Exploiting Dual-Rail Register Invariants for Equivalence Verification of NCL Circuits

Son N. Le
Electrical and Computer Engineering
North Dakota State University
Fargo, USA
son.ngoc.le@ndsu.edu

Sudarshan K. Srinivasan
Electrical and Computer Engineering
North Dakota State University
Fargo, USA
sudarshan.srinivasan@ndsu.edu

Scott C. Smith
Electrical Engineering and Computer Science
Texas A&M University - Kingsville
Kingsville, USA
scott.smith@tamuk.edu

***Abstract*—Equivalence checking is one of the most scalable and useful verification techniques in industry. NULL Convention Logic (NCL) circuits utilize dual-rail signals (i.e., two wires to represent one bit of DATA), where the wires are inverses of each other during a DATA wavefront. In this paper, a technique that exploits this invariant at NCL register boundaries is proposed to improve the efficiency of equivalence verification of NCL circuits.**

Keywords*—*asynchronous circuits, formal verification, formal methods, equivalence checking, NULL Convention Logic

I. Introduction

NULL Convention Logic (NCL) [1] is one type of Quasi-Delay Insensitive (QDI) asynchronous circuit design paradigm, which has benefits over its synchronous counterpart by being more robust to process, voltage, and temperature (PVT) variations, allowing NCL circuits to operate in environments with extreme high or low temperatures, or with large temperature fluctuations, and with an ultra-low supply voltage [2]. Hence, NCL circuits are ideal candidates for applications in space exploration, the power industry, the automotive industry, oil/gas exploration, medical imaging instrumentation, the laser industry, superconducting computing and energy storage systems, and wireless sensor nodes or other low voltage or low power applications.

NCL circuits do not utilize a clock signal for synchronization, instead NCL utilizes multi-rail logic, such as dual-rail, along with a 4-phase handshaking protocol to achieve delay-insensitivity. A dual-rail signal, *D*, consists of two wires, D^0 and D^1, which may assume any value from the set {DATA0, DATA1, NULL}. The DATA0 state ($D^0 = 1$ and $D^1 = 0$) corresponds to a Boolean logic 0, the DATA1 state ($D^0 = 0$ and $D^1 = 1$) corresponds to a Boolean logic 1, and the NULL state ($D^0 = 0$ and $D^1 = 0$) corresponds to the empty set meaning that the value of *D* is not yet available. The two rails are mutually exclusive, such that both rails can never be asserted simultaneously; this state is defined as an ILLEGAL state. Hence, when an NCL register is DATA, its rail^1 and rail^0 outputs are inverses of each other, referred to herein as the Dual-Rail Register Invariant.

One of the very effective formal verification techniques for digital system design is equivalence checking. In commercial design cycles, the design is continuously optimized for performance, power, and area. Significant time and effort are invested in design testing. If the design is further optimized, it is not feasible to exhaustively test the design again. Equivalence checking has been found to be very effective to address this problem as the verified design can be checked against the more optimized design directly. Equivalence checking has been previously extended to NCL circuits [3], where the NCL circuits are checked for equivalence against their synchronous counterparts. In this paper, the Dual-Rail Register Invariant is exploited to speed up the previous equivalence checking method for NCL circuits, which is demonstrated by using identical benchmarks to show the improvement in verification times.

II. Related Work

In the NCL equivalence checking method presented in [3], the NCL netlist is first converted to a synchronous netlist. This conversion has three steps. First, the NCL threshold gates are replaced with their Boolean set function; and gate hysteresis is ignored. Second, rail^1 of each dual-rail input is replaced with a Boolean input; and each rail^0 input is generated by negating its corresponding Boolean input. Third, the reset-to-NULL registers (henceforth termed as Reg_NULL) are removed, and their inputs connected directly to their outputs. Each dual-rail Reset-to-DATA register is replaced by a 2-bit synchronous register. Well Founded Equivalence Bisimulation (WEB) refinement [4] is used as the notion of equivalence to compare the synchronous version of the NCL circuit and the synchronous specification circuit. The WEB refinement property is checked using an SMT solver. This approach was found to be very scalable because the equivalence verification is performed at the synchronous-level instead of directly verifying the NCL circuit, which is difficult due to nondeterministic signal transitions in NCL circuits. Section III presents the proposed Dual-Rail Register Invariant technique that modifies the conversion technique described above for the register components to improve efficiency.

III. Equivalence Verification

The 3×3 unsigned NCL multiplier that implements the function *p(5:0) = xi(2:0) × yi(2:0)*, as shown in Fig. 1 without its completion logic, will be used as the example circuit to show the circuit transformation done in the previous work and contrast that to the proposed Dual-Rail Register Invariant. It is comprised of several components including dual-rail inputs and outputs, input-complete NCL AND functions (represented with a C inside the AND symbol), input-incomplete NCL AND functions (represented with an I inside the AND symbol), NCL Half-Adders (HA) and Full-Adders (FA), and dual-rail Reset-to-NULL registers (REG_NULL).

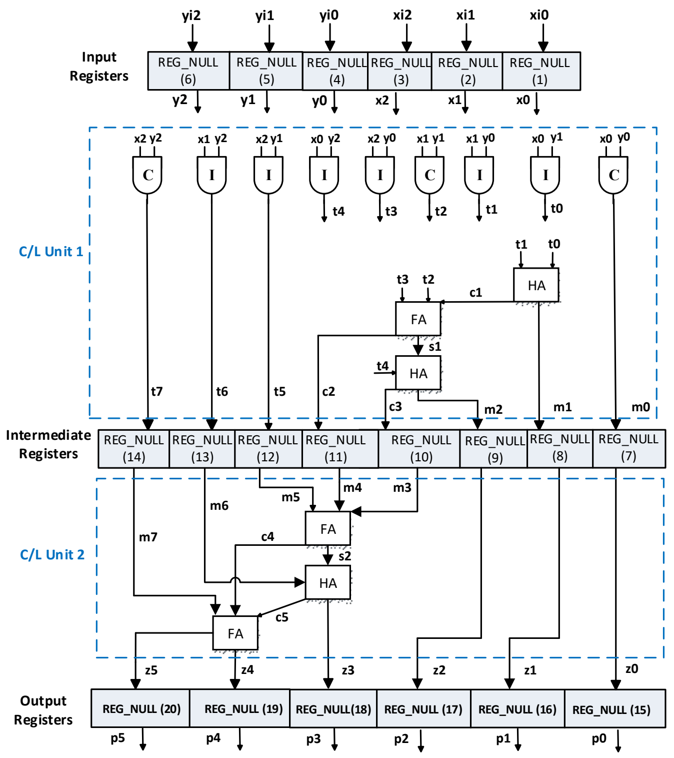

Fig. 1. 3×3 NCL multiplier circuit [3]

To accompany the circuit in Fig. 1, its netlist, and the netlist of the previous work's circuit transformation, are shown in Fig. 2(a) and 2(b), respectively. In Fig. 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, 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 3 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 Cn, 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.

A. *Previous Circuit Transformation*

For Fig. 2(b), each NCL gate from Fig. 2(a) is replaced with its corresponding Boolean gate without hysteresis, and the dual-rail primary inputs are replaced by their respective rail^1 input, which are then complemented by inserting invertors (lines 3-8) to generate their corresponding rail^0 signals. The Reg_NULL components are removed by connecting their inputs to their outputs; and the handshaking C-elements are also removed.

B. *Proposed Dual-Rail Register Invariant*

Instead of removing the Reg_NULL components by connecting their inputs to their outputs, the proposed Dual-Rail Register Invariant removes the Reg_NULL components by connecting their rail^1 inputs to their corresponding rail^1 outputs, and then generates each rail^0 output by inverting its corresponding rail^1 input. This transformation is possible due to the inherent NCL property where the rail^1 and rail^0 values are inverses of each other in the DATA phase. Note that both this inverse signal property and correctness of the NULL phase are checked as part of the NCL formal verification method presented in [3]. The proposed Dual-Rail Register Invariant allows the SMT solver to trim the circuit by removing all logic solely used to generate the Reg_NULL rail^0 inputs, replacing this instead with a single inverter, as shown in Fig. 2(c).

C. *Proof Obligation*

The proof obligation for equivalence verification is the same for the original approach and the proposed approach that exploits the Dual-Rail Register Invariant. We now describe the proof obligation. Consider an NCL circuit with *p* inputs and *q* outputs. Note that NCL inputs and outputs are dual-rail. The corresponding synchronous circuit will have *p* Boolean inputs and *q* Boolean outputs. We step both the specification synchronous circuit and the NCL-reduced synchronous circuit, with the same symbolic inputs $i^1, \ldots, i^p$, to generate outputs $O_{sync}^1, \ldots, O_{sync}^q$, and $O_{NCL}^1, \ldots, O_{NCL}^q$, respectively. The proof obligation itself is constructed using the predicates from Table 1: $\{p_0 \wedge p_1\} \rightarrow p_2$. p_0 corresponds to the symbolic step of the NCL reduced synchronous circuit, and p_1 corresponds to the symbolic step of the synchronous specification circuit. p_2 is the equivalence predicate, which states that the rail^1's of the NCL reduced synchronous outputs should be equal to the synchronous specification circuit outputs.

IV. Results

For comparison, the same unsigned NCL Multiply and Accumulate (MAC) circuits as in [3], with increasing operand sizes to show scalability, were used. These implement the function *acci* = *acci* + *xi* × *yi*, as shown in Fig. 3 for a 4+2×2 NCL MAC, without its completion logic. As shown in the Fig. 3 example, each MAC's C/L is partitioned into 2 stages by inserting a Reset-to-NULL register between the last carry-save adder and the final ripple-carry adder; and the feedback loop

TABLE I. Equivalence Checking Predicates

p_n	Predicate
p_0	$(O_{NCL}^1, \ldots, O_{NCL}^q) = NCLStep(i^1, \ldots, i^p)$
p_1	$(O_{sync}^1, \ldots, O_{sync}^q) = SyncStep(i^1, \ldots, i^p)$
p_2	$\bigwedge_{n=1}^{n=q} \left(rail1({o_{NCL}}^n) = {o_{sync}}^n\right)$

```
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)

```
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. not    p0_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.  not    t6_1   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.  not    t5_1   t5_0
25.  th22  xi2_1,yi2_1  t7_1
26.  not    t7_1   t7_0
27.  th24comp    t0_0,t1_0,t0_1,t1_1   p1_1
28.  not    p1_1   p1_0
29.  th22    t0_1, t1_1   c1_1
30.  th12    t0_0,t1_0   c1_0
31.  th23    t3_1,t2_1,c1_1   c2_1
32.  not    c2_1   c2_0
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.  not    p2_1   p2_0
37.  th22    s1_1,t4_1   c3_1
38.  not    c3_1   c3_0
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.  not    p3_1   p3_0
45.  th22    s2_1,t6_1   c5_1
46.  th12    s2_0,t6_0   c5_0
47.  th23     t7_1,c4_1,c5_1   p5_1
48.  not    p5_1   p5_0
49.  th34w2     p5_0,t7_1,c4_1,c5_1   p4_1
50.  not    p4_1   p4_0
```

(c)

Fig. 2. (a) 3×3 NCL multiplier netlist (b) converted netlist using method in [3] (c) converted netlist using proposed Dual-Rail Register Invariant

TABLE II. VERIFICATION RESULTS

Circuit	Speedup
$8+4\times 4\ MAC$	1.14
$12+6\times 6\ MAC$	1.17
$16+8\times 8\ MAC$	2.75
$20+10\times 10\ MAC$	1.31
$22+11\times 11\ MAC$	3.63
$24+12\times 12\ MAC$	TO/67,599 sec
$16+8\times 8\ MAC-5\ Reg$	3.44
$20+10\times 10\ MAC-5\ Reg$	1.46

contains 4 registers for increased performance. Note that the proposed Dual-Rail Register Invariant can also be applied to Reset-to-DATA registers, resulting in their replacement with a single synchronous register, plus an inverter to generate the rail^0 output, instead of the previous conversion technique that required 2 synchronous registers, as described in Section II.

The Z3 SMT solver [5] was used to check for equivalence, but any combinational equivalence checker could be used. Table II lists the verification results, where the first column indicates the MAC size, and the second column is speedup (i.e., equivalence verification time using the method described in [3] divided by equivalence verification time using the proposed Dual-Rail Register Invariant). Timeout (TO) denotes that the verification time exceeded one day. The last 2 rows in Table II are for MACs with an additional Reset-to-NULL register inserted between the partial product generation circuitry (i.e., AND functions) and the first carry-save adder.

The results show speedups ranging from 14% - 263% for the various 4-register MACS, and an additional speedup of 25% and 11% when adding an extra 5th register stage in the 16+8×8 and 20+10×10 MACs, respectively. Note that the 24+12×12 MAC timed out using the previous approach in [3], but was successfully verified in less than 1 day utilizing the proposed Dual-Rail Register Invariant.

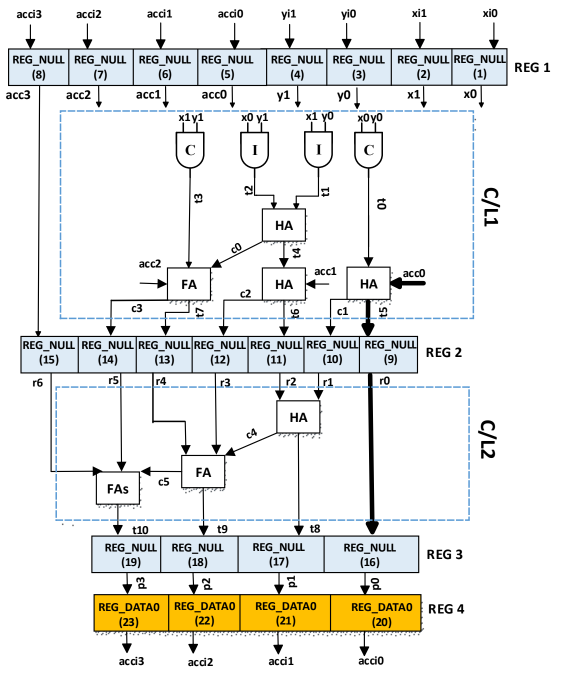

Fig. 3. 4+2×2 NCL MAC [3]

V. CONCLUSIONS AND FUTURE WORK

This paper proposes the Dual-Rail Register Invariant technique to speedup equivalence checking of NCL circuit implementations with respect to their Boolean/synchronous specifications. It has been shown to significantly reduce equivalence checking times, and to further speedup equivalence checking when additional pipeline stages are added to the NCL circuit, as this allows for more usage of the technique.

The proposed Dual-Rail Register Invariant technique is also applicable to speedup equivalence verification of Sleep Convention Logic (SCL) circuits [6], as the circuit transformation and safety verification are the same for SCL and NCL, only liveness verification differs [7].

Future work includes investigating additional refinements and invariants to potentially further reduce verification time, such as applying the dual-rail invariant described herein, to generate rail^0 as the inverse of rail^1, at the outputs of every NCL C/L function (e.g., HA, FA, AND), instead of only at register boundaries. This would require NCL C/L functions to be reconstructed from a flattened NCL gate netlist, where the two wires of a dual-rail signal are disassociated. Additionally, the invariant check from [3], which ensures that the rail^1 and rail^0 outputs of each NCL register are always inverses of each other during the DATA phase, would also need to be checked at the output of each NCL C/L function, instead of only at the register boundaries.

Acknowledgement: This paper is based upon work supported by the National Science Foundation under Grant No. CCF-1717420.

REFERENCES

[1] K. M. Fant and S. A. Brandt, “NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis,” *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261-273, August 1996.

[2] J. Di and S. C. Smith, “Asynchronous Digital Circuits,” in *Extreme Environment Electronics*, pp. 663 – 673, CRC Press, November 2012.

[3] A. A. Sakib, S. Le, S. C. Smith, and S. K. Srinivasan, “Formal Verification of NCL Circuits,” in *Asynchronous Circuit Applications*, pp. 309-338, IET, December 2019.

[4] 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.

[5] 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.

[6] L. Zhou, R. Parameswaran, F. Parsan, S. C. Smith, and J. Di, “Multi-Threshold NULL Convention Logic (MTNCL): An Ultra-Low Power Asynchronous Circuit Design Methodology,” *Journal of Low Power Electronics and Applications*, Vol. 5/2, pp. 81-100, May 2015.

[7] 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*, pp. 1-5, May 2019.