

Proteus: An ASIC Flow for GHz Asynchronous Designs

Peter A. Beerel

University of Southern California
and Fulcrum Microsystems

Georgios D. Dimou and Andrew M. Lines

Fulcrum Microsystems

Editors' note:

The high-performance benefits of asynchronous design have hitherto been obtained only using full-custom design. This article presents an industrial-strength asynchronous ASIC CAD flow that enables the automatic synthesis and physical design of high-level specifications into GHz silicon, greatly reducing design time and enabling far wider use of asynchronous technology.

—Montek Singh (UNC Chapel Hill) and
Luciano Lavagno (Politecnico di Torino)

Another example, a flow by Cortadella et al.,² starts with RTL and targets low-power single-rail micropipeline templates,⁴ which achieve similar performance to the comparable synchronous design. The effort most like ours is Boston University's Weaver project,⁵ which targets high-performance pipelines using quasi-delay-insensitive templates. Compared with the Weaver

■ **DESPITE THE NUMEROUS** asynchronous designs that have demonstrated substantial benefits over their synchronous counterparts, widespread adoption of asynchronous logic has been significantly hampered by the lack of CAD tools and the challenges of managing asynchronous-synchronous interfaces. This article describes Proteus, an asynchronous ASIC CAD flow, developed by TimeLess Design Automation. The flow's objective is to achieve two to three times higher performance, namely 1.1 GHz in a TSMC 65-nm process technology, than typically possible for synchronous counterparts. Proteus is based on a proprietary cell library of domino logic and asynchronous control cells—small collections of gates sized, laid out, and characterized as single library cells, designed to implement robust high-performance pipelined circuits.¹ The Proteus flow leverages both synchronous synthesis and place-and-route tools and, as a starting point, supports legacy register transfer language (RTL) designs as well as Fulcrum Microsystems' proprietary higher-level language based on Communicating Sequential Processes (CSP).

Similar automated design flows, developed by other researchers,² have targeted a range of performance and start from a variety of high-level languages. One flow, developed by Handshake Solutions, starts with Haste, a language similar to CSP, but which generally targets low-power, lower-performance designs.³

flow, the novel aspects of our flow include the ability to start with CSP, which is more expressive than RTL; a novel clustering algorithm that reduces area, latency, and place-and-route complexity; and a performance-driven place-and-route back end. The sidebar, "Bridging the Gap between Asynchronous Research and Industry Application," offers more information.

In this article, we describe the potential benefits of achieving high-performance asynchronous designs from high-level specifications. As we explain, a key advantage of asynchronous designs stems from breaking the traditional alignment of flip-flops in synchronous implementations. The result is typically shorter critical paths in asynchronous implementations from the same RTL. This critical path is often referred to as the *algorithmic cycle*, and its relative shortness is the origin of the high-performance capabilities of the resulting circuits.

In an overview of the Proteus flow, we describe a novel syntax-directed approach for translating high-level CSP programs into synthesizable RTL and discuss how we adapt commercially standard synthesis tools to create a synchronous-image netlist. The core of the flow is a new tool called ClockFree, which translates the image netlist into its asynchronous target netlist, optimizing for both latency and throughput constraints, and generates timing constraints on the target netlist that guarantee performance. Finally, we

describe how we couple Clock-Free with commercially standard place-and-route tools to produce a final layout that meets our gigahertz target.

Throughout, we highlight how we support the use of macros: blocks abstracted and characterized for use in a semicustom environment. In particular, we discuss the support of full-custom macros within a Proteus block and how the final layout produced by Proteus can be used hierarchically either within a full-custom block or as a macro within a larger Proteus block. This hierarchy support enables the Proteus flow to scale and ultimately be used throughout the high-performance data path of Fulcrum Microsystems' next-generation Ethernet switch chip.

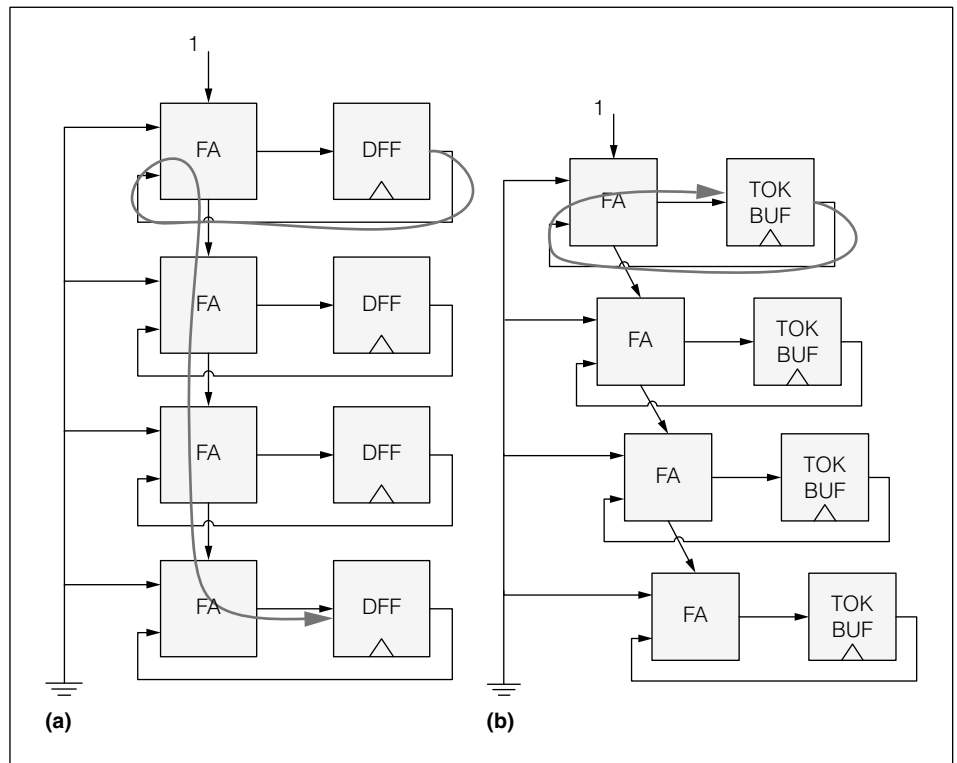


Figure 1. Advantages of breaking the traditional alignment of flip-flops: original synchronous design (a) and optimized asynchronous design (b).

Breaking flip-flop alignment

A synchronous circuit's throughput is dictated by the global clock's fastest achievable cycle time, which is in turn dictated by the longest path from flip-flop to flip-flop. In traditional synchronous flows, clock skew is minimized, and the clocking of all flip-flops is as simultaneous as possible. Consider the four-bit incrementer shown in Figure 1a. The critical path arises from the carry chain and starts at the least-significant-bit flip-flop to the most-significant-bit flip-flop.

In more advanced synchronous flows, however, relaxing the alignment constraint is allowed with a technique called *useful skew*. In the four-bit incrementer example, allowing the flip-flops associated with the more significant bits to be relatively skewed in time provides more time for the carry chain and enables a faster cycle time. Unfortunately, this technique is often recommended only for the last stages of physical design and allows only a small amount of useful skew; otherwise, the complications in clock gating and clock tree synthesis become formidable.

However, unlike synchronous circuits, asynchronous implementations can easily support arbitrary flip-flop alignment by using a local enable signal in

an asynchronous pipeline stage to control only the flip-flops we want aligned. Different pipeline stages can then be naturally staggered in time, based on the bit-level data flow implicit in the logic dependency between pipeline stages. At the extreme, each full adder and each flip-flop can be implemented in its own pipeline stage that communicates asynchronously with its neighbors, as Figure 1b illustrates.

This arbitrary alignment of flip-flops forms a lower bound of the cycle time, τ_a , of an asynchronous implementation, referred to as the *algorithmic cycle time*.⁶ It can be computed using a variety of algorithms for computing the maximum cycle mean of a graph (see, e.g., Dasdan and Gupta⁷). One efficient means of computing τ_a is the following linear program:

min τ such that

$$a_j > a_i + d_{ij} - m_j \tau$$

where a_j is the arrival time for gate j , d_{ij} is the delay from the output of gate i to the output of gate j , and m_j is 1 if gate j is a flip-flop and 0 otherwise.

Consider again our 4-bit accumulator example in which each full adder (FA) and flip-flop is implemented

Bridging the Gap between Asynchronous Research and Industry Application

Asynchronous logic has not been widely adopted for several reasons, chief among them being a lack of CAD tools and difficulties in managing asynchronous-synchronous interfaces. Adoption has been slow despite the evidence from numerous asynchronous designs of yielding substantial benefits over their synchronous counterparts. However, several research groups and associated start-up companies have begun clearing these hurdles in a variety of domains.

Handshake Solutions, originating from research at Eindhoven University,¹ commercialized a fully automated CAD flow with far lower power and energy consumption than is possible with traditional synchronous approaches and began developing a significant low-power niche market. Silistix, originating from research at the University of Manchester, addressed the SoC network market with low-power asynchronous network-on-chip IP blocks with some success.² Tiempo, originating from research from TIMA Laboratory, has developed low-power asynchronous IP blocks for cryptography and other security applications as well as EDA tools (<http://www.tiempo-ic.com>). Achronix Semiconductor, originating from research at Cornell University, has developed ultra-high-performance FPGAs using asynchronous logic (<http://www.achronix.com>). Fulcrum Microsystems, originating from research at

Caltech,³ leverages proprietary high-performance asynchronous circuits and tools to develop high-performance Ethernet switches (<http://www.fulcrummicro.com>).

Several other start-ups, including Theseus Logic—which merged with Camgian Microsystems (<http://www.camgian.com>), Elastix (<http://www.elastixcorp.com>), Nanochronous (<http://www.nanochronous.com>), and TimeLess Design Automation—have addressed the asynchronous ASIC market. The common goal is to provide a fully automated ASIC CAD flow that is as easy to use as traditional synchronous flows but generates superior circuits in terms of some significant metric, such as high performance or low power. Typically, this process involves adapting as many of the available synchronous tools as possible to gain the advantages of reducing

- the capital needed to create these flows,
- the risk associated with using these flows, and
- the barrier of entry to their use by designers trained in synchronous design.

The typical target metric is low power; however, low electromagnetic noise, reliability to process variability, and high performance are factors that are important in some markets.

The Proteus flow has its roots in a joint Columbia University, University of Southern California, and National

in its own pipeline stage that has unit delay. The linear program would include up to two constraints per FA, one for each nontrivial input, each indicating that the arrival time of the particular FA must be at least one unit delay greater than that of its driving gate. The constraints also include one constraint per flip-flop of the type $a_j \geq a_k + 1 - \tau$, where a_j is the arrival time of the flip-flop and a_k is the arrival time of the particular FA driving the flip-flop. The result of the minimization would show that the minimum algorithmic cycle time is $\tau_a = 2$. In particular, an optimal set of arrival times is one in which the full adder for bit i has arrival time i and the flip-flop for bit i has arrival time $i + 1$. The result indicates that when the flip-flops are allowed to shift in time with respect to each other, the longest critical loop includes only two gates, as Figure 1b shows.

Proteus design flow

The Proteus flow, as Figure 2 shows, includes translation of CSP-based high-level language into

synthesizable Verilog, synthesis into an image netlist, translation and optimization of the image netlist into an asynchronous implementation, and subsequently the implementation's physical design. The flow's salient feature is that it reuses standard logic synthesis tools as well as physical design tools.

Cell library

Our cell library is based on a proprietary domino-based implementation of the precharged half-buffer (PCHB) template.¹ Figure 3 shows the basic version of this template, which is used to implement synthesized logic. Each pipeline stage consists of one level of dual-rail domino logic for computation; Muller C-elements to gather acknowledgment from various fan-outs; and an integrated control circuit that implements the needed I/O completion, the local enable signal, and the pipeline stage's acknowledgment signal output. These pipeline stages are unconditional in nature because they receive tokens from all input channels and generate a token on their output

Science Foundation Information Technology Research (ITR) grant, which supported our early work on automated placement and routing of high-performance asynchronous designs.⁴ In addition, an SRC research grant supported our work on library characterization and static timing analysis of asynchronous designs.⁵ The results of these efforts paved the way for a joint research project between USC and Fulcrum Microsystems, the goal of which was a proof-of-concept high-performance RTL-to-GDSII flow through place and route using quasi-delay-insensitive (QDI) circuits.

The promising results of the USC-Fulcrum research prompted Fulcrum to offer an initial six-month business contract to integrate this technology into their full-custom design flow. Fulcrum's goal was to prove that GHz performance in 65-nm TSMC process technology was reliably achievable on sufficiently complex real examples. Using this opportunity as a financial basis, Peter A. Beerel and Georgios D. Dimou founded TimeLess Design Automation in spring 2008. Their initial goal was to target high-performance ASICs, with a longer-term goal that included targeting low-power design.

Within six months, the first test chip was completed, which demonstrated working synthesized circuits operating at 1.2 GHz. Within two years, the entire Fulcrum engineering team was using the Proteus flow throughout the high-performance data path of the team's upcoming

next-generation Ethernet switch chip. The tool became an integral part of Fulcrum's design flow, providing reasonable circuits with a fraction of the company's full-custom design effort and enabling design by asynchronous designers with far less experience. This success led to Fulcrum Microsystems' purchasing TimeLess in August 2010. Intel announced plans to acquire Fulcrum in July 2011.

References

1. A.M.G. Peeters, "Single-Rail Handshake Circuits," PhD thesis, Eindhoven Univ. of Technology, 1996.
2. A.M. Scott et al., "Asynchronous On-Chip Communication: Explorations on the Intel PXA27x Processor Peripheral Bus," *Proc. Int'l Symp. Asynchronous Circuits and Systems (ASYNC 07)*, IEEE CS Press, 2007, pp. 60-72.
3. A.J. Martin, M. Nystrom, and C.G. Wong, "Three Generations of Asynchronous Microprocessors," *IEEE Design & Test*, vol. 20, no. 6, 2003, pp. 9-17.
4. R.O. Ozdag and P.A. Beerel, "An Asynchronous Low-Power High-Performance Sequential Decoder Implemented with QDI Templates," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, 2006, pp. 975-985.
5. M. Prakash, "Library Characterization and Static Timing Analysis of Template Based Asynchronous Circuits," master's thesis, Ming Hsieh Dept. of Electrical Eng., Univ. of Southern California, 2007.

channels on every operation cycle. A more detailed description of how this template operates, including timing diagrams, can be found elsewhere.⁶

The logic cells use staticizers to save state and improve the domino logic's noise margins. For example, Figure 4 shows the cell for a dual-rail domino 2-input OR gate. Note that each logic cell also has a 2-input NAND gate tied to the two dynamic logic output nodes to drive 1-bit output signal V , which will go high when the domino logic evaluates. Integrating this NAND gate into the logic cells isolates the design's sensitive dynamic nodes within individual library cells, improving noise margins.

The proprietary library we developed implements all logic functions with 2- and 3-bit inputs as well as many functions of 4, 5, and 6 bits. (Because using dual-rail logic inverters can be implemented for free by twisting wires, we can implement all 2-input logic functions with only two real library cells and implement all 3-input logic functions with 10 real library cells.) The library includes control circuit

cells that support up to four logic gates (a cluster) within a pipeline stage. The library includes up to 4-input C-elements, restricting the maximum fan-out of each cluster to 3 because one input is reserved for the local enable signal.

The library also includes

- dedicated cells that implement pipeline buffers, used to copy their input data to their output for both slack matching and fan-out fixing,
- token buffers used to implement flip-flops,
- specialized scan cells used to implement a proprietary linear asynchronous scan chain, and
- send/receive cells, the only cells in our library with conditional communication semantics.

Please note that *slack* here doesn't refer to a difference in arrival time of a circuit node from that of a budget, as the term is typically used in commercial CAD tools. Rather, *slack matching* refers to a form of pipeline optimization particular to

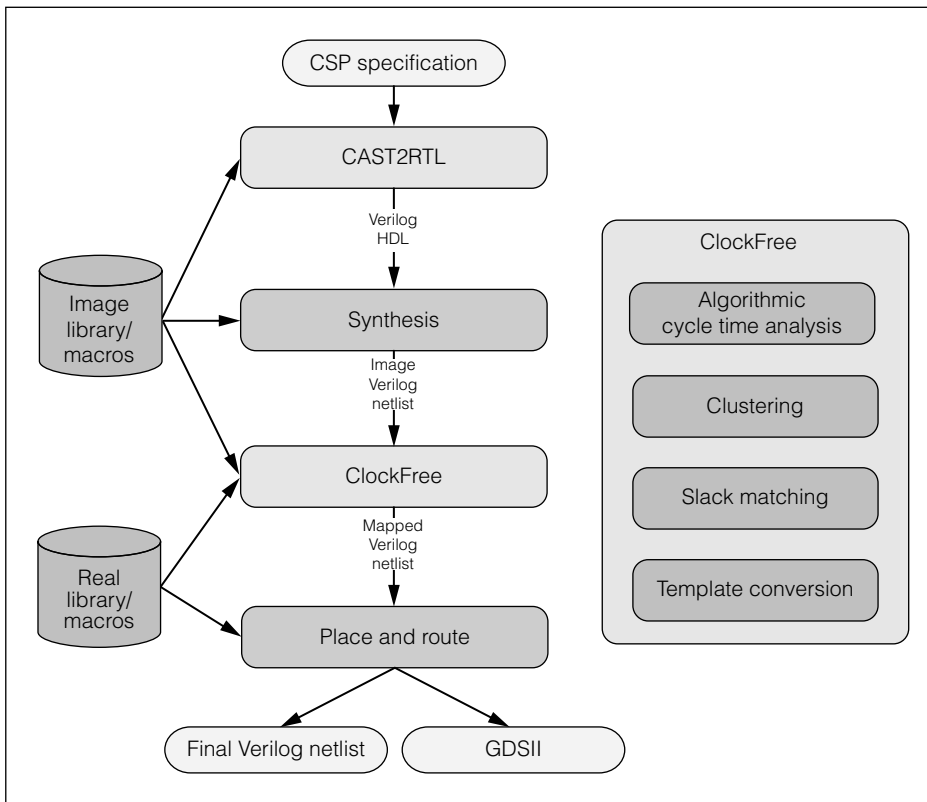


Figure 2. Overview of Proteus design flow.

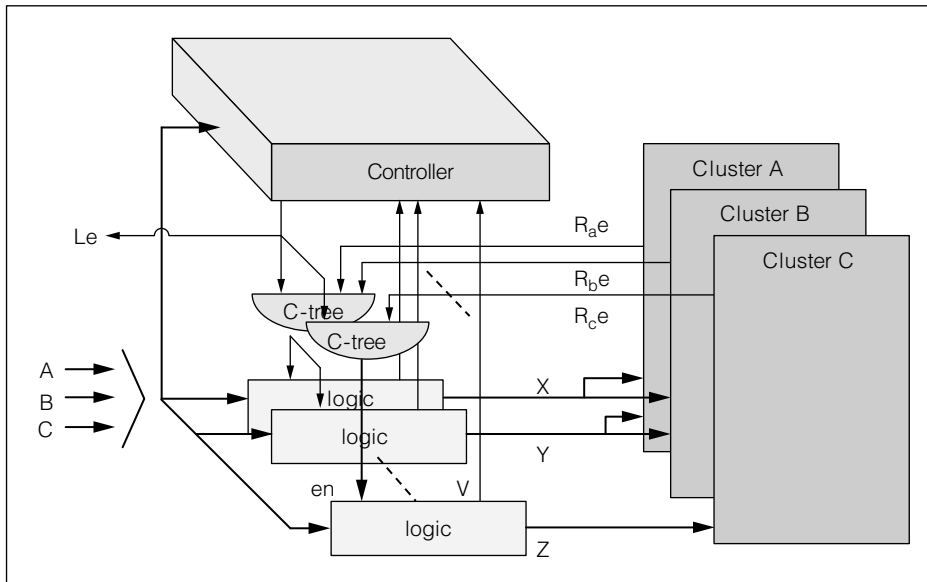


Figure 3. Precharged half-buffer (PCHB) template using proprietary domino-logic-based control cells.

asynchronous designs,⁸ as we explain in the “ClockFree” section.

A send cell conditionally sends tokens on its output according to the value read from an unconditionally

read enable input channel. A receive cell conditionally receives a token on its input according to an unconditionally read enable channel input. These two cells facilitate our implementation of the conditional communication semantics of communicating sequential processes.⁶

Most of the library cells have both real and image forms with four different drives. The image form is a temporary form in the flow and acts as a target for standard synthesis tools that generally don’t understand asynchronous handshaking and can only map to single-rail logic. ClockFree then transforms this image netlist into the real version by converting the image gates into their real forms and adding the necessary asynchronous control cells and C-elements to implement the desired asynchronous handshaking. In both cases, the cells are described in the gold-standard Liberty file format, which details the pins, area, and timing characteristics. Timing characterization challenges for such cells and some solutions have been studied extensively by Prakash.⁹ As needed in synchronous flows, the real cells also have physical descriptions in the standard Library Exchange Format (LEF).

All real cells have asynchronous dual-rail or 1-of-4 rail interfaces.⁶ The image cells have the same logic description as their dual-rail counterparts, but can have a mixture of single-rail and dual-rail interfaces. All logic and token-buffer cells

have only single-rail interfaces (and no handshaking signals), allowing standard synthesis tools to map to them. For example, as Figure 4 shows, the port list of the single-rail image of the 2-input OR gate does


```

module synthesis.examples.csp2rtl.
    accumulate_scan_base;
import lib.serial.scan.ChanDft;
define ABSTRACT_ACCUMULATE16() (e1of2[16] -L,
    +R; ChanDft -LS, +RS) {
    csp { s=0; *[L?x; s=s+x; R!s] }
}

```

Figure 6. Communicating Sequential Process (CSP) description of an accumulator with scan for DFT.

```

module ACCUMULATE16$body (
input [15:0] \value$L ,
output reg \do$L ,
output reg [15:0] \value$R ,
output reg \do$R ,
input \vdd ,
input \GND ,
input \_RESET ,
input CLK);
reg signed \LS,C.d [2:0];
reg signed \RS,C.d [2:0];
bit signed [128:0] s;
bit signed [128:0] x;
bit signed [128:0] temp$0;
bit signed [128:0] ff$s;
bit signed [128:0] ff$x;
always ff @(posedge CLK, negedge _RESET)
    if (!_RESET) begin
        ff$s = '0;
        ff$x = '0;
        ff$s = 1'sd0;
    end
    else begin
        ff$s = s;
        ff$x = x;
    end

always_comb
begin
    s = ff$s ;
    x = ff$x ;
    do$L = '0;
    do$R = '0;
    value$R = '0;

    do$L = '1;
    x = ($signed({1'd0, value$L }
    temp$0 = '0;
    temp$0 = {s + x } ;
    s = temp$0 ;
    do$R = '1;
    value$R = s ;
end

```

Figure 7. Generated RTL body for the accumulator example.

as a postsynthesis image interface between completed blocks (either from previous Proteus runs or a full-custom flow) and newly synthesized logic. For example, full-custom asynchronous SRAMs with complete asynchronous interfaces can be connected to synthesized logic in the image netlist through such image send and receive cells. The advantage is that images of full-custom asynchronous blocks need not be created.

CAST2RTL: A syntax-driven CSP-to-RTL translator

The principal task of the CAST2RTL translator is to map a high-level language called Caltech Asynchronous Synthesis Tools (CAST) into an image netlist. CAST is a hierarchical, object-oriented high-level language in which each cell can be described as a netlist of subblocks, a CSP behavioral description, a Verilog description, or low-level production rules. It also allows each cell to be tagged with directives to control the design process. For Proteus' purposes, the CAST description of each cell is either a stand-alone CSP or a netlist of other blocks, some of which are defined using CSPs to be synthesized while others are previously completed macros.

Figure 6 shows a simple example of an accumulator with 16-bit dual-rail L input and R output channels and proprietary scan input and output channels. The cell is defined with a CSP body and contains largely standard CSP syntax; for more details of this syntax see the literature.⁶ The cell defines a state variable *s* outside the repeat-forever outer loop. Within the loop, the cell reads an input from channel L, adds it to the state variable, and then sends out the resulting output on channel R.

The mapping into Verilog creates a hierarchical Verilog model with instantiated conditional send/receive image cells surrounding a synthesizable RTL body. For example, Figure 7 shows the RTL body for the 16-bit accumulator. Notice that the body's inputs/outputs are single-rail and connect to the single-rail pins of image send/receive cells (latter not shown). Also notice that channels L and R are translated into *valueR* and *valueL* signals to capture the data values along with *doR* and *doL* auxiliary variables. These auxiliary variables are tied to the image enable pins of the send/receive cells.

Each cycle of the CSP repeat-forever outer loop is modeled as one RTL clock cycle. The CSP loop's main functionality is captured in an `always_comb` logic block whose structure closely follows that of

the CSP loop. At the beginning of this block, all *do* signals are initialized to 0. Then, for every location in the CSP loop in which a channel communication occurs, CAST2RTL adds an assignment of 1 to the *do* signal output tied to the associated send/receive cell. For the accumulator, we can see from the RTL code that the result is that *doR* and *doL* signals will always be assigned to a 1. This makes sense, because in this example the channels are unconditionally read or written. Thus we rely on the standard synthesis engine to generate the correct logic for all *do* signals, and the CAST2RTL algorithm can be quite simple.

The use of these auxiliary variables implicitly restricts the class of synthesizable CSP descriptions to those that communicate only a maximum of once per channel per outer CSP loop. The reason is that the values of these auxiliary variables are computed combinatorially in each RTL cycle, and the data value associated with the last communication on a channel would override the desired data value needed for any previous communications on the same channel. Fortunately, this restriction is quite reasonable for the high-performance applications we've targeted, and any CSP description can be rewritten as an explicit state machine that communicates, on any channel, at most once per iteration of the main loop.

The CAST2RTL algorithm conservatively assumes that any CSP variable not initialized at the beginning of the outer CSP repeat-forever block is a state variable. For example, both *s* and *x* in our accumulator are assumed to be state variables. Each state variable is associated with a new flopped version that we add to the RTL body. For example, the RTL body includes `ff$s` as the flopped version of state variable *s*. CAST2RTL generates an `always` block to update these flopped variables on the rising edge of an added clock signal as well as on reset, as usual in synchronous designs. (Although the existence of this reset signal suggests the design can be periodically reset, the translation into an asynchronous circuit only supports reset on power-up.) We rely on the synthesis engine to generate flip-flops only for needed state variables and to optimize away flopped variables that aren't necessary. For example, in the accumulator example, the synthesis engine is sophisticated enough to realize variable `ff$x` is never needed: although it is used to update *x* in the top of the `always_comb` loop, the subsequent assignment of *x* to the data value of the L channel makes this assignment redundant.

Another interesting aspect of the CAST2RTL algorithm is how it assigns bit widths to variables. The basic problem is that in CAST, the bit width of CSP internal variables can be undefined, which isn't possible in RTL. Thus, CAST2RTL assumes a programmable maximum bit width, initially set to 129 bits, which it assigns to any CSP variables of undefined length. For example, as Figure 7 shows, the accumulator variable *s* and related temporary variables in the generated RTL all have the default 129-bit width. We then rely on the synthesis engine to remove any logic and state associated with unneeded most significant bits. In fact, we've found that synthesis engines do this quite well, sometimes at the cost of substantial extra runtime. Of course, we can reduce this overhead by properly annotating CSP bit widths.

Adapting logic synthesis tools

We use a commercially standard synthesis tool to map the CAST2RTL output to a netlist of image gates and macros. Figure 8 shows a portion of the image netlist for the 16-bit accumulator example. The logic gates here have single-rail inputs and outputs. Our naming convention for logic gates is somewhat unusual—essentially a juxtaposition of the number of inputs the logic gate has followed by the hex encoding of its underlying Karnaugh map specification.

The principal advantage of using standard logic synthesis tools is their ability to perform decomposition into logic gates far faster than manual decomposition. In fact, for some complex logic, the result is often better because of the synthesis tools' sophisticated algorithms for sharing subterms.

One of the main challenges of using standard logic synthesis tools, on the other hand, is to properly guide the performance/area trade-off implicit in logic synthesis. As we explained earlier, unlike those in standard synchronous designs, worst-case flop-to-flop paths are not always the critical path for asynchronous circuits, and if they're overconstrained, significant area can be wasted. One common synthesis constraint users specify is to relax all flop-to-flop constraints except for flop self-loops by defining a large default cycle time and explicitly constraining all flop self-loops with a smaller maximum delay. This generally minimizes the algorithmic cycle time without unnecessarily expanding logic area. For example, this constraint would minimize the latency around all flop self-loops in the incrementer example

```

LOGIC3_69_X4 \body.add_40_17.g15833 (.A0 {\value$L[12]}, .A1
    (\body.s[12]), .A2 (\body.add_40_17.n_50), .X {\value$R[12]});
LOGIC3_69_X4 \body.add_40_17.g15834 (.A0 {\value$L[8]}, .A1
    (\body.s[8]), .A2 (\body.add_40_17.n_49), .X {\value$R[8]});
LOGIC2_2_X4 \body.add_40_17.g15835 (.A0 (\body.add_40_17.n_2), .A1
    (\body.add_40_17.n_50), .X (\body.add_40_17.n_55));
LOGIC5_1F11FFFF_X2 \body.add_40_17.g15836 (.A0 (\body.add_40_17.n_32
    ), .A1 {\body.add_40_17.n_47}, .A2 (\body.add_40_17.n_37), .A3
    {\value$L[14]}, .A4 (\body.add_40_17.n_42), .X
    (\body.add_40_17.n_54));
LOGIC5_4F44FFFF_X2 \body.add_40_17.g15837 (.A0 (\body.add_40_17.n_44
    ), .A1 {\value$L[6]}, .A2 (\body.add_40_17.n_44), .A3
    (\body.s[6]), .A4 (\body.add_40_17.n_15), .X
    (\body.add_40_17.n_53));
LOGIC3_69_X4 \body.add_40_17.g15838 (.A0 {\value$L[10]}, .A1
    (\body.s[10]), .A2 (\body.add_40_17.n_47), .X {\value$R[10]});
LOGIC3_96_X4 \body.add_40_17.g15839 (.A0 {\value$L[5]}, .A1
    (\body.s[5]), .A2 (\body.add_40_17.n_45), .X {\value$R[5]});
LOGIC2_2_X4 \body.add_40_17.g15840 (.A0 (\body.add_40_17.n_27), .A1
    (\body.add_40_17.n_46), .X (\body.add_40_17.n_50));
LOGIC3_0B_X4 \body.add_40_17.g15841 (.A0 (\body.add_40_17.n_43), .A1
    (\body.add_40_17.n_22), .A2 (\body.add_40_17.n_104), .X
    (\body.add_40_17.n_49));
LOGIC3_69_X4 \body.add_40_17.g15842 (.A0 {\value$L[6]}, .A1
    (\body.s[6]), .A2 (\body.add_40_17.n_44), .X {\value$R[6]});
LOGIC3_01_X4 \body.add_40_17.g15843 (.A0 (\body.add_40_17.n_102),
    .A1 (\body.add_40_17.n_34), .A2 (\body.add_40_17.n_39), .X
    (\body.add_40_17.n_47));

```

Figure 8. Postsynthesis image netlist.

illustrated in Figure 1 while not overconstraining the logic along the carry chain, which is not along the algorithmic cycle. However, guiding the synthesis to consider not only algorithmic cycle time but also block latency and temporal alignment is also important, and we believe additional automation of this function may be possible.

ClockFree

ClockFree is a performance-driven optimization and translation tool that takes the synthesized netlist and emits a complete asynchronous netlist of real gates that achieves a specified target performance.

Peephole optimization. ClockFree's first step is a series of peephole optimizations in which we optimize the conditional send/receive cells. For example, in the case of the 16-bit accumulator, the send/receive cells will have constant 1s on their enable signals. These cells can thus be transformed into simpler versions that perform only unconditional communication. In addition, back-to-back send/receive pairs controlled by the same enable signal can simply be removed.

Cycle time analysis. After peephole optimization, ClockFree checks for algorithmic cycle time using the linear programming (LP) formulation we've already discussed. Unfortunately, at times the algorithmic cycle time is larger than the target performance, and the tool must then quit with an error message saying that the desired cycle time cannot be met. At this point, the designer must alter the synthesis constraints, alter the CSP input description, or design the offending algorithmic loop in full-custom logic. Tighter synthesis constraints on paths related to the offending algorithmic loop sometimes solve the problem at the cost of additional area. For some cases, however, a solution doesn't exist even in full-custom logic because the amount of logic necessary to process in one cycle is too large. At this point, we must make algorithmic or architectural changes, such as adding more tokens to the loop; examples are presented elsewhere.^{6,10}

Clustering. If the algorithmic cycle time satisfies the desired target, ClockFree enters a clustering algorithm that partitions logic gates into small collections called clusters. The most obvious advantage of clustering is

that it saves area, but we will later show that it also reduces latency and place-and-route complexity. Although in principle each logic gate could be implemented in one pipeline stage with its own dedicated controller, as Smirnov et al. proposed,⁵ the control overhead would be unnecessarily large; the more logic cells in each cluster, the more the pipeline control logic and C-element overhead are minimized.

For example, Figure 9 shows the effect of clustering two 2-input logic gates that share one input. The result is a larger cluster that, when implemented in the PCHB template, has a single control cell efficiently combining the functionality of the two original control circuits by integrating the completion logic for three inputs and two valid signals into one cell. The area savings, however, also extend to the cluster driving the shared input, which now instead of fanning out to two separate clusters, fans out to one. Although the clustering algorithm is generic, it adheres to the limits of the cell library provided. As we described earlier, the proprietary PCHB cell library limits each cluster to one logic cell deep and up to four logic cells wide, with up to six inputs.

While attempting to minimize area, the clustering algorithm also has important performance constraints. First, it must not create clusters that effectively create combinational cycles, resulting in deadlock, as Figure 10 shows. Second, it must not increase the algorithmic cycle time beyond the performance target. For example, consider a slightly different scenario than in Figure 10 wherein stage v3 is a token buffer. Assuming we performed the same clustering as illustrated, the new loop produced wouldn't cause deadlock because it has a token buffer in it. However, the latency around this new loop could be longer than the algorithmic cycle time. Various ways to prevent clustering from causing these adverse scenarios have been described and analyzed by Dimou.¹¹

Fan-out fixing. After clustering, ClockFree must address the underlying fan-out limitation of each PCHB cluster. Specifically, it must add *copy trees* to any cluster that fans out to more than three clusters and

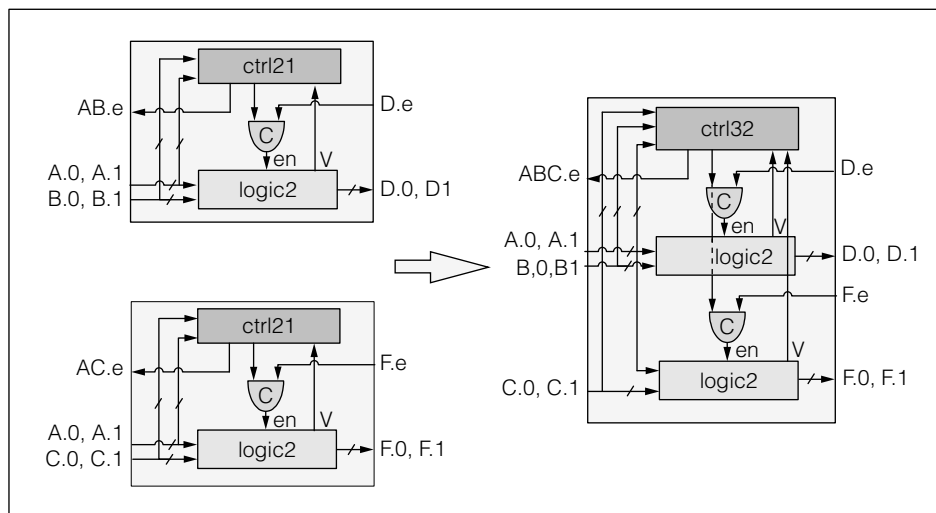


Figure 9. Example of the effect of clustering two 2-input logic gates that share one input.

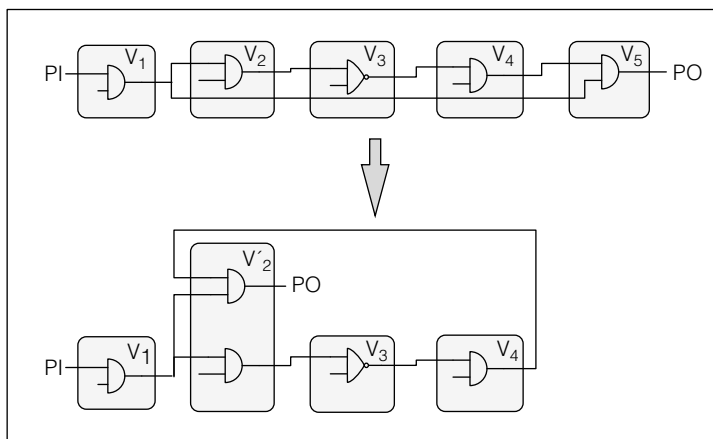


Figure 10. An example of clustering that can lead to deadlock.

effectively pipelines the fan-out effort. A *copy tree node* is a pipeline stage that typically consists of a single pipeline buffer and a 4-input C-element that together allow the data input to be copied to four other fan-outs. The challenge here is to design the shape of these copy trees so as to minimize impact on algorithmic cycle time. To do this, we've designed an algorithm that conservatively identifies channels that are globally critical—that is, on a cycle whose algorithmic delay equals the target cycle time.

We then use a heuristic algorithm to create unbalanced trees of buffers in which globally critical channels are given no extra buffers. After each level of a tree is created, however, the set of channels that might be globally critical can change; consequently, we run an update-globally-critical algorithm.

Table 1. Area improvements of circuit components due to clustering.

Circuit component	Slack buffer area (μm^2)			Fan-out buffer area (μm^2)			Control and C-element area (μm^2)		
	No clustering	Clustering	Diff (%)	No clustering	Clustering	Diff (%)	No clustering	Clustering	Diff (%)
	Hit_Detect_512_Mux4	18,264	14,598	20.1	12,820	11,414	11	32,879	28,007
SISO	4,761	5,259	-10.5	5,946	2,961	50.2	30,823	21,883	29.0
Accumulate16	2,206	1,875	15.0	23	0	100.0	2,608	1,908	26.8
Add_16	2,704	2,389	11.7	269	57	78.7	5,523	3,839	30.5
CRC_NoReg_Output	6,768	6,240	7.8	3,659	2,349	35.8	13,823	11,204	29.0
ECC_Parity	1,427	995	30.2	821	1,319	-60.6	3,082	2,627	14.8
C3540	12,027	12,695	-5.6	6,773	5,351	21.0	15,381	13,095	14.9
S1423	1,924	1,991	-3.4	1,635	975	40.3	8,038	6,475	19.4
S298	1,078	1,063	1.4	365	175	52.1	1,486	1,140	23.3
S400	1,311	1,178	10.1	429	135	68.4	1,950	1,481	24.0
MAC16	20,554	16,148	21.4	8,770	3,506	60.0	41,576	27,855	33.0
Avg			8.9			41.5			22.7

Unfortunately, at times there is no solution that doesn't involve buffering a globally critical channel, and the tool must again quit with an error message that the desired cycle time cannot be met. As in the situation we discussed in which algorithmic cycle time exceeded the target performance, the designer must at this point alter the synthesis constraints, alter the CSP input description, or design the offending algorithmic loop in full-custom logic.

Slack matching. After clustering, we perform performance-driven slack matching, which tries to add the minimum number of pipeline buffers necessary to achieve the desired performance. While there are many proposed slack-matching approaches,¹² we've adopted an LP-based approach based on having one arrival time variable for each cluster and each macro.^{8,11} Macros, as well as the external interface, can also have specified relative arrival times that precisely define the desired relative timing of all associated channels. A benefit of this approach is that it implicitly supports hierarchy by using the final arrival times of the interface to define the relative arrival times of its abstracted macro view. This definition is needed to slack-match higher levels of the design hierarchy. In this way, the slack-matching framework arbitrarily scales to chips with arbitrary complexity.

A convenient feature of this flow is that if the algorithmic cycle time still meets the performance target

after clustering and fan-out fixing, we know that a slack-matching solution exists and we can meet the desired performance. This is because the remaining performance bottlenecks will be due to either unbalanced fork-join pipelines or cycles with too few pipeline stages.⁸ Because of the unit delay model we adopt, we can solve both of these problems by adding an integral number of slack-matching buffers.

Optimized image netlist. The final step in Clock-Free is translating the optimized image netlist. It is only in this step that the specific organization of the asynchronous control and C-elements is needed. In fact, an important point of our flow is that peephole optimization, cycle time analysis, clustering, and fan-out fixing are performed on the netlist's single-rail image. Thus, all the algorithms are largely agnostic to the specific asynchronous template used and rely only on general library characteristics such as forward and backward latencies of a cluster and maximum cluster fan-out.¹¹ This makes the Proteus flow a convenient framework for mapping to other pipeline templates, including various forms of bundled-data micropipelines.⁴

Several of these steps are quite similar to those of Boston University's Weaver project. However, the idea of clustering the circuit netlist to minimize area while preserving performance is new. To appreciate its value, see Tables 1 and 2, in which we compared

Table 2. Overall area, instance count, and latency improvements due to clustering.

Circuit component	No clustering	Clustering	Diff (%)	No clustering	Clustering	Diff (%)	Latency improvement
Hit_Detect_512_Mux4	71,360	62,205	12.8	4,516	3,690	18.3	0.0
SISO	54,149	43,991	18.8	3,397	2,410	29.1	25.0
Accumulate16	6,648	5,599	15.8	405	325	19.8	8.3
Add_16	17,209	15,083	12.4	778	575	26.1	0.0
CRC_NoReg_Output	28,433	24,870	12.5	1,789	1,375	23.1	9.1
ECC_Parity	6,538	6,149	6.0	436	358	17.9	0.0
C3540	36,378	33,907	6.8	2,236	1,971	11.9	7.7
S1423	15,422	13,508	12.4	966	777	19.6	17.6
S298	3,585	3,117	13.1	220	178	19.1	10.0
S400	4,744	3,952	16.7	297	225	24.2	0.0
MAC16	83,930	62,720	25.3	5,138	3,450	32.9	33.3
Avg			13.9			22.0	10.1

the area of the final prelayout netlist with clustering turned off to that of a clustered netlist. The tables' examples show that clustering saves an average of 13.9% in total circuit area. As detailed in Table 1, this total can be decomposed into 8.9% less area in slack-matching buffers, 41.5% less area in fan-out-fix buffers, and 22.7% less area in C-elements and control cells. Table 2 shows that clustering can save as much as 25.3% in area. Moreover, it shows that clustering also reduces latency by an average of 10.1% by reducing the amount of cluster fan-out and thus the number of needed fan-out-fix pipeline buffers along latency-critical paths. Lastly, clustering reduces place-and-route complexity by 22% in terms of lower cell count.

Timing-driven place and route

The timing model we use in Proteus is essentially an amortized unit-delay model. Each I/O timing arc adds an integer number of delays to a timing budget. Paths are cut at predetermined key points to ensure that all timing loops are decomposed into a manageable number of segments in a manner similar to the work by Prakash.⁹ This decomposition is necessary because standard static timing and place-and-route tools don't support timing loops and have runtimes that are highly dependent on the number of given timing constraints. Specifically, we break paths at the inputs of all domino logic gates, effectively decomposing all paths through clusters of logic gates into acyclic segments.⁹ Each segment receives an explicit constraint on its maximum delay, generated

by ClockFree in the standard Synopsys Design Constraint (SDC) format. Standard place-and-route tools can then be used to automatically place and route the designs.

The PCHB library and the ClockFree synthesis engine are designed to target designs that run at 18 transitions per cycle. These 18 transitions can be decomposed into two transitions of forward latency through the domino logic and 16 transitions of backward latency through neighboring control circuits, the latter involving data acknowledgment, data reset, and subsequent acknowledgment reset.^{1,6} In 65-nm TSMC process technology, we allow 50 ps per transition. I/O timing arcs are associated with a multiple of this delay based on the number of transitions they represent. Segments are then given a constraint associated with the sum of the delay budget of their associated timing arcs.

By ensuring that all cycles are decomposed into segments, we thus generally have a 900-ps cycle time target. This is 1.11 GHz, two to three times faster than that typically achieved in this process using conventional synchronous flows. (Allowing segments to have budgets that are not a multiple of 50 ps is also possible and could generally lead to better circuits. In fact, intelligently giving segments fractional delay budgets in a way that also preserves a fixed delay target such as 900 ps around any cycle is an area of future work.)

Unfortunately, the place-and-route engine typically cannot initially meet all these constraints for anything

but the smallest blocks. Despite very sophisticated cell placement, sizing, and routing algorithms, the place-and-route tools often complete without meeting many constraints. Typically, the failure can be attributed to wires that end up too long. This is likely not only because our constraints are aggressive but also because the density of constraints per mm^2 is far greater than in a typical flow.

Fortunately, a feature of asynchronous designs is that pipeline buffers usually can be added to channels with no impact on the circuit's functionality. Thus, to address this failure problem, we've built scripts that extract information about the failing constraints and call ClockFree to edit the netlist by adding pipeline buffers to channels containing long wires. Of course, to maintain overall performance, once these pipeline buffers are added, ClockFree must also slack-match the entire design again before emitting the new netlist to the place-and-route engine. Moreover, ClockFree must not add pipeline buffers to any channel that it perceives might be globally critical because this may make the algorithmic cycle time exceed the target cycle time. Consequently, some long wires associated with globally critical channels might not explicitly be fixed. When this happens, we expect sufficient other channels to be shortened elsewhere in the design such that the place-and-route engine can compensate by moving cells associated with unfixed globally critical channels with long wires closer together without causing many other constraint violations. In particular, other channels may be shortened by our explicit introduction of pipeline buffers if they also contain problematic long wires or by the subsequent slack-matching routine.

To implement this performance-aware repipelining, we rely on our update-globally-critical algorithm that conservatively identifies which channels are globally critical. As with fan-out fixing, the basic challenge is that the set of globally critical channels can grow after each pipeline buffer is added; this algorithm must, therefore, be run after each pipeline buffer is added. Thus, this update algorithm's speed is critical to the scalability of this approach.

The difference between the original and new netlists resulting from this repipelining is treated as a place-and-route engineering change order (ECO), and the physical design process proceeds. Fortunately, for blocks of up to approximately 20,000 cell instances (each $1\text{-}2\text{ mm}^2$) with reasonable floorplans,

this process typically converges within two to six iterations and less than 12 hours of CPU time. For example, the final layout for the 16-bit accumulator takes only a few minutes to complete.

The relatively low complexity supported by place and route is a flow limitation addressed by hierarchy support. At each hierarchy level, a finalized block's layout is abstracted using the standard Library Exchange Format, and its interface arrival times, input capacitance, and output drive strength are characterized using the Liberty format. These macros then effectively become a black box in the next-higher level of the hierarchy. The flow thus supports Proteus blocks that include full-custom macros, Proteus blocks that include smaller Proteus blocks, as well as full-custom blocks that include smaller Proteus blocks. This enables us to arbitrarily scale the design's size while preserving our target performance.

Digital and analog verification

We have no formal process for verifying that the asynchronous implementation matches that of the CSP specification. Instead, we rely on a digital-simulation *cosimulation* environment to ensure that, for specific environments we create, the implementation matches the behavior of the CSP specification. We also use proprietary code coverage tools to ensure that our environments cover all cases of interest.

Analog concerns with these designs are also very important. Because of inaccuracies in library characterization, we validate each block's performance using analog simulation. In addition, we take extra care to analyze the noise sensitivity in our design, particularly because dynamic logic is often more sensitive to noise than its static counterparts. We have created an automated Spice-based simulation framework that conservatively tests for worst-case noise-induced bumps on all wires. We also paid careful attention to the sizing of transistors and staticizers, analyzed and minimized charge sharing as part of library design, set a reasonable global constraint on slew rates during place and route, and adhered to a reasonable global nonminimum wire-spacing requirement. As a result, even the most complex placed and routed blocks we've produced have passed this rigorous analog verification. One explanation for this result is that most wires are relatively short at these high frequencies, minimizing the potential for cross-talk noise.

For further validation of this flow, Fulcrum fabricated a 65-nm test chip that included several Proteus blocks ranging in complexity from 2,000 to 10,000 standard cells. They all worked as expected, running at 1.2 GHz.

Impact and conclusions

As part of the evaluation of this flow, we performed a detailed comparison of a full-custom block and its Proteus counterpart on a small circuit called pickClass, which implements a variant of deficit-weighted round-robin.¹³ This algorithm includes hierarchical accounting, prioritization, and traffic shaping over 16 queues. The full-custom implementation took a month to design, had a final latency of seven pipeline stages (14 transitions), and had a scaled logic area, excluding state, of 0.025-mm², in a TSMC 65-nm process technology. The first Proteus runs had no latency constraints during synthesis and yielded poor results. The latency was 70 pipeline stages, and the design was dominated by slack-matching buffers. Constraining latency during synthesis yielded a latency of 16 pipeline stages with a smaller logic area of 0.011 mm². Further rewriting of the CSP to better guide synthesis toward a low-latency solution led to a final design that matched the full-custom implementation's latency and had a logic area of 0.020 mm².

These Proteus experiments were all performed over the span of two days. The case study showed that, at least for small designs, Proteus could produce results comparable to those of full-custom design at a fraction of the design time. But it also showed that to get good results requires careful attention to the synthesis constraints, as well as the style of the initial CSP description.

DESPITE SUCCESSFULLY CLOSING timing on many more small- and medium-sized blocks, the Proteus flow is not yet mature. There are numerous areas in which further R&D can lead to its effective use on more and larger blocks, as well as to its generally providing smaller and lower power results. We know that more-advanced peephole optimization techniques can reduce the latency of critical algorithmic loops, making the tool effective on more blocks. Moreover, as we've mentioned, it's generally the place-and-route tool's convergence and runtimes that limit the complexity of blocks we can successfully run through the flow. We believe a tighter integration

with place-and-route engines is possible and can at least double the complexity of candidate blocks.

Other areas of the flow, however, may require more fundamental CAD research. For example, slack-matching buffers can typically represent 30% of the design's area. The current algorithm conservatively models the conditional circuit using an unconditional model, which can lead to unnecessary buffers being added. Efficiently slack-matching conditional circuits, as Gill et al. have demonstrated,¹² is an interesting and valuable area of research. ■

Acknowledgments

The Proteus flow would not have been possible without the tireless effort of staff of both TimeLess Design Automation and Fulcrum Microsystems. Specifically, Prasad Joshi of TimeLess was instrumental in implementing many aspects of the flow. Harry Liu and Aubrey Grey of Fulcrum developed CAST2RTL as well as the flow for cell/macro characterization. We also acknowledge Jonathan Dama of Fulcrum for providing the data on the pickClass case study. Lastly, we thank TimeLess Design Automation's mentors, who supported the business side of this effort with invaluable guidance and wisdom: Bill Collins, Ken Deemer, Ivan Sutherland, Ty Garibay, and Aiguo Xie.

References

1. A.M. Lines, "Pipelined Asynchronous Circuits," master's thesis, California Inst. of Technology, Record Number Caltech CSTR:1998.cs-tr-95-21, 1995.
2. A. Taubin et al., "Design Automation of Real-Life Asynchronous Devices and Systems," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 1, 2007, pp. 1-133.
3. A.M.G. Peeters, "Single-Rail Handshake Circuits," PhD thesis, Eindhoven Univ. of Technology, 1996.
4. I.E. Sutherland, "Micropipelines," *Comm. ACM*, vol. 32, no. 6, 1989, pp. 720-738.
5. A. Smirnov et al., "An Automated Fine-Grain Pipelining Using Domino Style Asynchronous Library," *Proc. 5th Int'l Conf. Application of Concurrency to System Design (ACSD 05)*, IEEE CS Press, 2005, pp. 68-76.
6. P.A. Beerel, M. Ferretti, and R.O. Ozdag, *A Designer's Guide to Asynchronous VLSI*, Cambridge Univ. Press, 2010.
7. A. Dasdan and R.K. Gupta, "Faster Maximum and Minimum Mean Cycle Algorithms for System-Performance

- Analysis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, 1998, pp. 889-899.
8. P.A. Beerel et al., "Slack Matching Asynchronous Designs," *Proc. Int'l. Symp. Asynchronous Circuits and Systems (ASYNC 06)*, IEEE CS Press, 2006, pp. 184-194.
 9. M. Prakash, "Library Characterization and Static Timing Analysis of Template Based Asynchronous Circuits," master's thesis, Ming Hsieh Dept. of Electrical Eng., Univ. of Southern California, 2007.
 10. G. Gill, J. Hansen, and M. Singh, "Loop Pipelining for High-Throughput Stream Computation Using Self-Timed Rings," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD 06)*, ACM Press, 2006, pp. 289-296.
 11. G.D. Dimou, "Clustering and Fanout Optimization for Asynchronous Circuits," doctoral dissertation, Ming Hsieh Dept. of Electrical Eng., Univ. of Southern California, 2009.
 12. G. Gill, V. Gupta, and M. Singh, "Performance Estimation and Slack Matching for Pipelined Asynchronous Architectures with Choice," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD 08)*, ACM Press, 2008, pp. 449-456.
 13. M. Shreedhar and G. Varghese, "Efficient Fair Queueing Using Deficit Round Robin," *ACM SIGCOMM Computer Communication Rev.*, vol. 25, no. 4, 1995, pp. 231-242.

Peter A. Beerel is an associate professor in the Department of Electrical Engineering—Systems and the faculty director of Innovation and Entrepreneurship in Engineering, at the University of Southern California's Viterbi School of Engineering. He is also Chief Scientist

at Fulcrum Microsystems. His research interests include a variety of topics in CAD and asynchronous VLSI design. He has a PhD in electrical engineering from Stanford University and is an IEEE Senior Member.

Georgios D. Dimou is a principal engineer at Fulcrum Microsystems. His research interests include CAD and VLSI design, with an emphasis on asynchronous circuits, as well as the design of digital telecommunication and signal processing systems. He has a PhD in electrical engineering from the University of Southern California.

Andrew M. Lines is a cofounder of Fulcrum Microsystems, where he leads the company's technical innovation, ranging from chip architecture and circuit design to CAD tools and methodologies. His interests include leveraging Fulcrum's clockless-technology foundation to offer high-performance chip products, currently for Ethernet networking. He has an MS in computer science from the California Institute of Technology.

■ Direct comments and questions about this article to Peter A. Beerel, University of Southern California, Ming Hsieh Dept. of Electrical Eng., EEB-350, 3740 McClintock Ave., Los Angeles, CA 90089; pabeerel@usc.edu.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

PURPOSE: The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.

MEMBERSHIP: Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

COMPUTER SOCIETY WEBSITE: www.computer.org

OMBUDSMAN: To check membership status or report a change of address, call the IEEE Member Services toll-free number, +1 800 678 4333 (US) or +1 732 981 0060 (international). Direct all other Computer Society-related questions—magazine delivery or unresolved complaints—to help@computer.org.

CHAPTERS: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

AVAILABLE INFORMATION: To obtain more information on any of the following, contact Customer Service at +1 714 821 8380 or +1 800 272 6657:

- Membership applications
- Publications catalog
- Draft standards and order forms
- Technical committee list
- Technical committee application
- Chapter start-up procedures
- Student scholarship information
- Volunteer leaders/staff directory
- IEEE senior member grade application (requires 10 years practice and significant performance in five of those 10)

PUBLICATIONS AND ACTIVITIES

Computer: The flagship publication of the IEEE Computer Society, *Computer*, publishes peer-reviewed technical content that covers all aspects of computer science, computer engineering, technology, and applications.

Periodicals: The society publishes 13 magazines, 18 transactions, and one letters. Refer to membership application or request information as noted above.

Conference Proceedings & Books: Conference Publishing Services publishes more than 175 titles every year. CS Press publishes books in partnership with John Wiley & Sons.

Standards Working Groups: More than 150 groups produce IEEE standards used throughout the world.

Technical Committees: TCs provide professional interaction in more than 45 technical areas and directly influence computer engineering conferences and publications.

Conferences/Education: The society holds about 200 conferences each year and sponsors many educational activities, including computing science accreditation.

Certifications: The society offers two software developer credentials. For more information, visit www.computer.org/certification.

NEXT BOARD MEETING

13–14 Nov., New Brunswick, NJ, USA



revised 24 August 2011

EXECUTIVE COMMITTEE

President: Sorel Reisman*

President-Elect: John W. Walz*

Past President: James D. Isaak*

VP, Standards Activities: Roger U. Fujii†

Secretary: Jon Rokne (2nd VP)*

VP, Educational Activities: Elizabeth L. Burd*

VP, Member & Geographic Activities: Rangachar Kasturi†

VP, Publications: David Alan Grier (1st VP)*

VP, Professional Activities: Paul K. Joannou*

VP, Technical & Conference Activities: Paul R. Croll†

Treasurer: James W. Moore, CSDP*

2011–2012 IEEE Division VIII Director: Susan K. (Kathy) Land, CSDP†

2010–2011 IEEE Division V Director: Michael R. Williams†

2011 IEEE Division Director V Director-Elect: James W. Moore, CSDP*

*voting member of the Board of Governors †nonvoting member of the Board of Governors

BOARD OF GOVERNORS

Term Expiring 2011: Elisa Bertino, Jose Castillo-Velázquez, George V. Cybenko, Ann DeMarle, David S. Ebert, Hironori Kasahara, Steven L. Tanimoto

Term Expiring 2012: Elizabeth L. Burd, Thomas M. Conte, Frank E. Ferrante, Jean-Luc Gaudiot, Paul K. Joannou, Luis Kun, James W. Moore

Term Expiring 2013: Pierre Bourque, Dennis J. Frailey, Atsuhiko Goto, André Ivanov, Dejan S. Milojcic, Jane Chu Prey, Charlene (Chuck) Walrad

EXECUTIVE STAFF

Executive Director: Angela R. Burgess

Associate Executive Director; Director, Governance: Anne Marie Kelly

Director, Finance & Accounting: John Miller

Director, Information Technology & Services: Ray Kahn

Director, Membership Development: Violet S. Doan

Director, Products & Services: Evan Butterfield

COMPUTER SOCIETY OFFICES

Washington, D.C.: 2001 L St., Ste. 700, Washington, D.C. 20036-4928

Phone: +1 202 371 0101 • **Fax:** +1 202 728 9614

Email: hq.ofc@computer.org

Los Alamitos: 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314

Phone: +1 714 821 8380

Email: help@computer.org

MEMBERSHIP & PUBLICATION ORDERS

Phone: +1 800 272 6657 • **Fax:** +1 714 821 4641 • **Email:** help@computer.org

Asia/Pacific: Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan

Phone: +81 3 3408 3118 • **Fax:** +81 3 3408 3553

Email: tokyo.ofc@computer.org

IEEE OFFICERS

President: Moshe Kam

President-Elect: Gordon W. Day

Past President: Pedro A. Ray

Secretary: Roger D. Pollard

Treasurer: Harold L. Flescher

President, Standards Association Board of Governors: Steven M. Mills

VP, Educational Activities: Tariq S. Durrani

VP, Membership & Geographic Activities: Howard E. Michel

VP, Publication Services & Products: David A. Hodges

VP, Technical Activities: Donna L. Hudson

IEEE Division V Director: Michael R. Williams

IEEE Division VIII Director: Susan K. (Kathy) Land, CSDP

President, IEEE-USA: Ronald G. Jensen