2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems

Uncle – An RTL Approach to Asynchronous Design

Robert B. Reese
Electrical and Computer Engr. Dept.
Mississippi State University
Starkville, MS, USA

Scott C. Smith
Dept. of Electrical Engr.
University of Arkansas
Fayetteville, AR, USA

Mitchell A. Thornton
Dept. Computer Science and Engr.
Southern Methodist University
Dallas, TX, USA

***Abstract*— Uncle (Unified NULL Convention Logic Environment) is an end-to-end toolset for creating asynchronous designs using NULL Convention Logic (NCL). Designs are specified in Verilog RTL, with the user responsible for specifying registers, datapath elements, and finite state machines for controlling datapath sequencing. A commercial synthesis tool is used to produce a gate-level netlist of primitive logic gates and storage elements, which is then transformed into an NCL netlist by the Uncle mapping flow. Performance optimizations supported by the flow are net buffering for target slew and delay balancing between latch stages. Both data-driven and control-driven (i.e. Balsa-style) schemes are supported. Transistor count, performance, and energy comparisons are made for Uncle versus Balsa-generated netlists for GCD and Viterbi decoder designs, with the Uncle designs comparing favorably in all three areas.**

Keywords-asynchronous; NULL Convention Logic; synthesis; RTL

I. INTRODUCTION

Advantages such as lower power, reduced EMI, robustness to environmental variations, and scalability have long been touted for asynchronous design versus clocked design. New startups such as [1] continue to appear, hoping to capitalize on these and other advantages of asynchronous design. Fortunately, the asynchronous research community has diligently worked over the last two decades (at least) to create an excellent array of asynchronous design tools. This paper discusses a toolset that adds new capabilities for those designers who want to create NCL-based systems.

Uncle (Unified NULL Convention Logic Environment) is a toolset for creating delay-insensitive dual-rail asynchronous designs based on NULL Convention Logic (NCL). An early version of the Uncle toolset was used by Camgian Microsystems to create a 65nm chip that investigates dynamic V_{DD} control of NCL blocks [2]. The paper organization presents a brief background on NCL, the details of the Uncle methodology, and some example designs with comparisons against Balsa [3][4], a mature and well-recognized toolset in the asynchronous community. Comparisons of the Uncle methodology versus the Balsa methodology are made continually throughout the paper, as appropriate.

II. NULL CONVENTION LOGIC

NULL Convention Logic (NCL) [5][6][7] is a four-phase dual-rail delay-insensitive logic style based on threshold logic. NCL circuits are comprised of 27 fundamental THmn threshold gates, where at least *m* of the *n* inputs must be asserted before the output will become asserted. These 27 gates constitute all functions of four or fewer variables. NCL gates are designed with *hysteresis* state-holding capability, such that after the output is asserted, all inputs must be deasserted before the output will be deasserted. Hysteresis ensures a complete transition of inputs back to NULL (spacer) before asserting the output associated with the next wavefront of input data. Therefore, NCL gates have both *set* and *hold1* equations, where the *set* equation determines when the gate will become asserted and the *hold1* equation determines when the gate will remain asserted once it has been asserted. The *set* equation determines the gate's functionality as one of the 27 NCL gates, whereas the *hold1* equation is simply all inputs ORed together. The general equation for an NCL gate with output Z is: $Z = set + (Z^- \bullet hold1)$, where Z^- is the previous output value and Z is the new value. For example, a THnn gate is equivalent to an *n*-input C-element and a TH1n gate is equivalent to an *n*-input OR gate.

To implement an NCL gate using CMOS technology, an equation for the complement of Z is also required, which in general form is: $Z' = reset + (Z^{-}' \bullet hold0)$, where *reset* is the complement of *hold1* (i.e., the complement of each input, ANDed together) and *hold0* is the complement of *set*, such that the gate output is deasserted when all inputs are deasserted, and then remains deasserted while the gate's *set* condition is false. To achieve hysteresis state-holding behavior, the new output value, Z, depends on the previous output value, Z^-, which requires internal gate feedback. For the static realization, the equations for Z and Z', given above, are directly implemented in NMOS and PMOS logic, respectively, after simplifying.

For example, the *set* equation for the TH23 gate is AB + AC + BC, and the *hold1* equation is A + B + C; therefore, the gate is asserted when at least 2 inputs are asserted and it then remains asserted until all inputs are deasserted. The *reset* equation is A'B'C' and the *hold0* equation is A'B' + B'C' + A'C'. Directly implementing these equations for Z and Z', after simplification, yields the static transistor-level realization shown in Figure 1.

Since NCL utilizes 27 fundamental state-holding gates for circuit design, rather than only C-elements, it has a greater potential for optimization than other delay-insensitive paradigms. For example, Figure 2 compares an input-complete NCL AND function with its DIMS [18] counterpart, showing

This work partially funded by NSF-CCF-1116405.

1522-8681/12 $26.00 © 2012 IEEE
DOI 10.1109/ASYNC.2012.14

that the NCL version requires far fewer gates (2 vs. 5), transistors (31 vs. 56), and gate delays (1 vs. 2). Other key components show similar savings; for example an NCL full adder requires 4 vs. 12 gates and 80 vs. 168 transistors, whereas an input-complete 2-input NCL MUX requires 5 vs. 10 gates and 80 vs. 148 transistors. All four circuits have two gate delays (Note: In this paper, a dual rail signal has *t_/f_* prefixes for true/false rails, respectively).

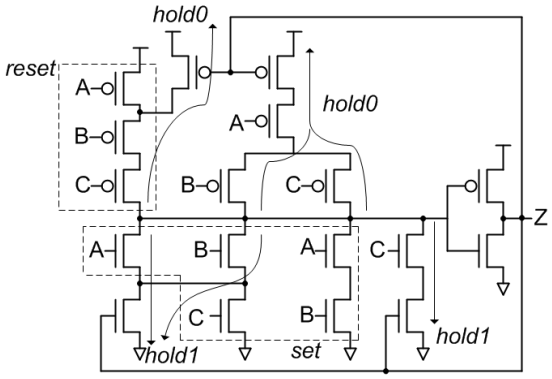

Figure 1. Static TH23 implementation.

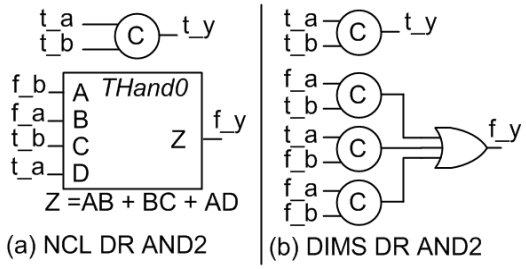

Figure 2. NCL/DIMS DR AND2 Implementations.

III. NCL Asynchrononous Systems

The previous section discussed the basics of implementing dual-rail Boolean logic using NCL gates. However, a complete system requires registers for data storage and a sequencing mechanism. Uncle supports two approaches for implementing registers and control; a data-driven approach that uses NCL gates for both registers and control, and a control-driven approach that uses Balsa-style registers and control. In both cases, the combinational logic between registers is implemented in NCL gates.

A. Data-driven Approach

Figure 3a shows a data-driven dual-rail half-latch (Balsa calls this a *PassivatorPush* component). In this system, the acknowledge signals *ki*, *ko* are at logic 1 when the data rails are at NULL. Because of this, an asynchronous reset signal is required in the C-element to force its output to NULL during system reset. This is a reset-to-NULL half-latch as both outputs are reset to 0 during a system reset.

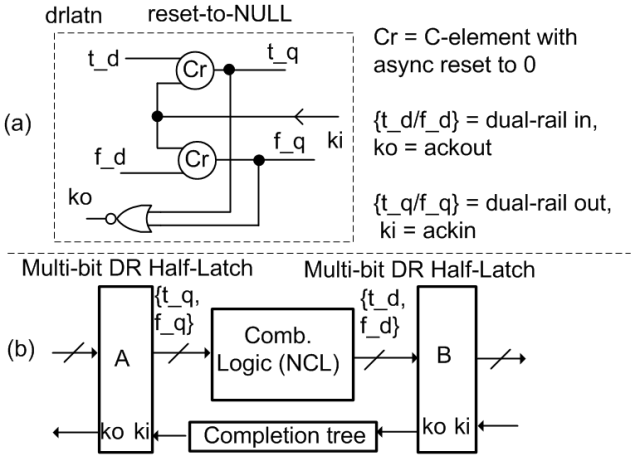

Figure 3. Data-driven dual-rail half-latch and usage.

Figure 3b shows how the acknowledge signals are used to control data transfer between two of these half-latches in the familiar micro pipeline arrangement. The ackin (*ki*) of a bit in latch A is tied to the output of a C-element completion tree whose inputs are the B-latch ackouts (*ko*) of all the destinations of that bit. The data sequencing between bits in the registers is controlled by arrival of data waves and NULL waves at the half-latch, and by the ack network.

A finite state machine with feedback requires a different form of latch element. If the C-element of Figure 3 that drives the *t_q* output is replaced with a C-element that resets to 1, then this becomes a reset-to-DATA1 (*drlats*) half-latch (the latch outputs have a dual-rail DATA1 at system reset). Conversely, a reset-to-DATA0 (*drlatr*) half-latch is formed by replacing the C-element driving the *f_q* output with a C-element that resets to 1. Figure 4 shows a finite state machine implementation with state registers implemented as three half-latches, with the middle half-latch containing initial data. This forms a three half-latch ring, which is the minimum required for data cycling; and the initial data in the middle half-latch is required in order to insert a data token in this loop.

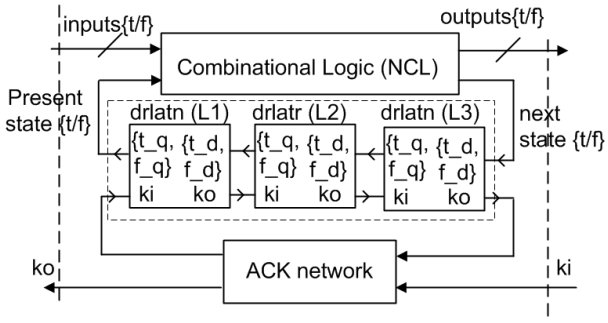

Figure 4. Finite State Machine.

This paper refers to a system using this style of registers/control as data-driven, since there is no separate control network other than the ack network. In the data-driven

style, all ports and all registers are read and written every compute cycle.

B. Control-driven Approach

Conversely, this paper refers to a *control-driven* system as one that has registers with selective read/writes and a control network that is separate from the datapath, such as that implemented by Balsa. Figure 5a shows a dual-rail register based on an SR-latch (this register has a low-true *ko*; Balsa uses a register with a high-true *ko*). Any number of read ports can be easily added to the register by placing AND2 gates on the dual-rail outputs, with each port enabled by a single-rail control signal as shown in Figure 5b. A write operation is triggered by data arrival, while a read is triggered by assertion of the associated read line with a port. This provides a selective read/write control capability for the register.

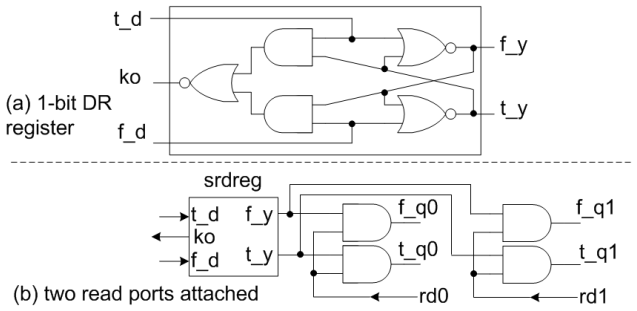

Figure 5. Dual-rail register based on an SR-latch.

Balsa uses handshaking modules known as S-elements and T-elements [14] to implement the separate control channel for control-driven transfers. These elements have elegant implementations that are small and fast; the signal transition graphs for these two elements are shown in Figure 6. Typical use is to connect a chain of these elements to form a sequencer, with the *la* output of one element connected to the *lr* input of the next element. The *Or* output is typically used to trigger a read on one or more registers, with the *Oa* input connected to the output of the ack network for the destination registers. The T-element offers more concurrency than the S-element, as it asserts *la+* (starts next sequencer element) when *Oa+* occurs, thus beginning the next datapath action while the current datapath action is returning to NULL. Balsa uses clever configurations of these elements with additional gating to accomplish various control structures such as *loop-while*, *if-else*, etc.

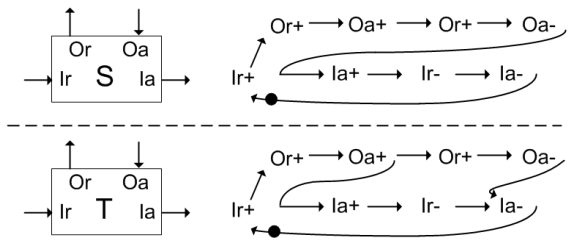

Figure 6. S-element, T-element STGs.

IV. The Uncle Synthesis Flow

The Uncle toolset allows for the RTL specification of asynchronous dual-rail systems using either the data-driven or control-driven styles as defined in the previous section. RTL specification means that the designer is responsible for both register and control implementation (except for ack networks). The motivation for Uncle is simple as there is not a readily available RTL-based synthesis toolset for dual-rail asynchronous systems. Balsa and its successor, Teak (to be discussed later) are the most well-known readily-available synthesis tools for complete dual-rail logic asynchronous systems, but both are higher level synthesis systems in that they synthesize all of the control logic and dictate the type of implementation style (control-driven for Balsa, data-driven for Teak) of the resulting design. Uncle is a lower-level synthesis tool that gives the user more flexibility (along with more responsibility) for design creation. Uncle provides assistance to the user in terms of combinational logic synthesis via a commercial toolset, automated single-rail to dual-rail conversion for combinational blocks, ack generation, and some automated performance and area optimizations, so it is a non-trivial improvement from manual net listing. Verilog is the chosen input language for Uncle so that commercial synthesis tools can be used for initial logic generation. The RTL specification must be transformed to a gate level implementation by the Uncle flow before simulation, where Balsa's custom input language can be simulated before gate-netlist generation.

A. From RTL to Gates

Figure 7 shows the Uncle synthesis flow controlled by python scripts that invoke the various tools in the flow. The input RTL is transformed to a gate level netlist using commercial synthesis tools (both Synopsys Design Compiler and Cadence RTL Encounter are supported). The input Verilog RTL file contains a mixture of behavioral and gate-level statements that describes a mixture of combinational and control logic. The gate-level statements are necessary for instantiating elements that support the asynchronous paradigm and which cannot be inferred from behavioral RTL statements. Parameterized modules are available from an Uncle-provided library and are used to reduce the code footprint of these constructs, reducing the RTL coding burden on the designer.

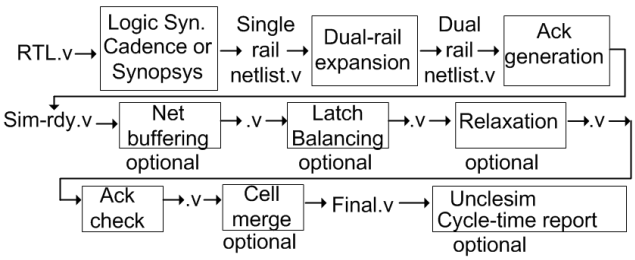

Figure 7. Uncle synthesis flow.

The target library read by the commercial synthesis tool contains *and2*, *xor2*, *or2*, *inverter*, D-flip-flop (DFF), D-latch (DLAT), and other gates that are either black boxes for special use (such as T-,S- elements), or are common complex gates

that have been mapped to an optimized NCL implementation (e.g., a full adder). These gates have unit delays for timing, and area figures that are relatively proportional to their transistor counts. The single-rail netlist is then expanded to a dual-rail netlist with gates and registers expanded to their actual dual-rail implementations. The ack network is then generated, at which point the gate level netlist is simulation-ready. The steps after this point are optional optimizations and checking. The ack checker is a tool that reverse engineers the ack network to mechanically check its correctness. This is primarily included as a check for coding errors in the ack generation tool when new approaches in ack network generation are tested.

B. Ack Generation

For the final asynchronous netlist to be live and safe, at a minimum each latch in a data-driven netlist must receive an acknowledgement from each destination latch of its output (or each control element, in a control-driven netlist that gates data, must receive an ack from each destination latch). One approach is to generate an individual C-gate network for each ack (with sharing of common C-gates between the networks if possible). This promotes bit-level pipelining at the probable cost of a larger acknowledgement network. A merged approach for ack generation merges acknowledgements for a group of latches to produce an ack that is used for all latches in the group. This generally reduces the size of the acknowledgement network at the cost of more synchronization in the design. Uncle supports both approaches, but all examples in this paper used a merged-ack approach. The merging algorithm used is simple; any latches that have at least one common destination have their ack networks merged. Common C-gate sub-networks between ack networks are extracted for additional transistor savings. C-element based demultiplexers that steer data to 1-of-N destinations complicate ack generation in that the destination acks must be OR'ed together since only one destination receives data (in the case of Uncle's low-true acks, AND gates are used for the low-true OR operation). Uncle detects demultiplexers in the netlists via special properties on the black box gates that define demultiplexer gates, and generates the appropriate ack gating.

C. Net Buffering

The net buffering step reads an external timing data file for the target library, where the timing data is non-linear delay model (NLDM) lookup tables for output transition time and propagation delay based on input transition time and output capacitive load (the timing data file also contains pin capacitance information). The target cell library for this timing data and used with the examples in this paper are pre-layout transistor-level spice sub-circuits with transistor models from a commercial 65nm process. The current net buffering implementation is a simple approach that buffers nets to meet a user-specified transition time (two different times can be specified; one for the global reset net and a default one for all other signal nets). The signal net target transition time used for all examples in this paper is approximately equivalent to a 1X inverter driving four separate 4X inverter loads. During net buffering, if a transition time failure is found, then the algorithm first tries to replace it with the smallest gate size (if multiple gate sizes are available for the problem gate) that meets the transition time target. If gate variants are not available, then a buffer tree using inverters is built. The target library has four drive-strength variants of inverters, three variants of AND2, two variants of reset-to-NULL data registers, and two variants of the most commonly used NCL gates. The net buffering is unsophisticated in that it does not buffer for performance by tracing critical loops. However, for many designs this approach does improve performance, but slowdowns were noted for a few designs in the Uncle regression suite. Simulations indicate that the NLDM delay engine in Uncle produces results that are within 5% of the transistor level simulations using the pre-layout transistor models. For silicon fabrication purposes, the NLDM characterization should be for library cells with parasitics extracted from cell geometry for more accurate timing.

D. Latch Balancing

Latch balancing is a performance optimization for the data-driven style that moves half-latches in the netlist to balance data delays with ack delays (for the remainder of this section, half-latches are simply referred to as latches). In a linear multi-stage pipeline, the stage with the longest delay loop formed by the forward data path and the backward ack path sets the pipeline's maximum throughput. In the finite state machine arrangement of Figure 4, the longest loop delay is formed by delay through the combination logic plus the backwards delay path of the ack network [8]. Figure 8a shows a data-driven FSM with an unbalanced delay where the data delay is approximately 2X that of the ack delay (the length of the delay boxes indicate relative delay). The ack delay is dependent on the number of destination points that sets the completion network depth, while the data delay depends on the data logic complexity. Figure 8b shows the design after delay balancing in which logic has been pushed through the L3 latches to reside between the L3 and L2 latches. Observe that the maximum loop delay has now decreased.

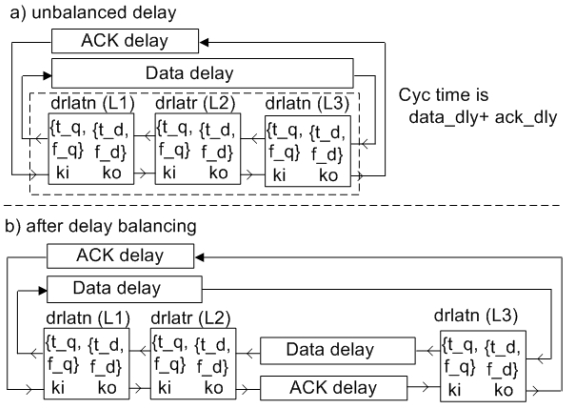

Figure 8. Latch Balancing.

Additional speed up could possibly be obtained by also pushing logic between latches L2 and L1, but the initial data on the L2 latches would then mean that the NCL logic between L2 and L1 would be in an unknown state. This could be solved by forcing the outputs of L2 low (but keep the internal state high) or by adding resets to the affected NCL logic. Latch balancing

generally results in more transistors as the datapath width increases moving towards the source registers requiring more latches, with a corresponding increase in the ack network size.

The latch balancing algorithm as currently implemented in Uncle supports latch pushing as shown in Figure 8b as well as latch pushing in linear pipelines. The latch balancing implementation is an iterative heuristic algorithm that searches for latches to push that satisfy the criteria in Figure 9a (LATj is pushed towards LATi). Several sorting/pruning stages based on data/ack/cycle delays are used to find latch candidates that are most likely to improve performance if pushed. During each iteration, latches are pushed one gate level, and affected ack networks are rebuilt. Latches that only feed primary outputs are ineligible for pushing to avoid additive delays when blocks are combined. An iterative algorithm is used instead of trying to push latches by multiple gate levels in one step because of the difficulty in predicting delays in the regenerated ack networks. This current implementation works appropriately for FSMs as defined by Figure 4 but has a problem with linear pipelines in that latches are pushed in one direction only. The algorithm will fail to find any latches to balance if the linear pipeline has a longest cycle delay as shown in Figure 9b as the destination of the longest cycle terminates on latches that source primary outputs (in this case, LATj would need to pushed towards LATk). The current implementation also does not automatically insert latch stages for performance improvement; it works only with existing latch stages. These shortcomings will be addressed in future tool revisions.

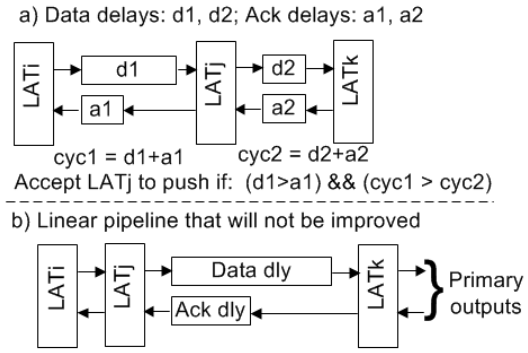

Figure 9. Latch criteria.

E. Relaxation, Cell Merging, Unclesim

Relaxation [9][10] is a technique that looks for redundant paths from a primary input to a primary output, and finds gates that do not have to be fully expanded to dual-rail versions, but can be implemented by eager versions that require fewer transistors. Uncle implements an area-driven relaxation algorithm, but this was not used in this paper as it is being retooled to a timing driven algorithm since a delay calculator has recently been installed in Uncle. A cell merging step is performed in which adjacent gates with no fanout are merged into more complex gates. This cell merger is a less-sophisticated version of the technology mapper/merging implemented in the ATN toolset [11] and is area-driven only. The toolset includes a simulator (Unclesim) that uses the internal delay calculator based on NLDM timing. The simulator can apply either random or user-specified stimuli and reports average cycle time, average switched capacitance per cycle and gate orphans.

V. Design Examples and Comparisons

This section presents Greatest Common Denominator (GCD) and Viterbi decoder design examples and compares them to Balsa-generated designs, as that allows direct complete-system comparisons (Teak [13][16], a successor to Balsa, will be discussed at the end of this section). There have been other toolsets that have addressed parts of the NCL asynchronous dual-rail synthesis problem, such as ATN [11] (combinational logic only, dual-rail expansion, timing-driven relaxation, technology mapping) and the toolset used by Theseus Logic [15] in 1995-2005 (roughly) timeframe (automated combinational logic only, ack networks manually created, synthesis using Synopsys); also had cell merging, and a simulator with orphan detection. However, these toolsets either do not generate complete systems for comparison purposes or are currently unavailable.

A. Example RTL Statements

Page length restrictions preclude inclusion of full code examples (these are available with regression tests in the Uncle distribution). Instead, a few code excerpts are presented to illustrate logic generation. Figure 10a shows how to infer a data-driven half-latch from RTL, while Figure 10b gives the inference of a data-driven register with initial data (a DFF is generated, which is then mapped to the three half-latch structure). Data-driven RTL requires a clock signal in order for the commercial synthesis tool to infer latches/DFFs, which is removed during the mapping process. Control in data-driven designs is simply combinational logic, so standard Verilog RTL constructs were used for this. For arithmetic blocks, RTL operators such as '+', '<' etc. can be used, but these designs used parameterized modules to ensure that known architectures are used instead of an architecture selected by the commercial synthesis tool. Ripple-carry architectures were used for all addition/subtraction operations (both Balsa and Uncle use the same NCL full-adder implementation).

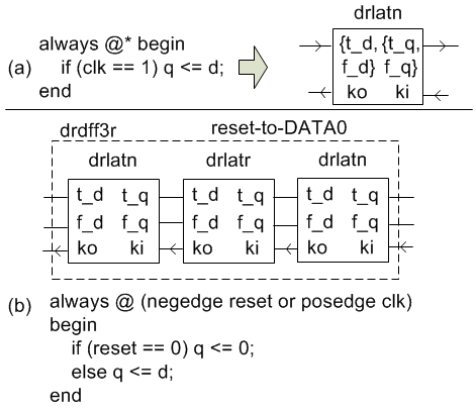

Figure 10. Data-driven latch inference from RTL.

Control-driven registers are instantiated via the use of parameterized modules as shown in Figure 11a, which instantiates a 16-bit register with two read ports (code taken from Uncle GCD example). Control-elements such as T/S elements are instantiated directly by the designer, with the ackin port exposed by the Uncle-defined cells. Generally, this ack network comes from destination registers and is generated by the Uncle mapping process, in which case the designer simply ties it to logic 0 to suppress Verilog lint messages. However, the ackin ports sometimes need to be manually connected in the netlist in the case of some control blocks like while-loops, which is the reason that it is exposed in the Uncle cell definition. Verilog modules that implement while-loop, choice structures, etc. are available to reduce RTL coding burden on the designer.

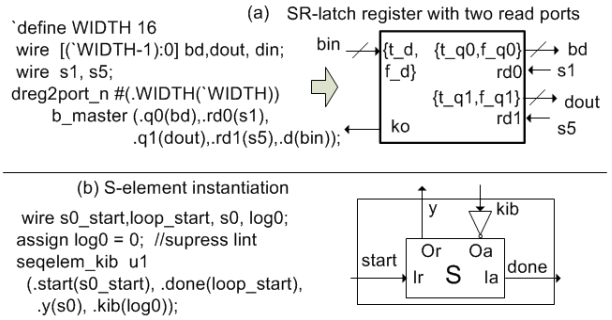

Figure 11. Control-driven registers and elements in RTL.

B. Simulation Comparisons

The data from transistor level simulations in this section was generated by Cadence Ultrasim, with gates that used pre-layout transistor-level spice netlists. The transistor models were from a commercial 65nm process. All designs were first verified at the gate level using Verilog testbenches, which were then converted to Verilog VAMS testbenches for the transistor level simulations. All testbenches were self-checking using an output vector golden file. All Balsa code was taken from published sources. The published Balsa code sources used were all performance-optimized versions. The Balsa GCD code is from [12] (datapath width changed from 8-bits to 16-bits), while the Viterbi Balsa code is from [16]. Balsa netlists were generated using dual-rail/NCL and SR-register options. The hierarchical Balsa gate-level netlists were flattened using Cadence RTL Encounter, and transformed to the gate library used by Uncle. The hierarchical Balsa gate-level netlists were edited to ensure that complex Boolean gates and NCL gates available in the Uncle library were used by the Balsa netlist, to ensure apples-to-apples comparison. A caveat is that a few small functional blocks in the Balsa-generated Viterbi units that used a special purpose NCL cell named DRFAPNCL from the Balsa library caused incorrect operation. This cell was replaced with the DRFAP cell, generated by using the dual-rail/balanced option.

Table I compares metrics for several different Uncle versions of a 16-bit GCD design that used successive subtraction. The control-driven Uncle versions are superior because this design benefits from selective read/write register control. While the data-driven/latch-balanced performance is close to the control-driven performance, both the transistor count and energy usage are factors larger. Extra logic had to be placed around the data-driven GCD in order to ensure that its input ports were only active when they required data, and the output port active only when the result was ready. Latch balancing did significantly improve the performance of the data-driven design.

TABLE I. GCD16 UNCLE VERSIONS

Uncle ver.	DD	DD/ NB	DD/LB /NB	CD	CD/NB
transistors	16192	16226	20128	8658	8662
*	1.87	1.87	2.32	1.00	1.00
cyc. time (ns)	105.7	86.0	64.9	75.7	62.4
*	1.69	1.38	1.04	1.21	1.00
energy (pJ)	32.4	35.3	49.7	10.2	10.8
*	3.17	3.44	4.85	1.00	1.05

Key: DD=data driven, CD=ctrl-driven, LB=latch-balanced, NB= net-buffered, *: ratio to best

Table II compares the fastest Uncle design (control-driven, net-buffered) against the Balsa design, showing that Uncle is better than Balsa in all categories. The Balsa netlist used separate read ports for each output register destination, which increased the number of transistors, but also distributed the fanout. The RTL written for the Uncle version used the same read port for multiple destinations that were active in the same state, with the automated net buffering handling the increased fanout.

TABLE II. GCD16 UNCLE VS. BALSA

	transistors		Cyc time (ns)		Energy (pJ)	
	Balsa	Uncle (CD/ NB)	Balsa	Uncle (CD/ NB)	Balsa	Uncle (CD/ NB)
	11455	8662	85.2	62.4	13.7	10.8
*	1.32	1.00	1.37	1.00	1.27	1.00

Tables III through VII give data from the Viterbi decoder design. The Viterbi decoder has three parts (Branch Metric Unit: BMU, Path Metric Unit: PMU, History Unit: HU) with each part presenting a different design problem. Balsa procedures from these units made liberal use of concurrent loops with active eager enclosures, which achieves pipelining without pipeline registers [16] (registers in this context means SR-latch registers, and not half-latches, which are liberally used in the Balsa-generated logic). Table III compares Uncle and Balsa versions of the BMU, which is just combinational logic. A half-latch was placed on the BMU outputs for the Uncle version as a means of generating the ack network. The

cycle times of the Uncle/Balsa versions are similar, with Uncle holding a slight edge. However, the Balsa transistor count and energy usage are much higher than the Uncle versions. The high transistor count in the Balsa implementation seems to originate from fine-grain functional blocks with control wrappers, while the Uncle implementation resulted in a block of combinational logic with a half-latch on its output.

TABLE III. BMU UNCLE VS. BALSA

	transistors		Cycle time (ns)		Energy (pJ)	
	Balsa	Uncle (DD/ NB)	Balsa	Uncle (DD/ NB)	Balsa	Uncle (DD/ NB)
	9040	5338	9.30	8.87	2.33	1.35
*	1.69	1.00	1.05	1.00	1.73	1.00

The PMU is an interesting design in that it is a set of parallel accumulator-like registers resulting in many parallel three half-latch loops. All registers and all ports are active every compute cycle, which makes it optimal for a data-driven design. Table IV shows the different Uncle implementations. The performance of the initial latch-balanced design was disappointing; examination of the algorithm's performance showed that the output terminal placement in relation to the latches being pushed was constraining latch movement. Two extra half-latch stages were added to the RTL; one stage to decouple the output from the latch loops, and one stage inserted into one of the larger combinational blocks present in the three half-latch loop. This provided more freedom for latch movement during automated latch balancing and resulted in a significant performance improvement.

TABLE IV. PMU UNCLE VERSIONS

Uncle ver.	DD/ NB	DD/NB/ LB	DD/NB/ LB+	CD/NB
transistors	20184	21778	24561	18838
*	1.07	1.16	1.30	1.00
cyc. time (ns)	13.4	13.4	6.9	13.3
*	1.93	1.93	1.00	1.91
energy (pJ)	5.1	5.7	6.8	4.6
*	1.12	1.24	1.48	1.00

Key: LB+ = latch-balanced, two set of half-latches added to RTL (one in FSM loop, and one on output port).

Table V compares the fastest Uncle PMU version (which was data-driven style) against the Balsa version, showing that the Uncle version is better in all categories. It should be noted that the Uncle control-driven version was slower than the Balsa version, as the control wrappers, with PassivatorPush components (half-latches) around the functional blocks in the Balsa netlist, resulted in more overlap of data/ack delays than in the Uncle control-driven netlist.

TABLE V. PMU UNCLE VS. BALSA

	transistors		Cycle time (ns)		Energy (pJ)	
	Balsa	Uncle (DD/ NB/ LB+)	Balsa	Uncle (DD/ NB/ LB+)	Balsa	Uncle (DD/ NB/ LB+)
	38328	24561	9.39	6.94	9.73	6.81
*	1.56	1.00	1.35	1.00	1.43	1.00

The History unit is considerably more complex than the BMU/PMU from a control standpoint, in that it has three 16-entry register files (4-bit, 2-bit, and 1-bit). An outer loop writes the registers, and can conditionally trigger an inner while loop that contains register read/write operations and executes a variable number of iterations. Only a control-driven Uncle version was designed, as it is clear that a data-driven style will be greatly inferior in terms of transistor count and energy usage. Table VI compares Balsa and Uncle implementations for two different vector sets. The cycle/energy numbers for vector set V1 represents an average of 16 vectors that caused the internal while loop to be skipped, with only register file contents being updated, resulting in a fast cycle time. Vector set V2 caused the internal while loop to be executed a maximum number of times, which looped reading the register file and conditionally updated it, resulting in a long cycle time. The Uncle design uses fewer transistors, and is better than the Balsa versions for energy usage on both vector sets.

TABLE VI. HU UNCLE VS. BALSA

		Balsa	Uncle CD/NB	Uncle CD
	transistors	21819	16471	16425
	*	1.33	1.00	1.00
v1	cyc. time (ns)	10.8	6.8	8.4
	*	1.60	1.00	1.25
	energy (pJ)	1.34	1.17	1.07
	*	1.26	1.09	1.00
v2	cyc. time (ns)	230.7	161.3	192.0
	*	1.43	1.00	1.19
	energy (pJ)	25.4	19.6	18.7
	*	1.36	1.05	1.00

Key: v1= average for first 16 vectors that initializes register files; v2 = vector that triggers internal while loop

For vector set V1, the improved Uncle performance is from a manually-created control optimization that paralleled the register-file write return-to-NULL with the other operations in the top-level loop only if the inner conditional while loop was skipped (the inner loop also contains a register file write and thus the outer loop register file write must return-to-NULL before this is executed). For vector set V2, the Uncle design

has better performance on the inner while loop execution, and performance on both vector sets benefited from net buffering due to loading on the register file address lines.

Table VII compares the completed Balsa and Uncle Viterbi decoders. The Uncle decoder uses the DD/NB/LB+ PMU RTL. The complete hierarchical Verilog RTL was run through the synthesis/mapping process with net buffering and latch balancing enabled to produce the gate level netlists. The Balsa gate level design was also produced by running the complete hierarchical Balsa code through the Balsa compilation/netlist generation process. The metrics were produced from an average of several test vectors. From Table VII, it is clear that the Uncle design compares well against the Balsa version. It is conjectured that this particular Balsa design was split into too many blocks with high performance control wrappers, resulting in excessive transistor count and energy usage, and that the same performance could have been achieved with fewer high performance wrappers.

TABLE VII. VITERBI UNCLE VS. BALSA

	transistors		Cycle time (ns)		Energy (pJ)	
	Balsa	Uncle	Balsa	Uncle	Balsa	Uncle
	71370	46752	22.0	17.3	15.0	10.5
*	1.53	1.00	1.27	1.00	1.43	1.00

The total transistor count for the Balsa Viterbi decoder is 71370, as compared to the published figure of 68595 from [16] which is an ~4% difference. These counts are close enough that we feel that we have done a good faith effort at recreating the design published in [16].

Teak is a successor toolset to Balsa that uses a data-driven style (data-driven registers are not the same as used in Uncle). One of Teak's goals is to automatically insert latch stages and balance delays for optimum throughput. A Teak-generated Viterbi decoder from [16] had ~52% more transistors, ~35% more energy, and was ~8% faster than the Balsa version. We did not do a Teak versus Uncle comparison because it appears from these numbers that Uncle would compare well. However, Teak is a fairly new tool with only one public release, and internal Teak versions have probably evolved significantly since those numbers were generated. As such, Uncle versus Teak comparisons are reserved as future work. Another data-driven toolset is described in [17], but it is unclear as to the availability of this toolset.

SUMMARY AND FUTURE WORK

Uncle is a toolset that provides non-trivial automated assistance to designers for RTL-based NCL asynchronous design. RTL-based design places more responsibility on the designer, but a knowledgeable designer may be able to produce higher quality designs than with a higher level synthesis tool. While the data presented in this paper shows that the current Uncle toolset generates designs that compare favorably to Balsa-generated designs, many improvements to the flow can be made such as better NCL logic generation (timing-driven relaxation, peephole logic optimization), automated insertion of half-stages for optimum performance, improved latch balancing, improved net buffering for performance, and timing-driven ack network generation.

ACKNOWLEDGMENT

We would like to thank Mentor Graphics, Synopsys and Cadence for use of their tools as per their respective university programs.

REFERENCES

[1] Wave Semiconductor. Available: *www.wavesemi.com*.

[2] G. Ansel, et.al, "Ultra Low Power Sub-VT Digital Circuits with Autonomous Block-level Supply Control," to appear at GOMACTech 12, Las Vegas, NV, 2012.

[3] A. Bardsley, "Balsa: An asynchronous cicuit synthesis system," M.Phil thesis, Sch. Computer Sci., Univ. Manchester, U. K., Manchester, 1998.

[4] D. Edwards, A. Bardsley, L. Jani, L. Plana, W. Toms, "Balsa: A Tutorial Guide," The University of Manchester, Manchester, U.K. 2006.

[5] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," in *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proc., Sixth Intrntl. Symp. on*, 2000, pp. 114-125.

[6] S. Smith and J. Di, *Designing Asynchronous Circuits using NULL Convention Logic (NCL)*: Morgan & Claypool, 86 pp, 2009.

[7] K. M. Fant and S. A. Brandt, "NULL Convention Logic: a complete and consistent logic for asynchronous digital circuit synthesis," in *Appl. Spec. Sys.,Proc. of Intrntl. Conference on*, 1996, pp. 261-273.

[8] T. E. Williams, "Analyzing and improving the latency and throughput performance of self-timed pipelines and rings," in *Circuits and Systems, 1992. ISCAS '92. Proc., 1992 Intrntl Sym. on*, 1992, pp. 665-668 vol.2.

[9] J. Cheoljoo and S. M. Nowick, "Optimization of Robust Asynchronous Circuits by Local Input Completeness Relaxation," in *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, 2007, pp. 622-627.

[10] J. Cheoljoo and S. M. Nowick, "Block-Level Relaxation for Timing-Robust Asynchronous Circuits Based on Eager Evaluation," in *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, 2008, pp. 95-104.

[11] J. Cheoljoo and S. M. Nowick, "Technology Mapping and Cell Merger for Asynchronous Threshold Networks," *Computer-Aided Design of Integ. Circuits and Systems, IEEE Trans. on,* vol. 27, pp. 659-672, 2008.

[12] L. A. Tarazona, D. A. Edwards, A. Bardsley, and L. A. Plana, "Description-level Optimisation of Synthesisable Asynchronous Circuits," in *13th Euromicro Conf. on Dig. Sys. Dsg.*, 2010, pp. 441-448.

[13] A. Bardsley, L. Tarazona, and D. Edwards, "Teak: A Token-Flow Implementation for the Balsa Language," in *Application of Concurrency to System Design, 2009. ACSD '09. Ninth Intl. Conf. on*, 2009, pp. 23-31.

[14] L. A. Plana, S. Taylor, and D. Edwards, "Attacking control overhead to improve synthesised asynchronous circuit performance," in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE Intrntl Conf. on*, 2005, pp. 703-710.

[15] A. Kondratyev and K. Lwin, "Design of asynchronous circuits using synchronous CAD tools," *Design & Test of Computers, IEEE,* vol. 19, pp. 107-117, 2002.

[16] L. T. Duarte, "Performance-oriented Syntax-directed Synthesis of Asynchronous Circuits," PhD, Sch. Computer Sci., Univ. Manchester, U. K., Manchester, 2010.

[17] S. Taylor, D. A. Edwards, L. A. Plana, and L. A. Tarazona, "Asynchronous Data-Driven Circuit Synthesis," *VLSI Systems, IEEE Trans. on,* vol. 18, pp. 1093-1106, 2010.

[18] J. Sparsø, J. Staunstrup, and M. Dantzer-Sorensen, "Design of delay insensitive circuits using multi-ring structures," in *Design. Automation Conference, 1992. EURO-VHDL '92, EURO-DAC '92. European*, 1992, pp. 15-20.