in comma separated format, and the last column is the C-element's output. For example, C4 on line 65 is a four-input C-element.

The NCL netlist is input to a conversion algorithm that converts it into an equivalent Boolean netlist, as shown in Figure 15.2(b) for the Figure 15.2(a) example. Each NCL C/L gate is replaced with its corresponding Boolean gate that has the same set function, but no hysteresis; each internal dual-rail signal is already represented as two Boolean signals, the first for rail^1 and the second for rail^0, so no changes are needed for these; and each primary dual-rail input is replaced with that signal's rail^1, as this corresponds to the equivalent Boolean signal. The rail^1 primary inputs are then inverted to produce internal signals corresponding to what

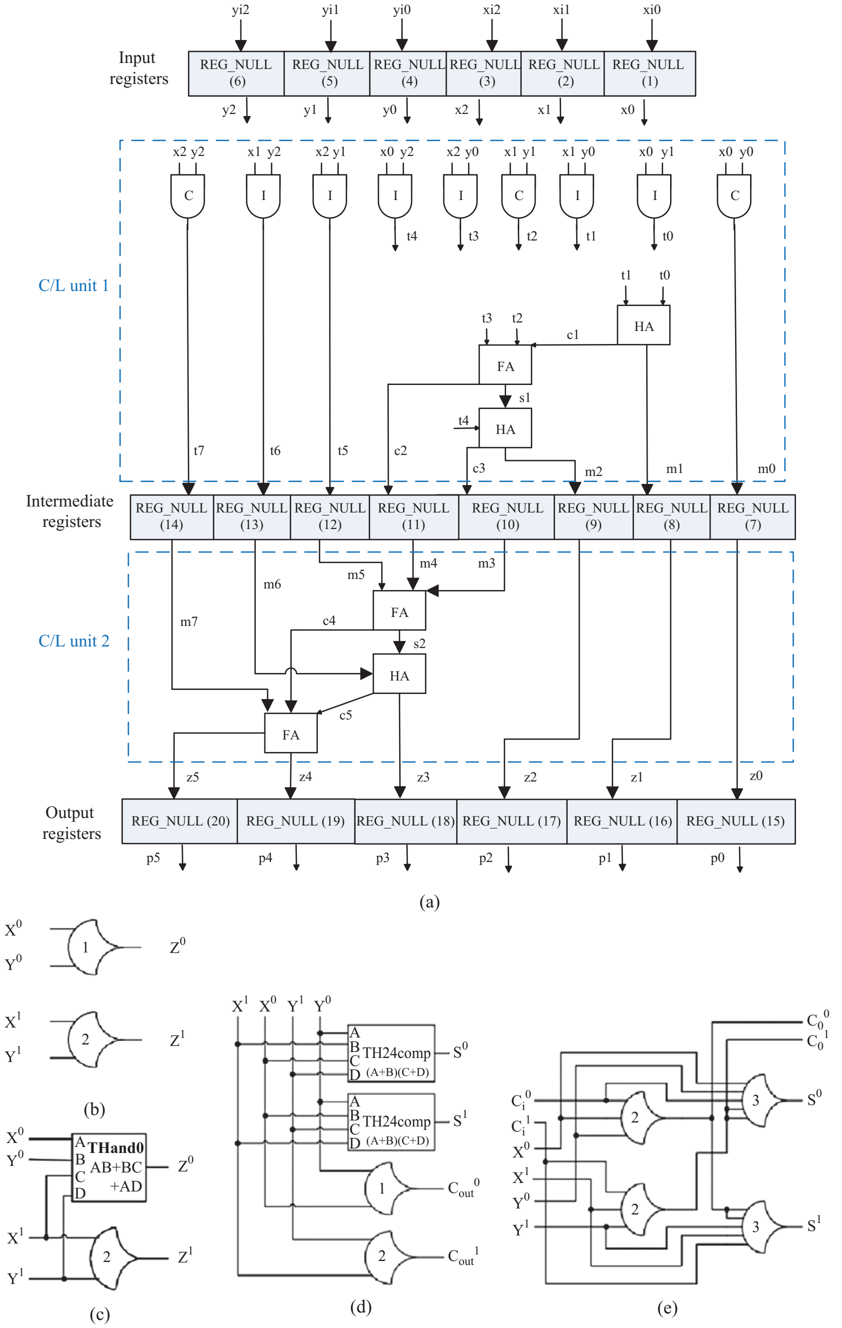

Figure 15.1 (a) 3 × 3 NCL multiplier circuit. (b) Input-incomplete NCL AND. (c) Input-complete NCL AND. (d) NCL HA. (e) NCL FA

AQ8

```
1.  xi0_0,xi0_1,xi1_0,xi1_1,…,yi1_0,yi1_1,yi2_0,yi2_1
2.  p0_0,p0_1,p1_0,p1_1,…,p5_0,p5_1
3.  th22 x0_1,y0_1  m0_1
4.  thand0 y0_0,x0_0,y0_1,x0_1  m0_0
5.  th22  x0_1,y1_1  t0_1
6.  th12  x0_0,y1_0  t0_0
7.  th22  x0_1,y2_1  t4_1
8.  th12  x0_0,y2_0  t4_0
9.  th22  x1_1,y0_1  t1_1
10. th12  x1_0,y0_0  t1_0
11. th22  x1_1,y1_1  t2_1
12. thand0  y1_0,x1_0,y1_1,x1_1  t2_0
13. th22  x1_1,y2_1  t6_1
14. th12  x1_0,y2_0  t6_0
15. th22  x2_1,y0_1  t3_1
16. th12  x2_0,y0_0  t3_0
17. th22  x2_1,y1_1  t5_1
18. th12  x2_0,y1_0  t5_0
19. th22  x2_1,y2_1  t7_1
20. thand0  y2_0,x2_0,y2_1,x2_1  t7_0
21. th24comp  t0_0,t1_0,t0_1,t1_1  m1_1
22. th24comp  t0_0,t1_1,t1_0,t0_1  m1_0
23. th22  t0_1,t1_1  c1_1
24. th12  t0_0,t1_0  c1_0
25. th23  t3_0,t2_0,c1_0  c2_0
26. th23  t3_1,t2_1,c1_1  c2_1
27. th34w2   c2_0,t3_1,t2_1,c1_1   s1_1
28. th34w2   c2_1,t3_0,t2_0,c1_0   s1_0
29. th24comp   s1_0,t4_0,s1_1,t4_1   m2_1
30. th24comp   s1_0,t4_1,t4_0,s1_1   m2_0
31. th22   s1_1,t4_1   c3_1
32. th12   s1_0,t4_0   c3_0
33. th23   m5_0,m4_0,m3_0   c4_0
34. th23   m5_1,m4_1,m3_1   c4_1
35. th34w2   c4_0,m5_1,m4_1,m3_1   s2_1
36. th34w2   c4_1,m5_0,m4_0,m3_0   s2_0
37. th24comp   s2_0,m6_0,s2_1,m6_1   z3_1
38. th24comp   s2_0,m6_1,m6_0,s2_1   z3_0
39. th22   s2_1,m6_1   c5_1
40. th12   s2_0,m6_0   c5_0
41. th23   m7_0,c4_0,c5_0   z5_0
42. th23   m7_1,c4_1,c5_1   z5_1
43. th34w2   z5_0,m7_1,c4_1,c5_1  z4_1
44. th34w2   z5_1,m7_0,c4_0,c5_0  z4_0
45. Reg_NULL   1   xi0_0 xi0_1   KO3   ko1    x0_0 x0_1
46. Reg_NULL   1   xi1_0 xi1_1   KO3   ko2    x1_0 x1_1
47. Reg_NULL   1   xi2_0 xi2_1   KO3   ko3    x2_0 x2_1
48. Reg_NULL   1   yi0_0 yi0_1   KO3   ko4    y0_0 y0_1
49. Reg_NULL   1   yi1_0 yi1_1   KO3   ko5    y1_0 y1_1
50. Reg_NULL   1   yi2_0 yi2_1   KO3   ko6    y2_0 y2_1
51. Reg_NULL   2   m0_0 m0_1   ko15   ko7    z0_0 z0_1
52. Reg_NULL   2   m1_0 m1_1   ko16   ko8    z1_0 z1_1
53. Reg_NULL   2   m2_0 m2_1   ko17   ko9    z2_0 z2_1
54. Reg_NULL   2   c3_0 c3_1   KO4   ko10   m3_0 m3_1
55. Reg_NULL   2   c2_0 c2_1   KO4   ko11   m4_0 m4_1
56. Reg_NULL   2   t5_0 t5_1   KO4   ko12   m5_0 m5_1
57. Reg_NULL   2   t6_0 t6_1   KO4   ko13   m6_0 m6_1
58. Reg_NULL   2   t7_0 t7_1   KO5   ko14   m7_0 m7_1
59. Reg_NULL   3   z0_0 z0_1   Ki   ko15   p0_0 p0_1
60. Reg_NULL   3   z1_0 z1_1   Ki   ko16   p1_0 p1_1
61. Reg_NULL   3   z2_0 z2_1   Ki   ko17   p2_0 p2_1
62. Reg_NULL   3   z3_0 z3_1   Ki   ko18   p3_0 p3_1
63. Reg_NULL   3   z4_0 z4_1   Ki   ko19   p4_0 p4_1
64. Reg_NULL   3   z5_0 z5_1   Ki   ko20   p5_0 p5_1
65. C4   ko7,ko8,ko9,ko10   KO1
66. C4   ko11,ko12,ko13,ko14   KO2
67. C2   KO1,KO2   KO3
68. C3   ko18,ko19,ko20   KO4
69. C2   ko19,ko20      KO5
70. C3   ko4,ko5,ko6   KO6
71. C3   ko1,ko2,ko3   KO7
72. C2   KO7,KO6   KO
```

(a)

```
1.  xi0_1,xi1_1,xi2_1,yi0_1,yi1_1,yi2_1
2.  p0_0,p0_1,p1_0,p1_1,…,p5_0,p5_1
3.  not    xi0_1   xi0_0
4.  not    xi1_1   xi1_0
5.  not    xi2_1   xi2_0
6.  not    yi0_1   yi0_0
7.  not    yi1_1   yi1_0
8.  not    yi2_1   yi2_0
9.  th22    xi0_1,yi0_1   p0_1
10. thand0    yi0_0,xi0_0,yi0_1,xi0_1   p0_0
11. th22    xi0_1,yi1_1   t0_1
12. th12    xi0_0,yi1_0   t0_0
13. th22    xi0_1,yi2_1   t4_1
14. th12    xi0_0,yi2_0   t4_0
15. th22  xi1_1,yi0_1 t1_1
16. th12  xi1_0,yi0_0 t1_0
17. th22  xi1_1,yi1_1 t2_1
18. thand0  yi1_0,xi1_0,yi1_1,xi1_1 t2_0
19. th22  xi1_1,yi2_1 t6_1
20. th12  xi1_0,yi2_0 t6_0
21. th22  xi2_1,yi0_1 t3_1
22. th12  xi2_0,yi0_0 t3_0
23. th22  xi2_1,yi1_1 t5_1
24. th12  xi2_0,yi1_0 t5_0
25. th22  xi2_1,yi2_1 t7_1
26. thand0    yi2_0,xi2_0,yi2_1,xi2_1   t7_0
27. th24comp    t0_0,t1_0,t0_1,t1_1   p1_1
28. th24comp    t0_0,t1_1,t1_0,t0_1   p1_0
29. th22    t0_1,t1_1   c1_1
30. th12    t0_0,t1_0   c1_0
31. th23    t3_0,t2_0,c1_0   c2_0
32. th23    t3_1,t2_1,c1_1   c2_1
33. th34w2    c2_0,t3_1,t2_1,c1_1   s1_1
34. th34w2    c2_1,t3_0,t2_0,c1_0   s1_0
35. th24comp    s1_0,t4_0,s1_1,t4_1   p2_1
36. th24comp    s1_0,t4_1,t4_0,s1_1   p2_0
37. th22    s1_1,t4_1   c3_1
38. th12    s1_0,t4_0   c3_0
39. th23    t5_0,c2_0,c3_0   c4_0
40. th23    t5_1,c2_1,c3_1   c4_1
41. th34w2    c4_0,t5_1,c2_1,c3_1   s2_1
42. th34w2    c4_1,t5_0,c2_0,c3_0   s2_0
43. th24comp    s2_0,t6_0,s2_1,t6_1   p3_1
44. th24comp    s2_0,t6_1,t6_0,s2_1   p3_0
45. th22    s2_1,t6_1   c5_1
46. th12    s2_0,t6_0   c5_0
47. th23    t7_0,c4_0,c5_0    p5_0
48. th23    t7_1,c4_1,c5_1   p5_1
49. th34w2     p5_0,t7_1,c4_1,c5_1   p4_1
50. th34w2     p5_1,t7_0,c4_0,c5_0   p4_0
```

(b)

Figure 15.2 (a) 3 × 3 NCL multiplier netlist. (b) Converted Boolean netlist used to be the rail^0 primary inputs, as these are utilized in the internal logic. The first two lines in the converted netlist are the list of primary inputs and outputs, respectively, where the inputs correspond to the original NCL netlist's rail^1 inputs, and the outputs include both rail^0 and rail^1 outputs. Lines 3–8 in the converted netlist are the added inverters used to produce the equivalent signals to the original rail^0 inputs, as these were removed in the conversion. The format of each gate is the same as explained above for the NCL netlist. All *Reg_NULL* components are removed during conversion by setting their data outputs equal to their data inputs, since these have no corresponding functionality in the equivalent Boolean circuit. Purely C/L circuits will not include *Reg_DATA* components, as these correspond to synchronous registers; these will be discussed in Section 15.4.

The converted Boolean netlist is automatically encoded in the SMT-LIB language [3], using a conversion tool we developed, which is then input to an SMT solver to check for functional equivalence with the corresponding specification. For the 3 × 3 multiplier example, the SMT solver checks for the following safety property: $\mathrm{F}_{\mathrm{NCL\_Bool\_Equiv.}}(x2\_1, x1\_1, x0\_1, y2\_1, y1\_1, y0\_1) = \mathrm{MUL}\ (x, y)$, where $(x2\_1, x1\_1, x0\_1)$ and $(y2\_1, y1\_1, y0\_1)$ are the x and y rail^1 inputs, respectively, starting with the MSB. We use the Z3 SMT solver [15] to check for equivalence, but any combinational equivalence checker could be used. Note that only the rail^1 outputs need to be checked here, as these correspond to the Boolean specification circuit outputs. The rail^0 outputs will be utilized for the invariant check, described next. AQ9

15.3.2 Invariant check

Since only the rail^1 outputs are utilized for the functional equivalence check, the rail^0 outputs must also be checked to ensure safety. To address correctness of the rail^0 outputs, an additional SMT invariant proof obligation is required for the original NCL circuit, which states that in any reachable NCL circuit state where the outputs are all DATA, every rail^0 output must be the inverse of its corresponding rail^1 output.

One way to achieve this is to initialize all registers to NULL, all C/L gate outputs to 0, and all register *Ki* inputs to *rfd* (i.e., logic 1), then make all the primary inputs DATA (i.e., represented in SMT as all combinations of valid DATA) and step the circuit. This will allow the input DATA to flow through all stages of the circuit, generating all possible combinations of valid DATA at the primary outputs. For each primary dual-rail output, the invariant is then checked to ensure that the rail^0 output is the inverse of its corresponding rail^1 output. For a C/L circuit with j registers $r^1, \ldots, r^j$, k C/L threshold gates $g^1, \ldots, g^k$, q dual-rail inputs $i^1, \ldots, i^q$, and l dual-rail outputs $o^1{<}R^0, R^1{>}, \ldots, o^l{<}R^0, R^1{>}$, where R^0 and R^1 are the output's rail^0 and rail^1, respectively, the proof obligation for this invariant check is shown below as Proof Obligation 1. Predicate *P1* indicates that all registers in are reset-to-NULL. *P2* and *P3* state that all threshold gates and *Ki* register inputs are initialized to logic 0 and 1, respectively. *P4* indicates that all inputs are DATA. *P5* represents the symbolic step of the circuit with all threshold gates set to 0 and all inputs set to DATA, with the new values of the threshold gates stored in $(g_B{}^1, \ldots, g_B{}^k)$. *P6* states that the rails of each dual-rail output are complements of each other. The proof obligation, *PO1*, indicates that if DATA is allowed to flow from the primary inputs to the primary outputs, then for all possible valid DATA inputs, each output's rail^0, R^0, is always the inverse of its respective rail^1 output, R^1.

Proof Obligation 1:
P1: $\wedge_{n=1}^{j}(r_A{}^n = 0b00)$
P2: $\wedge_{n=1}^{k}(g_A{}^n = 0)$
P3: $\wedge_{n=1}^{j}(Ki_A{}^n = 1)$
P4: $\wedge_{n=1}^{q}(i_A{}^n = 0b01) \vee (i_A{}^n = 0b10)$
P5: $\left(g_B{}^1, \ldots, g_B{}^k\right) = NCLStep(i_A{}^1, \ldots, i_A{}^q)$
P6: $\wedge_{n=1}^{n} o_B^n < R^0 > = \neg o_B{}^n < R^1 >$
PO1: $\{P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5 \Rightarrow P6\}$

An alternative, faster method to check invariants is to check each NCL circuit stage independently. To do this, we developed an algorithm that reads the original NCL circuit netlist and separately extracts each circuit stage. Then, for each extracted stage, we set all gate outputs to 0, all stage inputs to DATA, and step the circuit, such that the stage's outputs become all possible combinations of valid DATA. Finally, the invariant is checked for each of the stage's dual-rail outputs to ensure that its rail^0 is the inverse of its corresponding rail^1. The proof obligation for this second invariant check method is shown below as Proof Obligation 2, where the extracted stage has *j* dual-rail inputs $i^1, \ldots, i^j$, *m* threshold gates $g^1, \ldots, g^m$, and *k* dual-rail outputs $o^1{<}R^0, R^1{>}, \ \ldots, o^k{<}R^0, R^1{>}$, where R^0 and R^1 are the output's rail^0 and rail^1, respectively. Predicate *P1* indicates that all stage inputs are valid DATA; *P2* indicates that all NCL threshold gates in the stage are initialized to 0; *P3* corresponds to a NULL to DATA transition of the stage; and *P4* states that the rails of each dual-rail output are complements of each other. The Proof Obligation, *PO2*, states that after a NULL to DATA transition of the stage with all possible valid DATA inputs, that each output's rail^0, R^0, is always the inverse of its respective rail^1 output, R^1.

Proof Obligation 2:
P1: $\wedge_{n=1}^{j}(i_A{}^n = 0b01) \vee (i_A{}^n = 0b10)$
P2: $\wedge_{n=1}^{m}(g_A{}^n = 0)$
P3: $(g_B{}^1, \ldots, g_B{}^m) = NCLStep(i_A{}^1, \ldots, i_A{}^j)$
P4: $\wedge_{n=1}^{k} o_B{}^n < R^0 > = \neg o_B{}^n < R^1 >$
PO2: $\{P1 \wedge P2 \wedge P3 \Rightarrow P4\}$

This second invariant check method is much faster than the first, since it breaks the problem into a set of smaller invariant checks (i.e., one per stage), whereas the first method checks the invariant for the entire circuit all at once. For example, Method 2 is 38% faster for a two-stage 10 × 10 multiplier, and becomes even faster when the circuit includes additional stages. Note that for both invariant check methods, the NCL gates are modeled in SMT as Boolean functions (i.e., no hysteresis), since invariant checking only requires the NULL to DATA transition, which only utilizes each gate's set function, that is, the same for the Boolean and NCL state-holding gate implementations. This optimization reduces the invariant check time by approximately half (e.g., 377 vs. 192 s for a non-pipelined 10-bit × 10-bit unsigned multiplier).

15.3.3 Handshaking check

Liveness means absence of deadlock in a circuit. For combinational NCL circuits, proper connections between handshaking signals, along with observable and input-complete C/L, ensures liveness. The same NCL netlist shown in Figure 15.2(a), used as input for the functional equivalence and invariant checks, is also utilized as input for the liveness checks. For the handshaking check, the NCL netlist is automatically converted into a graph structure, and the handshaking paths and C-element connections are traced back to verify proper handshaking, ensuring that every register acknowledges all preceding stage register outputs that took part in calculating its input. For each NCL register, *i*, its dual-rail input is traced back through its preceding C/L to identify every NCL register's dual-rail output that took part in its calculation, generating a fan-in list, *reg_fanin(i)*. For example, referring to Figure 15.1(a), *reg_fanin(8)* would be *1*, *2*, *4*, *5*, since *x0*, *x1*, *y0*, and *y1* are all used to generate *m1*. Also, for each NCL register, *i*, its *Ko* output is traced through the C-element handshaking circuitry to identify every NCL register's *Ki* input that register$_i$'s *Ko* output took part in calculating, generating a *Ko* fanout list, *ko_fanout(i)*. For example, referring to Figure 15.3, which shows the handshaking circuitry for the 3 × 3 multiplier example, *ko_fanout(8)* would be *1*, *2*, *3*, *4*, *5*, *6*, since *ko8* takes part in the generation of the *Ki* input for all of the preceding stage's registers (i.e., 1–6).

After *reg_fanin* and *ko_fanout* for each NCL register is calculated, as shown in Figure 15.4 for the 3 × 3 multiplier example, *reg_fanin(k)* is checked to ensure that it is a subset of *ko_fanout(k)*, for all NCL registers. Note that 0 in *reg_fanin* denotes a primary data input; and 0 in *ko_fanout* denotes the external *Ko* output. Bit-wise completion results in *reg_fanin(k)* being equal to *ko_fanout(k)*, while full-word completion results in *reg_fanin(k)* being a proper subset of *ko_fanout(k)*, with the restriction that each register that is in *ko_fanout(k)* and not in *reg_fanin(k)* must be from the immediate preceding register stage of register *k*. *reg_fanin(k)* not being a subset of *ko_fanout(k)* could result in deadlock, while *reg_fanin(k)* being a proper subset of *ko_fanout(k)* but violating the stage restriction described above, could either result in deadlock or may just decrease circuit performance. Hence, if *reg_fanin(k)* is a proper subset of *ko_fanout(k)*, then each register that is in *ko_fanout(k)* and not in *reg_fanin(k)* is automatically inspected to ensure that it meets this stage restriction. If not, a warning message is generated denoting the

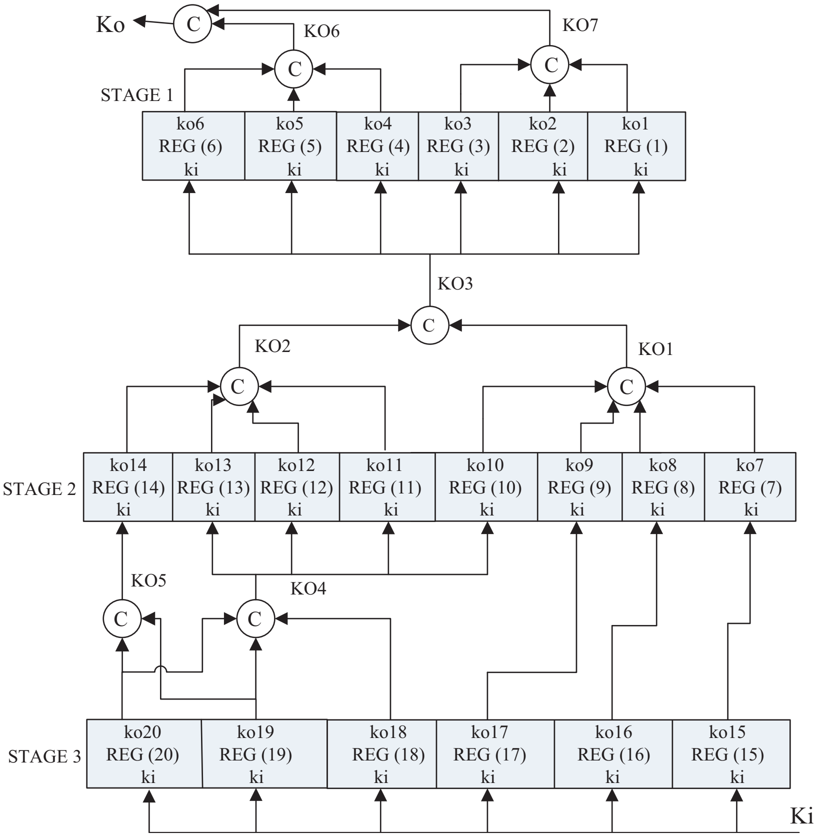

Figure 15.3 Handshaking connections for the 3 × 3 NCL multiplier

extra register in that particular register's *ko_fanout* list, to allow for easier manual inspection. For the Figure 15.3 example, the first stage utilizes full-word completion, while the second stage uses bit-wise completion.

An additional check is needed to ensure correct connection of the external *Ki* input, namely that the external *Ki* should be the *Ki* input to every register that produces a primary data output. The developed algorithm generates an appropriate descriptive error message in case the NCL circuit fails to satisfy any of these handshaking checks. Furthermore, it checks to ensure that no data signal is part of the handshaking circuitry, and that no handshaking signal is part of a data signal.

The methodology has been demonstrated on several multipliers and ISCAS-85 [16] combinational circuit benchmarks, as shown in Table 15.1. *umultN* represents a non-pipelined *N*-bit × *N*-bit unsigned multiplier. The NCL-to-Boolean netlist conversion time was negligible compared to the safety and invariant checks

```
 1:  reg_fanin: 0                       ko_fanout: 0
 2:  reg_fanin: 0                       ko_fanout: 0
 3:  reg_fanin: 0                       ko_fanout: 0
 4:  reg_fanin: 0                       ko_fanout: 0
 5:  reg_fanin: 0                       ko_fanout: 0
 6:  reg_fanin: 0                       ko_fanout: 0
 7:  reg_fanin: [1, 4]                  ko_fanout: [1, 2, 3, 4, 5, 6]
 8:  reg_fanin: [1, 2, 4, 5]            ko_fanout: [1, 2, 3, 4, 5, 6]
 9:  reg_fanin: [1, 2, 3, 4, 5, 6]      ko_fanout: [1, 2, 3, 4, 5, 6]
10: reg_fanin: [1, 2, 3, 4, 5, 6]       ko_fanout: [1, 2, 3, 4, 5, 6]
11: reg_fanin: [1, 2, 3, 4, 5]          ko_fanout: [1, 2, 3, 4, 5, 6]
12: reg_fanin: [3, 5]                   ko_fanout: [1, 2, 3, 4, 5, 6]
13: reg_fanin: [2, 6]                   ko_fanout: [1, 2, 3, 4, 5, 6]
14: reg_fanin: [3, 6]                   ko_fanout: [1, 2, 3, 4, 5, 6]
15: reg_fanin: [7]                      ko_fanout: [7]
16: reg_fanin: [8]                      ko_fanout: [8]
17: reg_fanin: [9]                      ko_fanout: [9]
18: reg_fanin: [10, 11, 12, 13]         ko_fanout: [10, 11, 12, 13]
19: reg_fanin: [10, 11, 12, 13, 14]     ko_fanout: [10, 11, 12, 13, 14]
20: reg_fanin: [10, 11, 12, 13, 14]     ko_fanout: [10, 11, 12, 13, 14]
```

Figure 15.4 reg_fanin and ko_fanout lists for the 3 × 3 NCL multiplier

Table 15.1 Verification results of various C/L NCL circuits

Circuits	**Functional check (s)**	**Invariant check (s)**	**Handshaking check (s)**	**Total time (s)**
ISCAS c17	0.01	0.01	0.0020	0.0220
umult2	0.02	0.01	0.0997	0.1297
umult3	0.04	0.02	0.1087	0.1687
umult6	0.32	0.33	0.8238	1.4738
umult8	10.62	6.79	9.3090	26.719
umult10	683.49	192.39	70.370	946.25
ISCAS c432	1.03	1.06	3.0111	5.1011
umult10-*B1*	0.08 (B)	0.10 (B)	70.370	70.550
umult10-*B2*	0.06 (B)	192.39	70.370	262.82
umult10-*B3*	683.49	192.39	69.1538 (B)	945.034
umult10-*B4*	683.49	0.08 (B)	72.0235 (B)	755.5935
umult10-*B5*	0.1 (B)	0.09 (B)	70.37	71.37

performed by the Z3 SMT solver [15] on an Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz. To test the methodology, we injected several bugs. The *umult*10-Bn multipliers are circuits with *n* different kinds of bugs, and the (B) in either the Functional Check, Invariant check, or Handshaking Check column denotes which check detected the bug. The –B1 bug incorrectly swaps rails of a dual-rail signal. –B2 represents a faulty data connection. For example, the *F* output of NCL $gate_i$ should be connected to the *X* input of NCL $gate_j$; however, *X* is instead connected to the output of NCL $gate_k$, which results in a logical error. –B3 corresponds to an incorrect handshaking connection; and external *Ki* and *Ko* bugs are represented by –B4. –B5 denotes a rail-duplication error, where rail^0 and rail^1 of a particular signal are the same wire. Z3 reported all functional and invariant bugs along with a counter example; and our handshaking check tool identified and reported the location of all inserted completion logic bugs.

15.3.4 Input-completeness check

Input-completeness requires that all outputs of a combinational circuit may not transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and that all outputs of a combinational circuit may not transition from DATA to NULL until all inputs have transitioned from DATA to NULL [12]. In circuits with multiple outputs, it is acceptable according to Seitz's "weak conditions" of delay-insensitive signaling, for some of the outputs to transition without having a complete input set present, as long as all outputs cannot transition before all inputs arrive [17]. Input-completeness of every C/L stage is required for NCL circuits to be QDI; an input-incomplete stage may cause the circuit to deadlock under some timing scenarios.

There are two proof obligations required for verification of input-completeness. These two proof obligations have been developed to accommodate two scenarios, the first for when the circuit transitions from NULL to DATA, and the second for the transition from DATA to NULL. Both proof obligations have been generalized so that they apply to all NCL combinational circuits. The proof obligations have been encoded in a decidable fragment of first-order logic, and are automatically checked using an SMT solver.

15.3.4.1 Input-completeness proof obligation: NULL to DATA

Assume an NCL circuit has m threshold gates, p dual-rail-inputs, and q dual-rail outputs. Let ${g_A}^1, \ldots, {g_A}^m$ represent Boolean variables that correspond to the current state of the NCL threshold gates before *step A*, and ${g_B}^1, \ldots, {g_B}^m$ represent the same threshold gates' state after *step A*. Let ${i_A}^1, \ldots, {i_A}^p$ represent the circuit inputs for *step A*, and ${i_B}^1, \ldots, {i_B}^p$ for *step B*. Let ${o_A}^1, \ldots, {o_A}^q$ be the circuit output values after symbolically stepping the circuit using inputs ${i_A}^1, \ldots, {i_A}^p$ and threshold gate states ${g_A}^1, \ldots, {g_A}^m$. Let ${o_B}^1, \ldots, {o_B}^q$ be the circuit output values after symbolically stepping the circuit using inputs ${i_B}^1, \ldots, {i_B}^p$ and threshold gate states ${g_B}^1, \ldots, {g_B}^m$. The predicates used in the proof obligations for input-completeness are given in Table 15.2.

p_0 indicates that no dual-rail inputs are in an illegal state. p_1 states that all the threshold gate's current output values are 0, which indicates that the circuit is in the NULL state before a DATA transition. p_2 indicates that at least one of the dual rail inputs is NULL, and p_3 indicates that at least one of the dual-rail outputs is NULL. Proof Obligation *PO3*, below, is used to check input-completeness of the NULL to DATA transition of the circuit. *PO3* essentially states that if none of the inputs are ILLEGAL, all current threshold gate outputs are 0, and at least one of the dual-rail

Table 15.2 Proof obligation predicates for input-completeness

p_n	Predicate
p_0	$\bigwedge_{n=1}^{n=p} \sim ({i_A}^n = 0b11)$
p_1	$\bigwedge_{n=1}^{n=m} ({g_A}^n = 0)$
p_2	$\bigvee_{n=1}^{n=p} ({i_A}^n = 0b00)$
p_3	$\bigvee_{n=1}^{n=q} ({o_A}^n = 0b00)$
p_4	$\bigwedge_{n=1}^{n=p} (({i_A}^n = 0b01) \vee ({i_A}^n = 0b10))$
p_5	$({g_B}^1, \ldots, {g_B}^m) = NCLStep({i_A}^1, \ldots, {i_A}^p)$
p_6	$\bigwedge_{n=1}^{n=p} (({i_B}^n = {i_A}^n) \vee ({i_B}^n = 0b00))$
p_7	$\bigvee_{n=1}^{n=p} ({i_B}^n = {i_A}^n)$
p_8	$\bigvee_{n=1}^{n=q} (({o_B}^n = 0b01) \vee ({o_B}^n = 0b10))$

inputs is NULL, then at least one of the dual-rail outputs must be NULL. Since the dual-rail inputs in the proof obligation are symbolic, the SMT solver checks this property for all possible input combinations.

$$PO3\colon \{p_0 \wedge p_1 \wedge p_2\} \rightarrow p_3$$

15.3.4.2 Input-completeness proof obligation: DATA to NULL

When NCL circuits are constructed using only threshold gates with hysteresis, ensuring input-completeness of the NULL to DATA transition guarantees input-completeness of the DATA to NULL transition, since gate hysteresis ensures that a gate output cannot transition to 0 until all its inputs transition to 0. However, this is not the case for relaxed NCL circuits [13], which are comprised of both threshold gates with hysteresis and Boolean gates. Hence, for relaxed NCL circuits, input-completeness of the DATA to NULL transition must also be checked.

To formulate the DATA to NULL proof obligation, the circuit must first be symbolically initialized with all possible threshold gate outputs after a transition from NULL to DATA. This is done by first initializing the circuit to the NULL state (i.e., all threshold gates are set to 0) and then stepping the circuit with valid symbolic DATA (i.e., not NULL and not illegal) inputs, identified as *step A*.

The symbolic values of the threshold gates from *step A* are retained, and the circuit is symbolically stepped again with new inputs, identified as *step B*, which represents the DATA to NULL transition.

p_1 initializes all threshold gate outputs to 0 before *step A*. p_4 indicates that all *step A* inputs are DATA. p_5 represents the symbolic step of the circuit with all threshold gates set to 0 and all inputs set to DATA, with the new values of the threshold gates stored in $(g_B{}^1, \ldots, g_B{}^m)$. p_6 indicates that each input for *step B* is either the same DATA value it was for *step A*, or has transitioned to NULL. p_7 indicates that at least one of the inputs for *step B* is still DATA; and p_8 indicates that at least one of the outputs of *step B* remains DATA. The final proof obligation for input-completeness of the DATA to NULL transition is given below as *PO4*. It states that after initializing the circuit to the NULL state and symbolically stepping the circuit with all possible DATA inputs to generate all possible DATA states, that if at least one dual-rail input remains DATA while other inputs may transition to NULL, then at least one of the outputs must remain DATA, meaning that the circuit has not fully transitioned to the NULL state, because all inputs have not yet transitioned to NULL. Like the NULL to DATA proof obligation, all inputs are symbolic, so the SMT solver checks all combinations.

$$PO4\text{: } \{p_1 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7\} \rightarrow p_8$$

15.3.4.3 Input-completeness results

Verification of the proof obligations for input-completeness can be performed using any SMT solver. To perform input-completeness verification, we developed a tool to automatically generate the circuit model and proof obligation specifications, encoded in SMT-LIB format, from the original circuit netlist, such as the one shown in Figure 15.2(a) for the 3 × 3 multiplier. For the verification results presented here, *N*-bit × *N*-bit unsigned dual-rail NCL multipliers were used as benchmarks, where $3 \leq N \leq 15$. The ISCAS-85 C432 27-channel interrupt controller circuit was also used as a benchmark [18]. The verification proof obligations were checked using the Z3 SMT solver on an Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz.

The verification results are listed in Table 15.3, where the first column is the Circuit Name, the second column is the verification time for the NULL to DATA proof obligation of a correct input-complete implementation, the third column is the verification time for the NULL to DATA proof obligation of an incorrect input-incomplete implementation, and columns four and five report the verification times for the DATA to NULL proof obligations for input-complete and input-incomplete implementations, respectively. *umultN* represents an *N*-bit × *N*-bit unsigned multiplier constructed using only NCL gates with hysteresis, while $r - umultN$ represents a relaxed version of the *N*-bit × *N*-bit multiplier, where NCL gates are replaced with Boolean gates when hysteresis is not required for input-completeness. Timeout (TO) is listed in the verification results when the verification time exceeded 1 day.

Table 15.3 Input-completeness verification times (s)

Circuit	N to D	Buggy N to D	D to N	Buggy D to N
*umult*3	0.02	0.01	0.03	0.04
*umult*4	0.02	0.05	0.06	0.06
*umult*5	0.09	0.05	0.12	0.11
*umult*6	0.11	0.15	0.38	0.24
*umult*7	0.38	0.27	1.49	1.23
*umult*8	1.44	0.49	5.47	3.60
*umult*9	5.30	2.37	22.38	1.28
*umult*10	20.22	8.92	102.42	18.45
*umult*11	54.09	2.99	430.29	22.81
*umult*12	236.00	8.21	1,909.44	23.17
*umult*13	885.30	3.85	7,401.11	15.11
*umult*14	3,424.89	114.41	34,961.6	8.26
*umult*15	9,663.01	19.41	***TO***	112.55
$r - umult3$	0.02	0.02	0.04	0.07
$r - umult4$	0.02	0.02	0.06	0.07
$r - umult5$	0.05	0.04	0.10	0.08
$r - umult6$	0.15	0.12	0.42	0.07
$r - umult7$	0.39	0.12	1.48	0.11
$r - umult8$	1.38	1.43	6.38	0.17
$r - umult9$	4.74	5.17	28.03	0.20
$r - umult10$	16.26	19.02	146.95	0.20
$r - umult11$	58.04	46.53	642.80	0.31
$r - umult12$	215.75	228.47	3,635.01	0.35
$r - umult13$	729.11	34.97	15,663.24	0.40
$r - umult14$	3,045.99	4,104.45	80,213.90	0.68
$r - umult15$	10,561.11	9,974.39	***TO***	0.308
*C*432	0.062	0.068	0.074	0.94

The benchmark multipliers were designed as shown for the 3 × 3 version in Figure 15.1, with input-complete AND functions to generate the X_iY_i partial products and input-incomplete AND functions for the X_iY_j partial products, where $i \neq j$, but without the intermediate NCL register (i.e., a single stage with only input and output registers [19]). To create the buggy non-relaxed versions, $1 \leq k \leq N$ was chosen at random and the input-complete AND function used to generate the X_kY_k partial product was replaced with an input-incomplete version. NCL HAs and FAs are inherently input-complete and therefore cannot be made input-incomplete when constructed only using NCL gates with hysteresis. The relaxed version of each multiplier was constructed by taking the non-relaxed version and replacing the TH22 gate within the input-incomplete AND functions and HAs with a Boolean AND gate. Buggy relaxed circuits were constructed by relaxing one of the following: either the TH22 or THand0 gate in a X_iY_i partial product AND function, a TH24comp gate in a HA, or either a TH34w2 or TH23 gate in a FA. The ISCAS-85 C432 circuit was designed using input-incomplete functions when possible while maintaining input-completeness. The buggy version was obtained by replacing one of the input-complete 3-input NAND functions that calculate RC, in Module M3 [20], with an input-incomplete version. Z3 reported all bugs along with a counter example.

15.3.5 Observability check

Observability requires every gate transition to be observable at the output, which means that every gate that transitions is necessary to transition at least one output. Observability of every gate in every C/L stage is required for NCL circuits to be QDI; an unobservable gate in any stage may cause the circuit to deadlock under some timing scenarios. Observability can be proven in a similar fashion to input-completeness. Two proof obligations are needed for each C/L gate, one for the NULL to DATA transition, and the other for the DATA to NULL transition. The proof obligations, like those for input-completeness, have been encoded in a decidable fragment of first order logic and are automatically checked using an SMT solver.

15.3.5.1 Observability proof obligation: NULL to DATA

To verify observability, a check must be performed on each C/L gate. For each gate g^1, …, g^m, assertion of that gate is first computed, denoted as $f_1, \ldots, f_m = 1$, respectively. During the NULL to DATA observability verification of g^n, where $1 \leq n \leq m$, the output of g^n is forced to 0. Simulation of a circuit with g^n forced to 0 is called a Gn0 simulation, and the resulting function is $nclcktGn0(i^1, \ldots, i^p)$. To formulate the DATA to NULL observability proof obligation, the circuit must first be symbolically initialized with all possible threshold gate outputs that assert g^n after a transition from NULL to DATA. This is done by first initializing the circuit to the NULL state (i.e., all threshold gates are set to 0) and then stepping the circuit with valid symbolic DATA (i.e., not NULL and not illegal) inputs, identified as *step A*. The symbolic values of the threshold gates from *step A* are retained as ${g_B}^1, \ldots, {g_B}^m$, and the circuit is symbolically stepped again with new inputs, identified as *step B*, which represents the DATA to NULL transition. During the verification of g^n, where $1 \leq n \leq m$, the output of g^n is forced to 1. Simulation of a circuit with g^n forced to 1 is called a Gn1 simulation, and the resulting function is $nclcktGn1(i^1, \ldots, i^p)$. Additional predicates used in the proof obligations for observability are given in Table 15.4.

p_1 states that all the threshold gates' current output value is 0, which indicates that the circuit is in the NULL state before a DATA transition. p_4 indicates that every circuit input is valid DATA. p_9 assigns the outputs of the NCL circuit for a Gn0 simulation, where the output of g^n, the gate under test, is forced to 0. p_{10} enables only valid input combinations that would assert g^n to be used to step the circuit in p_9. Finally, p_{11} ensures that at least one of the outputs is NULL. The proof obligation to test observability of the NULL to DATA transition is given below as *PO5*, which tests observability of all gates, g^1, …, g^m. If true for g^n, this ensures that there is at least one output that will not be asserted if g^n is not asserted, for all

Table 15.4 Additional proof obligation predicates for observability

p_n	**Predicate**
p_9	$({o_A}^1, \ldots, {o_A}^q) = nclcktGn0({i_A}^1, \ldots, {i_A}^p)$
p_{10}	$f_n = 1$
p_{11}	$\bigvee_{n=1}^{n=q} ({o_B}^n = 0b00)$
p_{12}	$\bigwedge_{n=1}^{n=p} ({i_B}^n = 0b00)$
p_{13}	$({o_B}^1, \ldots, {o_B}^q) = nclcktGn1({i_B}^1, \ldots, {i_B}^p)$
p_{14}	$\sim \bigwedge_{n=1}^{n=q} ({o_B}^n = 0b00)$

sets of inputs in which g^n should be asserted, therefore proving that g^n is observable for the NULL to DATA transition.

$$PO5: \bigwedge_{n=1}^{n=m} (\{p_1 \wedge p_4 \wedge p_9 \wedge p_{10}\} \rightarrow p_{11})$$

15.3.5.2 Observability proof obligation: DATA to NULL

Like input-completeness, NCL circuits consisting only of NCL gates with hysteresis are inherently observable for the DATA to NULL transition if observable for the NULL to DATA transition, since gate hysteresis ensures that a gate output cannot transition to 0 until all its preceding gates' outputs transition to 0. However, this is not the case for relaxed NCL circuits, which are comprised of both threshold gates with hysteresis and Boolean gates. Hence, for relaxed NCL circuits, observability of the DATA to NULL transition must also be checked.

p_1 initializes all threshold gate outputs to 0 before *step A*. p_4 indicates that all *step A* inputs are DATA. p_5 represents the symbolic step of the circuit with all threshold gates set to 0 and all inputs set to DATA, with the new values of the threshold gates stored in $({g_B}^1, \ldots, {g_B}^m)$. p_{10} enables only valid input combinations that would assert g^n to be used to step the circuit in p_5. p_{12} indicates that all inputs for *step B* have transitioned to NULL. p_{13} assigns the outputs of the NCL circuit for a Gn1 simulation, where the output of g^n, the gate under test, is forced to 1. Finally, p_{14} ensures that all outputs are not NULL. The proof obligation to test observability of the DATA to NULL transition is given below as *PO6*, which tests observability of all gates, $g^1, \ldots, g^m$. If true for g^n, this ensures that following a NULL to DATA transition that asserts g^n, there is at least one output that will not become NULL during the subsequent DATA to NULL transition while g^n remains asserted, therefore proving that g^n is observable for the DATA to NULL transition.

$$PO6: \bigwedge_{n=1}^{n=m} (\{p_1 \wedge p_4 \wedge p_5 \wedge p_{10} \wedge p_{12} \wedge p_{13}\} \rightarrow p_{14})$$

Table 15.5 Observability verification times (s)

Circuit	**N to D**	**D to N**
*umult*4	0.001	0.001
*umult*5	8.203	8.944
*umult*6	13.7599	16.1921
*umult*7	27.8229	36.528
*umult*8	54.062	105.4979
*umult*9	138.3139	412.605
*umult*10	363.7079	1,968.434
*umult*11	902.046	9,657.475
*umult*12	2,384.504	52,093.64
*umult*13	5,797.037	***TO***
*C*432*M*1	1.53	3.882

15.3.5.3 Observability results

Verification of the proof obligations for observability can be performed using any SMT solver. To perform observability verification, we developed a tool to automatically generate the circuit model and proof obligation specifications, encoded in SMT-LIB format, from the original circuit netlist, such as the one shown in Figure 15.2(a) for the 3×3 multiplier. For the verification results presented here, N-bit $\times$ N-bit unsigned dual-rail NCL multipliers were used as benchmarks, where $3 \leq N \leq 13$. The ISCAS-85 C432 27-channel interrupt controller circuit was also used as a benchmark [18]. The verification proof obligations were checked using the Z3 SMT solver on an Intel® Core™ i7-4790 CPU with 32GB of RAM, running at 3.60 GHz.

The verification results are listed in Table 15.5, where the first column is the Circuit name, the second column is the verification time for the NULL to DATA proof obligation, and the third column is the verification time for the DATA to NULL proof obligation. *umultN* represents an N-bit $\times$ N-bit unsigned multiplier constructed using only NCL gates with hysteresis, while $r - umultN$ represents a relaxed version of the N-bit $\times$ N-bit multiplier, where NCL gates are replaced with Boolean gates when hysteresis is not required. TO is listed in the verification results when the verification time exceeded 1 day.

The test multipliers were designed exactly the same as the ones used for testing input-completeness (i.e., input-complete AND functions generate the X_iY_i partial products, and input-incomplete AND functions generate the X_iY_j partial products, where $i \neq j$). To create buggy multipliers that were input-complete but not observable, an HA was chosen at random and the XOR function to generate its sum (i.e., the two TH24comp gates in Figure 15.1(d)) was replaced with the unobservable XOR function, shown in Figure 15.5. To check observability of relaxed circuits, the M1 module of the ISCAS-85 C432 benchmark [21] was used, where the nine-input NAND function that generates *PA* was composed AQ10
of two relaxed input-incomplete four-input AND functions, followed by an

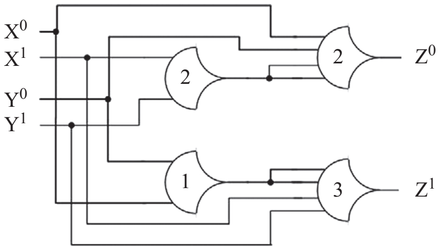

Figure 15.5 Unobservable NCL XOR

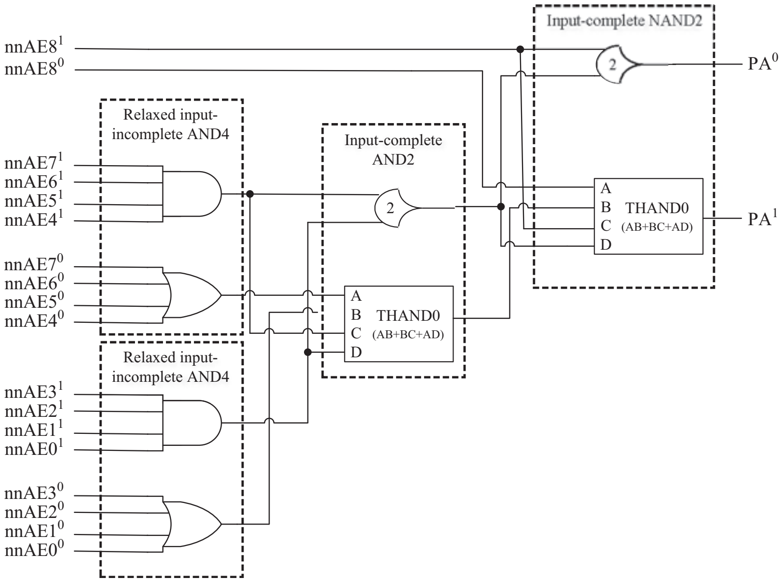

Figure 15.6 ISCAS-85 C432 M1 module nine-input NCL NAND that generates PA

input-complete two-input AND function, and then an input-complete two-input NAND function, as shown in Figure 15.6. To create a buggy version that was input-complete but not observable, either of the two gates comprising the two-input AND function in Figure 15.6 could be relaxed. The test times reported for the circuits are for testing every single gate for observability, even if a previous gate was found to be unobservable. Therefore, the time to detect a buggy circuit will be less than or equal to the reported times, since the rest of the gates would no longer need to be tested once an unobservable gate was identified. Z3 reported all bugs along with a counter example.

15.4 Equivalence verification for sequential NCL circuits

As described in Section 15.3.1, our equivalence verification methodology proved to be a fast and scalable approach for C/L NCL circuits. Hence, in this section we extend that approach to verify both safety and liveness of sequential NCL circuits, which is more complex due to datapath feedback.

To describe our methodology, we'll use an unsigned Multiply and Accumulate (MAC) unit as an example circuit. Figure 15.7(a) shows a synchronous MAC, where $A' = A + X \times Y$; and Figure 15.7(b) shows the equivalent NCL version. The four-phase QDI handshaking protocol utilized for NCL circuits requires at least $2N + 1$ NCL registers in a feedback loop that contains N DATA tokens, in order to avoid deadlock [12].

Hence, at least three NCL registers are needed in the MAC feedback loop to avoid deadlock, as shown in Figure 15.7(b). Although the synchronous and NCL MACs seem similar, they are structurally very different. Synchronous registers are clocked, whereas alternating DATA/NULL transitions in NCL are maintained via C-elements and a well-defined handshaking scheme. *Ki* and *Ko* are the external *request* input and *acknowledge* output, respectively.

Figure 15.8 shows the datapath diagram for a 4 + 2 × 2 NCL MAC with two C/L stages and four registers in the feedback loop (note that including a 4th register

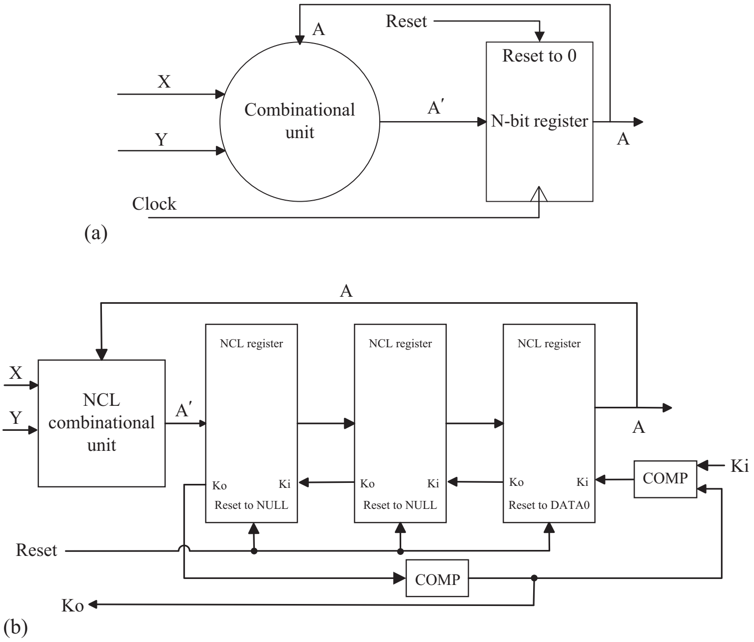

Figure 15.7 MAC circuit: (a) synchronous; (b) NCL

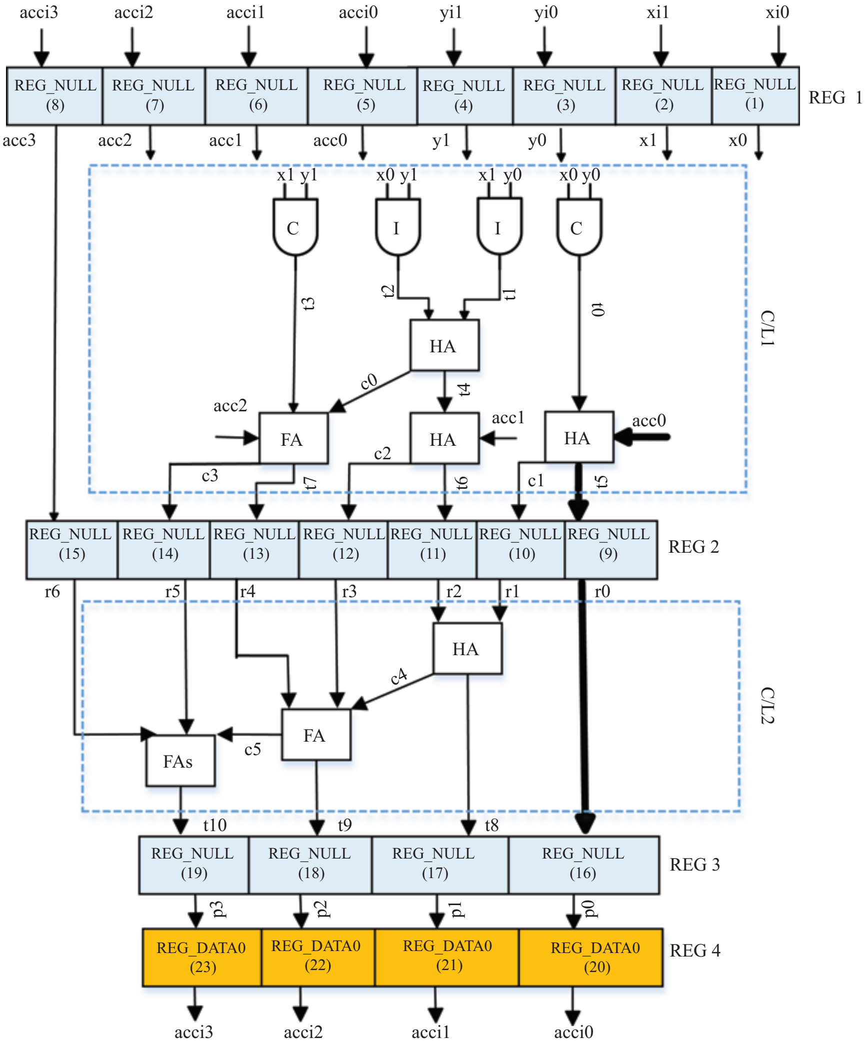

Figure 15.8 $4 + 2 \times 2$ NCL MAC datapath

in the feedback loop increases throughput compared to using the minimum required three registers, since this allows the DATA and NULL wavefronts to flow more independently [12]). (Xi_1, Xi_0) and (Yi_1, Yi_0) are the two bits of inputs *Xi* and *Yi*, respectively. The product of *Xi* and *Yi* is added with the 4-bit accumulator output, *Acci*, where $Acci_3$ and $Acci_0$ are the MSB and LSB, respectively. All signals shown in Figure 15.8 are dual-rail signals. *HA* and *FA* are the NCL half-adder and full-adder components, shown in Figure 15.1(d) and (e), respectively; and *FA* is a full-adder component without a *carry* output; hence, it utilizes two two-input XOR AQ11 functions, each comprised of two TH24comp gates (same as the HA *sum* output shown in Figure 15.1(d)), to compute its *sum* output. The highlighted components in Figure 15.8 are the NCL registers.

Figure 15.9(a) shows the netlist of the NCL $4 + 2 \times 2$ MAC, following the same structure as described in Section 15.3.1. The first two lines are the circuit inputs and outputs, respectively; lines 3–38 are the NCL threshold gates; lines 39–61 are the NCL registers; and lines 62–69 are C-elements used in the handshaking network.

15.4.1 Safety

Safety verification requires two steps. In the first step, we take a sequential NCL circuit and convert it to an equivalent synchronous circuit. We utilize the theory of WEB refinement [2] to compare the synchronous netlist generated from the NCL circuit with the original synchronous specification, as the notion of correctness. The major advantage of applying WEB refinement to the generated equivalent synchronous circuit instead of the actual NCL circuit is that a synchronous circuit is much more deterministic compared to its NCL equivalent, which makes the verification time much faster. The generated synchronous circuit, specification synchronous circuit, and the WEB-refinement property are automatically encoded in the SMT-LIB language. The resulting equivalence property is then checked using an SMT solver. In the second step, we check the invariant for each C/L stage, the same as previously discussed in Section 15.3.2.

The converted netlist (NCL-SYNC) is depicted in Figure 15.9(b). The conversion algorithm for sequential NCL circuits is slightly different than for C/L NCL circuits, described in Section 15.3.1, since the sequential NCL circuit contains reset-to-DATA registers, which are replaced with a 2-bit resettable synchronous register, 1 bit for each rail of the corresponding NCL dual-rail register. Like for C/L NCL circuits, all reset-to-NULL registers, handshaking signals, and C-elements are eliminated; and all C/L NCL gates are replaced with their corresponding relaxed (i.e., Boolean) gate.

The NCL-SYNC netlist must next be checked against the synchronous specification (SPEC-SYNC) netlist for equivalence. When verifying C/L NCL circuits, the circuit functionality could be specified as a Boolean function. However, since sequential circuits involve states and transitions, we use TSs as the formal model to capture the behaviors of both the NCL-SYNC netlist as well as the SPEC-SYNC netlist. The theory of WEB refinement [2] defines what it means for an implementation TS to be functionally equivalent to a specification TS. Therefore, we use the theory of WEB refinement for checking equivalence for sequential circuits.

The theory of WEB refinement allows for stutter between the implementation TS and the specification TS. What this means is that multiple but finite transitions of the implementation can match to a single specification transition. Rank functions (functions that map circuit states to natural numbers) are used to distinguish finite stutter from deadlock (infinite stutter). Another characteristic of WEB refinement is the use of refinement maps, which are functions that map implementation states to

```
1.  xi0_0,xi0_1,xi1_0,xi1_1,yi0_0,yi0_1,yi1_0,yi1_1
2.  acci0_0,acci0_1,acci1_0,acci1_1,…,acci3_0,acci3_1
3.  th22   x0_1,y0_1   t0_1
4.  thand0  y0_0,x0_0,y0_1,x0_1   t0_0
5.  th12   x1_0,y0_0   t1_0
6.  th22   x1_1,y0_1   t1_1
7.  th12   x0_0,y1_0   t2_0
8.  th22   x0_1,y1_1   t2_1
9.  th12   x1_0,y1_0   t3_0
10. th22   x1_1,y1_1   t3_1
11. th24comp   t2_0,t1_1,t1_0,t2_1   t4_0
12. th24comp   t2_0,t1_0,t2_1,t1_1   t4_1
13. th12   t2_0,t1_0   c0_0
14. th22   t1_1,t2_1   c0_1
15. th24comp   acc0_0,t0_1,t0_0,acc0_1   t5_0
16. th24comp   acc0_0,t0_0,acc0_1,t0_1   t5_1
17. th12   acc0_0,t0_0   c1_0
18. th22   t0_1,acc0_1   c1_1
19. th24comp   acc1_0,t4_1,t4_0,acc1_1   t6_0
20. th24comp   acc1_0,t4_0,acc1_1,t4_1   t6_1
21. th12   acc1_0,t4_0   c2_0
22. th22   t4_1,acc1_1   c2_1
23. th23   t3_0,acc2_0,c0_0   c3_0
24. th23   t3_1,acc2_1,c0_1   c3_1
25. th34w2   c3_1,t3_0,acc2_0,c0_0   t7_0
26. th34w2   c3_0,t3_1,acc2_1,c0_1   t7_1
27. th24comp   r1_0,r2_1,r2_0,r1_1   t8_0
28. th24comp   r1_0,r2_0,r1_1,r2_1   t8_1
29. th12   r1_0,r2_0   c4_0
30. th22   r2_1,r1_1   c4_1
31. th23   r4_0,r3_0,c4_0   c5_0
32. th23   r4_1,r3_1,c4_1   c5_1
33. th34w2   c5_1,r4_0,r3_0,c4_0   t9_0
34. th34w2   c5_0,r4_1,r3_1,c4_1   t9_1
35. th24comp   r5_0,r6_1,r6_0,r5_1   c6_0
36. th24comp   r5_0,r6_0,r5_1,r6_1   c6_1
37. th24comp   c5_0,c6_1,c6_0,c5_1   t10_0
38. th24comp   c5_0,c6_0,c5_1,c6_1   t10_1
39. Reg_NULL 1   xi0_0,xi0_1   KO2   ko1   x0_0,x0_1
40. Reg_NULL 1   xi1_0,xi1_1   KO2   ko2   x1_0,x1_1
41. Reg_NULL 1   yi0_0 yi0_1   KO2   ko3   y0_0 y0_1
42. Reg_NULL 1   yi1_0 yi1_1   KO2   ko4   y1_0 y1_1
43. Reg_NULL 1   acci0_0 acci0_1  KO2   ko5   acc0_0 acc0_1
44. Reg_NULL 1   acci1_0 acci1_1  KO2   ko6   acc1_0 acc1_1
45. Reg_NULL 1   acci2_0 acci2_1  KO2   ko7   acc2_0 acc2_1
46. Reg_NULL 1   acci3_0 acci3_1  KO2   ko8   acc3_0 acc3_1
47. Reg_NULL  2  t5_0  t5_1  ko16  ko9     r0_0  r0_1
48. Reg_NULL  2  c1_0  c1_1  KO3   ko10  r1_0  r1_1
49. Reg_NULL  2  t6_0 t6_1   KO3   ko11    r2_0 r2_1
50. Reg_NULL  2  c2_0 c2_1  KO3   ko12    r3_0 r3_1
51. Reg_NULL  2  t7_0 t7_1  KO3   ko13    r4_0 r4_1
52. Reg_NULL  2  c3_0 c3_1  KO3   ko14    r5_0 r5_1
53. Reg_NULL  2  acc3_0 acc3_1  KO3   ko15    r6_0 r6_1
54. Reg_NULL  3  r0_0 r0_1  ko20   ko16    p0_0 p0_1
55. Reg_NULL  3  t8_0 t8_1  ko21   ko17    p1_0 p1_1
56. Reg_NULL  3  t9_0 t9_1  ko22   ko18    p2_0 p2_1
57. Reg_NULL  3  t10_0 t10_1  ko23   ko19   p3_0 p3_1
58. Reg_DATA0 4  p0_0  p0_1   KO4   ko20   acci0_0  acci0_1
59. Reg_DATA0 4  p1_0  p1_1   KO5   ko21   acci1_0  acci1_1
60. Reg_DATA0 4  p2_0  p2_1   KO6   ko22   acci2_0  acci2_1
61. Reg_DATA0 4  p3_0  p3_1   KO7   ko23   acci3_0  acci3_1
62. C4   ko9,ko10,ko11,ko12  KO1
63. C4   ko13,ko14,ko15,KO1   KO2
64. C3   ko17,ko18,ko19  KO3
65. C2   Ki,ko5   KO4
66. C2   Ki,ko6   KO5
67. C2   Ki,ko7   KO6
68. C2   Ki,ko8   KO7
69. C4   ko1,ko2,ko3,ko4   KO
```

(a)

```
1.  xi0_1,xi1_1, yi0_1,yi1_1
2.  acci0_0,acci0_1,acci1_0,acci1_1,…,acci3_0,acci3_1
3.  not   xi0_1  xi0_0
4.  not   yi0_1  yi0_0
5.  not   xi1_1  xi1_0
6.  not   yi1_1  yi1_0
7.  th12   xi0_0,yi0_0   t0_0
8.  th22   xi0_1,yi0_1   t0_1
9.  th12   xi1_0,yi0_0   t1_0
10. th22   xi1_1,yi0_1   t1_1
11. th12   xi0_0,yi1_0   t2_0
12. th22   xi0_1,yi1_1   t2_1
13. th12   x1_0,y1_0   t3_0
14. th22   x1_1,y1_1   t3_1
15. th24comp   t2_0,t1_1,t1_0,t2_1   t4_0
16. th24comp   t2_0,t1_0,t2_1,t1_1   t4_1
17. th12   t2_0,t1_0   c0_0
18. th22   t1_1,t2_1   c0_1
19. th24comp   acci0_0,t0_1,t0_0,acci0_1   t5_0
20. th24comp   acci0_0,t0_0,acci0_1,t0_1   t5_1
21. th12   acci0_0,t0_0   c1_0
22. th22   t0_1,acci0_1   c1_1
23. th24comp   acci1_0,t4_1,t4_0,acci1_1   t6_0
24. th24comp   acci1_0,t4_0,acci1_1,t4_1   t6_1
25. th12   acci1_0,t4_0   c2_0
26. th22   t4_1,acci1_1   c2_1
27. th23   t3_0,acci2_0,c0_0   c3_0
28. th23   t3_1,acci2_1,c0_1   c3_1
29. th34w2   c3_1,t3_0,acci2_0,c0_0   t7_0
30. th34w2   c3_0,t3_1,acci2_1,c0_1   t7_1
31. th24comp   c1_0,t6_1,t6_0,c1_1   t8_0
32. th24comp   c1_0,t6_0,c1_1,t6_1   t8_1
33. th12   c1_0,t6_0   c4_0
34. th22   t6_1,c1_1   c4_1
35. th23   t7_0,c2_0,c4_0   c5_0
36. th23   t7_1,c2_1,c4_1   c5_1
37. th34w2   c5_1,t7_0,c2_0,c4_0   t9_0
38. th34w2   c5_0,t7_1,c2_1,c4_1   t9_1
39. th24comp   c3_0,acci3_1,acci3_0,c3_1   c6_0
40. th24comp   c3_0,acci3_0,c3_1,acci3_1   c6_1
41. th24comp   c5_0,c6_1,c6_0,c5_1   t10_0
42. th24comp   c5_0,c6_0,c5_1,c6_1   t10_1
43. Reg_0   t5_0  t5_1   acci0_0  acci0_1
44. Reg_0   t8_0  t8_1   acci1_0  acci1_1
45. Reg_0   t9_0  t9_1   acci2_0  acci2_1
46. Reg_0   t10_0  t10_1   acci3_0  acci3_1
```

(b)

Figure 15.9 (a) 4 + 2 × 2 NCL MAC netlist. (b) Converted synchronous equivalent netlist specification states. Refinement maps allow for the implementation and specification to be specified at significantly different abstraction levels. However, since the rail^1 registers of NCL-SYNC and the registers of SPEC-SYNC have a one-to-one mapping, there is no stutter between these two TSs, and the refinement is simply a projection of the rail^1 registers of the implementation state to the registers of the specification state. Therefore, the correctness proof obligations required for verifying WEB refinement can be reduced to the proof obligation depicted in Figure 15.10, where s is a state of NCL-SYNC; u is a SPEC-SYNC state obtained by projecting the values of the rail^1 registers from state s; $Step_{SYNC\_NCL}$ and $Step_{SYNC\_SPEC}$ are the functions that correspond to a single step of the NCL-SYNC circuit and the SPEC-SYNC circuit, respectively; w is the state obtained by stepping NCL-SYNC from state s; and v is the state obtained by stepping SPEC-SYNC from state u. The proof obligation states that the two circuits are functionally equivalent if for every state s of NCL-SYNC, the corresponding projection of values from the rail^1 registers of the w state are equivalent to the values of the corresponding registers in the v state. This proof obligation can be encoded in the SMT-LIB language, as shown below in *PO7*, and checked using an SMT solver.

$$\begin{aligned}
&PO7\colon \{\forall s :: s \in S_{SYNC\_NCL} :: \\
&\qquad [u = Reg\_Proj(s) \wedge w = Step_{SYNC\_NCL}(s) \wedge v = Step_{SYNC\_SPEC}(u)] \\
&\qquad \Rightarrow Reg\_Proj(w) = v\}.
\end{aligned}$$

After verifying function equivalence, the rail^0 outputs of each C/L stage must also be checked to ensure safety, as detailed in Section 15.3.2. Note that for sequential circuits, which include datapath feedback, the first invariant check method that checks the entire circuit simultaneously won't work; hence, the second, much faster method that performs the invariant check independently for each stage is utilized.

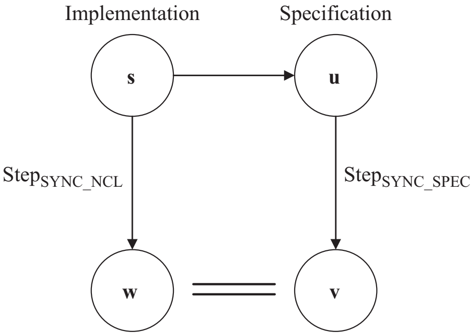

Figure 15.10 Depiction of proof obligation to check equivalence of NCL-SYNC and SPEC-SYNC netlists

15.4.2 Liveness

Figure 15.11 shows the handshaking connections for the 4 + 2 × 2 NCL MAC. Full-word completion is used by the input register, Reg 1, to generate a single *Ko*. Full-word completion is also utilized between Reg 1 and Reg 2, since bit-wise completion would have the same delay and require more area. Partial bit-wise

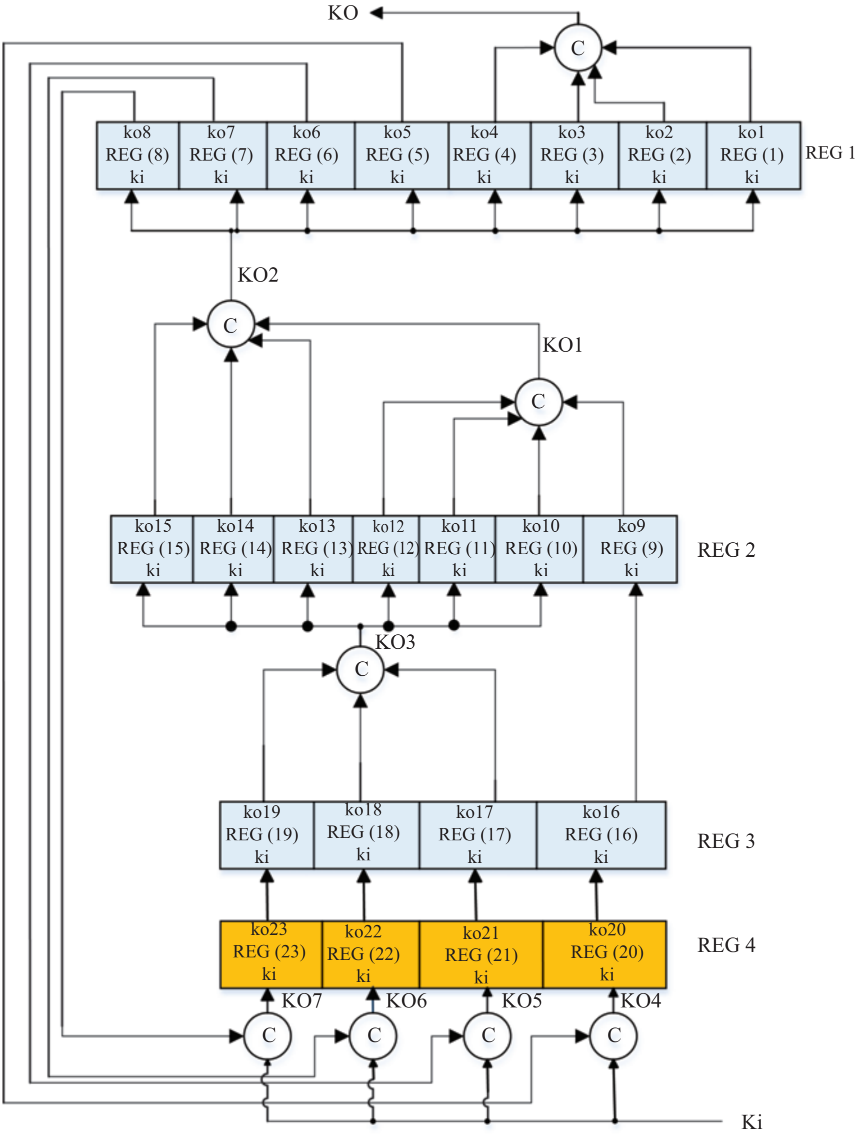

Figure 15.11 Handshaking connections for the 4 + 2 × 2 NCL MAC

completion is utilized between Reg 2 and Reg 3, since full bit-wise completion would have the same delay and require more area. Bit-wise completion is utilized between Reg 3 and Reg 4, and for the output register, Reg 4. The handshaking check for sequential NCL circuits is essentially the same as that for C/L NCL circuits, described in Section 15.3.3. The only addition is calculating a feedback register's level, which should be assigned the same level as other registers that share its *Ki* input signal, or one level more than its previous register, if its *Ki* input signal is not shared with another register already assigned a level. For the MAC example in Figure 15.11, feedback registers 5–8 would be assigned level 1, since they share their *Ki* input with the other level 1 registers, 1–4; and feedback register 15 would be assigned level 2, since it shares its *Ki* input with other level 2 registers, 9–14. Figure 15.12 shows the *reg_fanin* and *ko_fanout* lists for each register in the 4 + 2 × 2 NCL MAC example.

After verifying handshaking correctness, each stage's C/L must also be checked for input-completeness and observability, utilizing the methods detailed in Sections 15.3.4 and 15.3.5, respectively, to guarantee liveness.

1: reg_fanin: 0	ko_fanout: 0
2: reg_fanin: 0	ko_fanout: 0
3: reg_fanin: 0	ko_fanout: 0
4: reg_fanin: 0	ko_fanout: 0
5: reg_fanin: [20]	ko_fanout: [20]
6: reg_fanin: [21]	ko_fanout: [21]
7: reg_fanin: [22]	ko_fanout: [22]
8: reg_fanin: [23]	ko_fanout: [23]
9: reg_fanin: [1, 3, 5]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
10: reg_fanin: [1, 3, 5]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
11: reg_fanin: [1, 2, 3, 4, 6]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
12: reg_fanin: [1, 2, 3, 4, 6]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
13: reg_fanin: [1, 2, 3, 4, 7]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
14: reg_fanin: [1, 2, 3, 4, 7]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
15: reg_fanin: [8]	ko_fanout: [1, 2, 3, 4, 5, 6, 7, 8]
16: reg_fanin: [9]	ko_fanout: [9]
17: reg_fanin: [10, 11]	ko_fanout: [10, 11, 12, 13, 14, 15]
18: reg_fanin: [10, 11, 12, 13]	ko_fanout: [10, 11, 12, 13, 14, 15]
19: reg_fanin: [10, 11, 12, 13, 14, 15]	ko_fanout: [10, 11, 12, 13, 14, 15]
20: reg_fanin: [16]	ko_fanout: [16]
21: reg_fanin: [17]	ko_fanout: [17]
22: reg_fanin: [18]	ko_fanout: [18]
23: reg_fanin: [19]	ko_fanout: [19]

Figure 15.12 reg_fanin and ko_fanout lists for the 4 + 2 × 2 NCL MAC

Table 15.6 Verification results for sequential NCL circuits

Circuits	**Functional check (s)**	**Handshaking check (s)**	**Total time (s)**
ISCAS s27	0.01	0.0019	0.0119
4 + 2 × 2 MAC	0.01	0.0045	0.0145
8 + 4 × 4 MAC	0.05	0.7852	0.8352
12 + 6 × 6 MAC	0.77	2.331	3.101
16 + 8 × 8 MAC	47.55	21.7411	69.2911
20 + 10 × 10 MAC	2,643.99	163.6463	2,807.6363
20 + 10 × 10 MAC-B1	0.11 (B)	163.6463	163.7563
20 + 10 × 10 MAC-B2	0.13 (B)	163.6463	163.7763
20 + 10 × 10 MAC-B3	2,643.99	169.8422 (B)	2,813.8322
20 + 10 × 10 MAC-B4	2,643.99	159.3253 (B)	2,803.3153
20 + 10 × 10 MAC-B5	0.20 (B)	163.6463	163.8463

15.4.3 Sequential NCL circuit results

The verification results for sequential NCL circuits, including functional equivalence and handshaking checks, are shown in Table 15.6. Since the invariant, input-completeness, and observability checks are exactly the same for combinational and sequential NCL circuits, these results are not included in Table 15.6. Test circuits include multiple MAC units and one ISCAS-89 benchmark, s27 [22]. The MAC units are represented as $A + M \times N$, where A, M, and N represent the length of the accumulator, multiplicand, and multiplier, respectively. The same types of bugs were tested for the MACs as tested for the multipliers, and the same machine was used to perform the sequential circuit verification, both as described at the end of Section 15.3.3. Z3 reported all functional bugs along with a counter example, and our handshaking check tool identified and reported the location of all inserted completion logic bugs.

15.5 Conclusions and future work

This chapter presents a novel methodology for formally verifying the correctness (both safety and liveness) of combinational and sequential NCL circuits. The approach includes methods for ensuring handshaking correctness, and functional correctness of both rail^1 and rail^0 outputs, and methods to ensure that NCL C/L circuits, or pipeline stages, are both input-complete and observable, which is required for correct operation under all timing scenarios. The presented methodology is applicable to both NCL circuits designed using only NCL gates with hysteresis and AQ12 relaxed NCL circuits, where NCL gates with hysteresis are replaced with their Boolean equivalent gate when hysteresis is not required for input-completeness and/or observability.

The framework of this verification methodology can also be applied to other QDI paradigms, such as MTNCL and PCHB. For MTNCL, the functional checking AQ13 and invariant checking methods are essentially the same as for NCL, but the handshaking check is slightly different [23]. Additionally, MTNCL circuits do not require input-completeness or observability, so these checks are not needed. For PCHB, the handshaking check is essentially the same as for NCL, but the functional checking method is a bit different [11]. Since PCHB gates consist of dual-rail input(s) and output(s), invariant, input-completeness, and observability checking are not required, as these are ensured within the primitive PCHB gates themselves.

References

[1] V. Wijayasekara, S.K. Srinivasan, and S.C. Smith, "Equivalence verification for NULL Convention Logic (NCL) circuits," *32nd IEEE International Conference on Computer Design* (ICCD), pp. 195–201, October 2014.

[2] P. Manolios, "Correctness of pipelined machines," in *FMCAD 2000*, ser. LNCS, W.A. Hunt, Jr. and S.D. Johnson, Eds., vol. 1954, Springer-Verlag, 2000, pp. 161–178. AQ14

[3] D. Monniaux, "A survey of Satisfiability Modulo Theory" [online]. Available: https://hal.archives-ouvertes.fr/hal-01332051/document [Accessed: May 5, 2019].

[4] C.W. Moon, P.R. Stephan, and R.K. Brayton, *Journal of VLSI Signal Processing* (1994) 7: 85. AQ15

[5] C.J. Myers, *Asynchronous Circuit Design*. New York: Wiley, 2001.

[6] A. Semenov and A. Yakovlev, "Verification of asynchronous circuits using time Petri net unfolding," *33rd Design Automation Conference*, Las Vegas, NV, USA, 1996, pp. 59–62.

[7] F. Verbeek and J. Schmaltz, "Verification of building blocks for asynchronous circuits," in *ACL2*, ser. EPTCS, R. Gamboa and J. Davis, Eds., vol. 114, 2013, pp. 70–84. AQ16

[8] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, "Click elements: An implementation style for data-driven compilation," *IEEE Symposium on Asynchronous Circuits and Systems*, 2010, pp. 3–14.

[9] S.K. Srinivasan and R.S. Katti, "Desynchronization: design for verification," in *FMCAD*, P. Bjesse and A. Slobodova, Eds., 2011, pp. 215–222.

[10] A.A. Sakib, S.C. Smith, and S.K. Srinivasan, "Formal modeling and verification for pre-charge half buffer gates and circuits," *60th IEEE International Midwest Symposium on Circuits and Systems*, Boston, MA, 2017, pp. 519–522.

[11] A.A. Sakib, S.C. Smith, and S.K. Srinivasan, "An equivalence verification methodology for combinational asynchronous PCHB circuits," *61st IEEE International Midwest Symposium on Circuits and Systems*, Windsor, ON, Canada, 2018, pp. 767–770.

[12] S.C. Smith and J. Di, "Designing asynchronous circuits using NULL Convention Logic (NCL)," in *Synthesis Lectures on Digital Circuits and Systems*, Morgan & Claypool Publishers, vol. 4/1, July 2009.

[13] C. Jeong and S.M. Nowick, "Optimization of robust asynchronous circuits by local input completeness relaxation," *Asia and South Pacific Design Automation Conference*, Jan. 2007, pp. 622–627.

[14] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, "Checking delay-insensitivity: 10^4 gates and beyond," *8th International Symposium on Asynchronous Circuits and Systems*, April 2002, pp. 149–157.

[15] L.M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," *in TACAS*, ser. Lecture Notes in Computer Science, C.R. Ramakrishnan and J. Rehof, Eds., vol. 4963, Springer, 2008, pp. 337–340.

[16] D. Bryan, "The ISCAS '85 benchmark circuits and netlist format" [online]. Available: https://ddd.fit.cvut.cz/prj/Benchmarks/iscas85.pdf [Accessed: May 5, 2019].

[17] C.L. Seitz, "System timing," in *Introduction to VLSI Systems*, Boston, MA, USA: Addison Wesley Longman Publishing Co., Inc., 1979, pp. 218–262.

[18] "ISCAS-85 c432 27-channel interrupt controller" [online]. Available: http://web.eecs.umich.edu/~jhayes/iscas.restore/c432.html [Accessed: May 5, 2019].

[19] S.C. Smith, R.F. DeMara, J.S. Yuan, D. Ferguson, and D. Lamb, "Optimization of null convention self-timed circuits," *Integr. VLSI J.* (2004) 37(3): 135–165.

[20] "ISCAS-85 c432 27-channel interrupt controller Module M3" [online]. Available: http://web.eecs.umich.edu/~jhayes/iscas.restore/c432m3.html [Accessed: May 5, 2019].

[21] "ISCAS-85 c432 27-channel interrupt controller Module M1" [online]. Available: http://web.eecs.umich.edu/~jhayes/iscas.restore/c432m1.html [Accessed: May 5, 2019].

[22] [Online] http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/bench/, [Accessed: May 5, 2019].

[23] M. Hossain, A.A. Sakib, S.K. Srinivasan, and S.C. Smith, "An equivalence verification methodology for asynchronous sleep convention logic circuits," *IEEE International Symposium on Circuits and Systems*, May 2019.

Chapter 15

Formal verification of NCL circuits

Author Queries

AQ1: Please check the sentence “Testing-based approaches have …” for sense and revise if needed.

AQ2: Please spell out “FDIV”, if needed.

AQ3: Please spell out “NCL” and “QDI” at first use, if needed.

AQ4: Please provide the city, state code (if USA) and country names for the affiliation of the authors

AQ5: Please spell out “PCHB” at first use, if needed.

AQ6: Please spell out “SOP”, if needed.

AQ7: Please confirm the steps after the sentence “The NCL equivalence verification method requires five steps …” have been set correctly.

AQ8: Please check in Figure 15.1, part (a) has been set correctly.

AQ9: Please spell out “MSB” at first use, if needed.

AQ10: Please spell out “PA” at first use, if needed.

AQ11: Please spell out “LSB”, if needed.

AQ12: Please approve edit to the sentence “The presented methodology is applicable …”.

AQ13: Please spell out “MTNCL” at first use, if needed.

AQ14: Please provide publisher location for Refs. [2, 12, 15].

AQ15: Please provide article title for Ref. [4]; please also check the page range.

AQ16: Please provide publisher name and location for Refs. [7, 8, 9].