Timed Circuits: A New Paradigm for High-Speed Design *

Chris J. Myers Wendy Belluomini Kip Killpack Eric Mercer Eric Peskin Hao Zheng
Department of Electrical Engineering
University of Utah
Salt Lake City, UT 84112
e-mail: myers@ee.utah.edu

Abstract— In order to continue to produce circuits of increasing speeds, designers must consider aggressive circuit design styles such as self-resetting or delayed-reset domino circuits used in IBM's gigahertz processor (GUTS) and asynchronous circuits used in Intel's RAPPID instruction length decoder. These new timed circuit styles, however, cannot be efficiently and accurately analyzed using traditional static timing analysis methods. This lack of efficient analysis tools is one of the reasons for the lack of mainstream acceptance of these design styles. This paper discusses several industrial timed circuits and gives an overview of our timed circuit design methodology.

I. Introduction

To achieve high performance, designers must consider aggressive *timed circuit* design styles. Timed circuits are defined to be any circuits that are optimized using explicit timing information. One example is the *self-resetting* and *delayed-reset domino* circuits used in IBM's gigahertz research microprocessor. Much of the improvement in speed in this processor can be attributed to these aggressive circuit styles [7]. Designers are also considering asynchronous circuits due to their potential for higher performance and lower power as demonstrated by Intel's RAPPID instruction length decoder [12]. This design was 3 times faster while using only half the power of the comparable synchronous design. These new circuit styles, however, cannot be efficiently and accurately analyzed using traditional static timing analysis methods. This lack of efficient analysis tools is one of the reasons for the lack of mainstream acceptance of these design styles.

It is impossible to reference the substantial amount of work that has been done in asynchronous design and timing verification in this short paper. An annotated bibliography can be found in our forthcoming book [9]. The goal of this paper is to describe several industrial timed circuit designs, and to give a overview of our timed circuit design methodology.

*This research is supported by NSF CAREER award MIP-9625014, SRC contracts 97-DJ-487 and 99-TJ-694, and a grant from Intel Corporation.

II. Design Motivations

This section describes three industrial designs which have guided the development of our timed circuit design methodology. The first is the Intel RAPPID chip which is a fully asynchronous instruction length decoder which is 3 times faster while using only half the power of the comparable synchronous design. RAPPID's speed is derived from a highly timed asynchronous design. The second design is IBM's gigahertz processor, GUTS. This was the first CMOS processor to run over 1 GHz using 1997 process technology. Its speed is derived from a highly timed synchronous design. Finally, Sonic Innovation's digital hearing aid provided a different sort of guide to our design methodology as its objective is low power and small area. We designed a key component, a multiplier. Our early analysis shows that our 24-bit design uses $\frac{1}{7}$ the area and only $\frac{1}{3}$ the power of a synchronous array.

A. Intel's RAPPID

Instructions in the x86 architecture can be from 1 to 15 bytes long depending on a large number of factors. In order to allow concurrent execution of x86 instructions, it is necessary to rapidly determine the positions of each instruction in a cache line. This was at the time a critical bottleneck in the x86 architecture. The length of instructions is determined using the following rules:

- Opcode can be 1 or 2 bytes.
- Opcode determines presence of the ModR/M byte.
- ModR/M determines presence of the SIB byte.
- ModR/M and SIB set length of displacement field.
- Opcode determines length of immediate field.
- Instructions may be preceded by upto 15 prefix bytes.
- A prefix may change the length of an instruction.
- The maximum instruction length is 15 bytes.

For real applications, it turns out that there are only a few common instruction lengths. As shown in Figure 1, 75 percent of instructions are 3 bytes or less in length. Nearly all instructions are 7 bytes or less. It is also the case that prefix bytes are extremely rare. This presents

an opportunity for an asynchronous design to optimize for the common case by optimizing for instructions of length 7 or less with no prefix bytes. Other less efficient methods are then used for longer instructions [6] and instructions with prefix bytes [4].

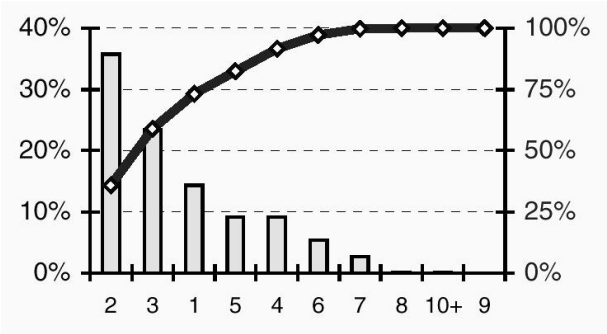

Fig. 1. Histogram for proportion of x86 instruction lengths and cummulative length statistics.

The RAPPID microarchitecture is shown in Figure 2. The RAPPID decoder reads in a 16 byte cache line, and it decodes each byte as if it is the first byte of a new instruction. The decode logic is implemented using large unbalanced trees of combinational logic that have been optimized for common instructions. Each byte speculatively determines the length of an instruction beginning with this byte. It does this by looking at possibly up to three additional downstream bytes. The actual first byte of the current instruction is marked with a tag. This byte uses the length that it determined to decide which byte is the first byte of the next instruction. It then signals that byte while notifying all bytes inbetween to squash their length calculations and forwards the bytes of the current instruction to an output buffer. In order to improve performance, four rows of tag units and output buffers are used in a round-robin fashion. In the case of a branch, the tag is forwarded to a branch unit that determines where to inject the tag back into the new cache line [5].

The key to achieving high performance is the tag unit, which must be able to rapidly tag instructions. The timed circuit for one tag unit is shown in Figure 3. Assuming that the instruction is ready (i.e., *InstRdy* is high indicating one $Length_i$ is high and all bytes of the instruction are available) and the crossbar is ready (i.e., *XBRdy* is high), then when a tag arrives (i.e., one of $TagIn_j$ is high), the first byte of the next instruction can be tagged within two gate delays (i.e., $TagOut_i$ is set to high). In other words, a synchronization signal can be created every two gate delays. It is difficult to imagine distributing a clock which has a period of only two gate delays. The tag unit in the chip is capable of tagging up to 4.5 instructions/ns.

This circuit, however, requires timing assumptions for correct operation. In typical asynchronous communication, a request is transmitted followed by an acknowledge being received to indicate that the circuit can reset. In this case, there is no explicit acknowledgment, but rather acknowledgement comes by way of a timing assumption. Once a tag arrives (i.e., *TagArrived* is high), if the instruction and crossbar are ready, the course is set to begin to reset *TagArrived*. The result is that the signal produced on $TagOut_i$ is a pulse. Let us consider now the affect of receiving a pulse on a *TagIn* signal. If either the instruction or crossbar are not ready, then *TagArrived* gets set by the pulse in effect latching the pulse. *TagArrived* will not get reset by the disappearence of the pulse but rather the arrival of a state in which both the instruction and crossbar are ready.

For this circuit to operate correctly, there are two critical timing assumptions. First, the pulse created must be long enough to be latched by the next tag unit. This can be satisfied by adding delay to the AND gate used to reset *TagArrived*. An arbitrary amount of delay, however, cannot be added since the pulse must not also be so long that another pulse could come before the circuit has reset. Therefore, we have a *two-sided timing constraint*. Our tool `ATACS` is designed to synthesize and analyze circuits with such types of constraints. `ATACS` was used to synthesize and analyze the tag circuit from RAPPID [12].

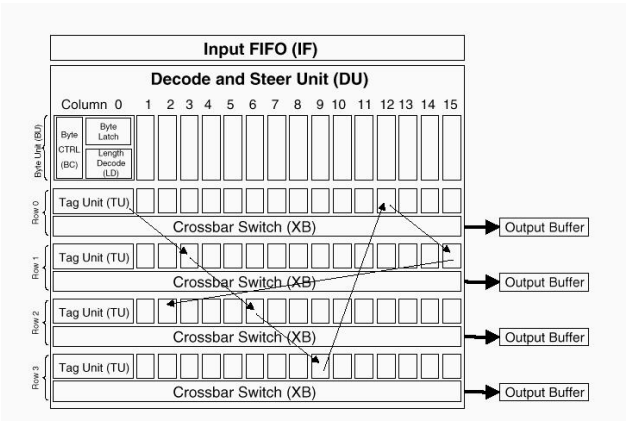

Fig. 2. RAPPID Microarchitecture.

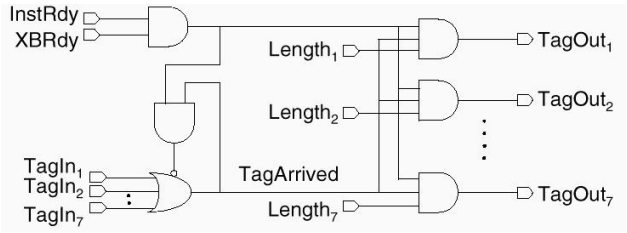

Fig. 3. The tag unit circuit.

B. IBM's GUTS Microprocessor

The next timed circuit design is IBM's gigahertz research microprocessor, GUTS. The key achieving such high performance was the use of aggressive circuit styles, namely, self-resetting and delayed-reset domino. While in this case, the design is synchronous, there are numerous local timing assumptions that must be satisfied for correct operation. These timing assumptions again pose two-sided timing constraints which are difficult to analyze using traditional static timing methods.

An example of a simple delayed reset domino circuit is shown in Figure 4. This circuit implements the function *out2* = (*a* or *b*) and *c*. The signals *clk1* and *clk2* are delayed versions of the global clock. If either *a* or *b* go high, then *out1* goes high. If *c* is also high, then *out2* goes high. Some short time after *out2* goes high, *clk1* goes low causing *out1* to reset to a low value. The timing of *clk2* to go low and precharge *out2* is set such that *out2* has time to be used by the next gate. In other words, *out1* and *out2* are pulses.

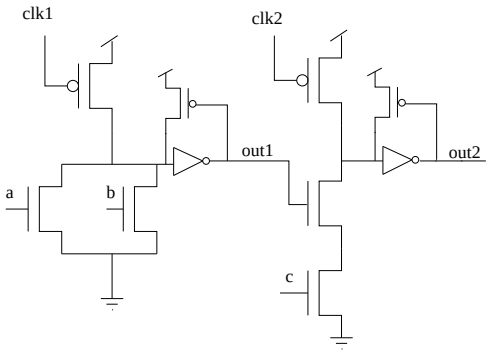

Fig. 4. A simple delayed-reset domino circuit that implements *out2* = (*a* or *b*) and *c*.

There are several timing assumptions required by this circuit. First, the pulldown stack must be hazard-free. A glitch by one of these circuits could cause the next gate to erroneously believe that it received a pulse. Second, the pulldown stack must stay on long enough to discharge the output node. This means the pulse must have a minimum width. Third, all inputs to the gate must turn off before the precharge phase begins. This means the pulse has a maximum width. Therefore, there is again a two-sided timing constraint.

The GUTS design also employs self-resetting logic such as the PLA controller shown in Figure 5. This circuit waits for a sufficient number of dual-rail inputs to indicate the arrival of valid data. It then sets the *propagate control* line high. At the same time, the signal is transmitted through a series of buffers which when fed back have the affect of resetting the *propagate control* signal. This creates a pulse on the *propagate control* signal. This type of circuit is called self-resetting because the setting of the signal puts into motion a series of events that leads to the resetting of the signal. The correctness of this circuit also depends on the satisfication of a two-sided timing constraint. ATACS was used to verify the PLA controller and several other circuits from the GUTS processor [2].

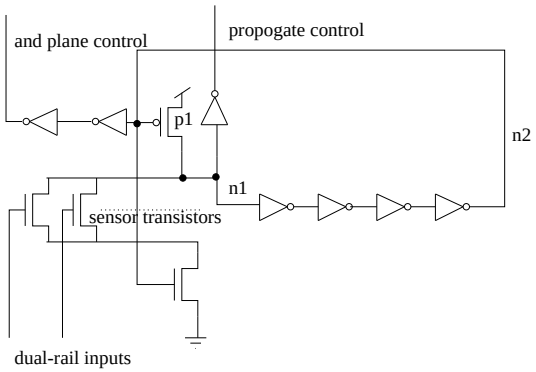

Fig. 5. PLA controller

C. Sonic Innovation's Hearing Aid

The last design example is a self-timed iterative multiplier that we are designing for a digital hearing aid application [8]. The goal of this design is quite different from the other two in that the design must have low power consumption and use a small chip area. We determined that an iterative multiplier using radix 4 Booth encoding met our delay constraints, and it has the best area and power. In a low power design, such as a hearing aid, it is not desirable to distribute a high-speed clock due to its power consumption and the interference it has with the analog circuitry. So a synchronous iterative multiplier in this application is not attractive. Therefore, we designed a self-timed multiplier in which iterations are controlled by a locally generated clock. While the rest of the design is fairly conventional and designed using standard cells, the clock generation circuit must be very carefully designed to meet the needed timing constraints. Again, the ATACS tool is ideally suited to the task. Although the multiplier is self-timed, it can be easily embedded in a synchronous system as long as the clock rate is long enough that the multiply has time to complete. The area of an N-bit multiplier is $O(N)$ as opposed to $O(N^2)$ for the synchronous array multiplier used in the original hearing aid. For a 24-bit word, the self-timed multiplier is $\frac{1}{7}$ the size of the synchronous array. While the power grows polynomially for both designs, the self-timed design has a much lower coefficient than the array. The power consumed by the self-timed multiplier with a 24-bit word size is $\frac{1}{3}$ that of the synchronous array.

III. Timed Circuit Design Methodology

We describe our timed circuit design methodology using a simple example. In a small town in Southern Utah, there's a little winery with a wine shop nearby. Being a small town in a community who thinks prohibition still exists, there is only one wine patron. The shop has a single small shelf capable of holding only a single bottle of wine. The winery and shop communicate a bottle of wine over a *channel*. A channel is simply a point-to-point means of communication between two concurrently operating processes. One process uses that channel to send data to the other process. The channel level block diagram for our example is shown in Figure 6.

Fig. 6. Channel block diagram for *wine_shop*.

The behavior of the winery, shop, and patron can be represented in VHDL as shown in Figure 7. This code uses two new packages: `nondeterminism` and `channel`. The nondeterminism package defines some functions to generate random delays and random selections for simulation. The channel package includes a definition of the channel data type and operations on it such as *send* and *receive*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity wine_example is
end wine_example;
architecture behavior of wine_example is
  type wine_list is (chardonnay, merlot);
  signal wine_drunk:wine_list;
  signal WineryShop:channel:=init_channel;
  signal ShopPatron:channel:=init_channel;
  signal bottle,shelf,bag:std_logic;
begin
winery:process
begin
  bottle <= selection(2);
  send(WineryShop,bottle);
  wait for delay(5,10);
end process winery;
shop:process
begin
  receive(WineryShop,shelf);
  send(ShopPatron,shelf);
end process shop;
patron:process
begin
  receive(ShopPatron,bag);
  wine_drunk <= wine_list'val(conv_integer(bag));
end process patron;
end behavior;
```

Fig. 7. Channel level model for the *wine_shop*.

For this example, we have defined two channels for communication. The *WineryShop* channel is used for delivering bottles of wine to the shop and the *ShopPatron* channel is used for selling bottles of wine to the patron. Both channels are initialized using the *init_channel* function. The behavior of the *winery* begins by randomly selecting whether to produce chardonnay or merlot. Next, it sends this bottle of wine to the shop with the procedure call *send*. This procedure has two parameters: a channel to communicate on and the data to be transmitted. The last step is that the winery waits for some random time between 5 and 10 minutes until it is ready to make another bottle of wine, and it then repeats forever. The behavior of the *shop* begins by receiving a bottle of wine from the winery with the procedure call *receive*. This procedure also has two parameters: a channel to communicate on and a location where the data is to be copied upon reception. After receiving the wine, the shop sends it to the patron over the *ShopPatron* channel. The behavior of the *patron* begins by receiving a bottle of wine which it then identifies (probably with a small sip). It then waits for the shop to send another bottle of wine.

A channel communication is implemented using a handshake protocol on two or more signal wires. Our example uses a *dual-rail protocol* (i.e., two wires) to encode the type of wine being transmitted and a third wire to acknowledge communication (see Figure 8).

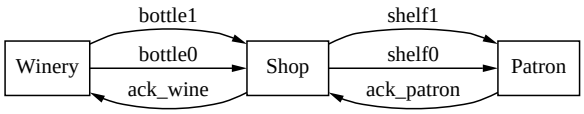

Fig. 8. Handshaking block diagram for *wine_shop*.

The behavior of the winery, shop, and patron can be represented in VHDL at the handshake level as shown in Figure 9. The model uses the *handshake* package instead of the *channel* package. This package includes the definitions of the procedures: *guard*, *guard_or*, *guard_and*, *assign*, and *vassign*. All signals of type channel are replaced with signals of type *std_logic*. The first of these new signals are *bottle1* and *bottle0* which are used to deliver a new bottle of chardonnay or merlot, respectively. The next signal, *ack_wine*, is used to indicate acknowledgment of the wine delivery to the shop. The *ShopPatron* channel is implemented with the signals *shelf1*, *shelf0*, and *ack_patron*.

Consider the behavior of the shop at the handshake level. In this protocol, the first thing the shop does is wait until *ack_patron* is '0' using the *guard* procedure. The procedure *guard*(s, v) takes a signal, s, and a value, v and waits until $s = v$. The next step in the protocol is to wait

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;
entity wine_example is
end wine_example;
architecture hse of wine_example is
  signal bottle1,bottle0:std_logic;
  signal ack_wine:std_logic:='0';
  signal shelf1,shelf0:std_logic:='0';
  signal ack_patron:std_logic;
begin
winery_dualrail:process
begin
  z:=selection(2);
  if (z=1) then assign(bottle0,'1',5,inf);
  else assign(bottle1,'1',5,inf);
  end if;
  guard(ack_wine,'1');
  vassign(bottle0,'0',5,7,bottle1,'0',5,7);
  guard(ack_wine,'0');
end process;
shopPA_dualrail:process
begin
  guard(ack_patron,'0');
  guard_or(bottle0,'1',bottle1,'1');
  if bottle0 = '1' then assign(shelf0,'1',1,2);
  elsif bottle1 = '1' then assign(shelf1,'1',1,2);
  end if;
  assign(ack_wine,'1',1,2);
  guard(ack_patron,'1');
  vassign(shelf0,'0',1,2,shelf1,'0',1,2);
  guard_and(bottle0,'0',bottle1,'0');
  assign(ack_wine,'0',1,2);
end process;
patron_dualrail:process
begin
  guard_or(shelf0,'1',shelf1,'1');
  assign(ack_patron,'1',2,3);
  guard_and(shelf0,'0',shelf1,'0');
  assign(ack_patron,'0',2,3);
end process;
end hse;
```

Fig. 9. Handshaking level model for the *wine_shop*.

until either *bottle0* or *bottle1* goes high using the *guard_or* procedure. The procedure *guard_or*($s1$, $v1$, $s2$, $v2$, ...) takes a set of signals and values and stalls a process until some signal s_i has taken value v_i. After *bottle0* or *bottle1* goes high, the protocol next sets *shelf0* or *shelf1* high using the *assign* procedure. The procedure *assign*(s, v, l, u) takes a signal, s, a value, v, a lower bound of delay, l, and an upper bound of delay, u. After the appropriate shelf signal goes high, *ack_wine* is set high and the shop waits for *ack_patron* to go high. Next, the shop resets the shelf signal. At this point, only one of the two shelf signals is high, so the *vacuous assign (vassign)* procedure is used since one assignment does nothing. After the shelf signals are reset, the shop waits until both bottle signals are low using the *guard_and* procedure. The procedure *guard_and*($s1$,$v1$,$s2$,$v2$,...) takes a set of signals and a set of values, and it stalls a process until each signal s_i has taken value v_i. Finally, the shop protocol assigns *ack_wine* to '0' and loops back to the beginning. A speed-independent circuit for the shop is shown in Figure 10(a).

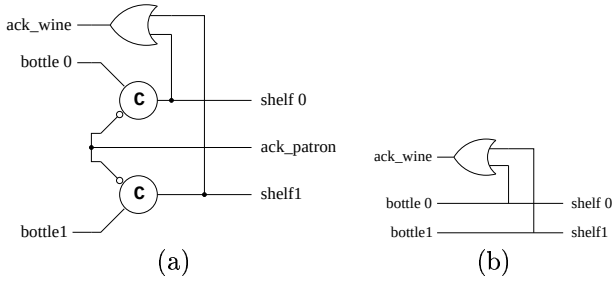

Fig. 10. (a) Speed-independent and (b) timed circuit.

Consider now the timing information that is known about the winery, shop, and the patron. First, the winery can produce a new bottle of wine every 5 minutes, but it make take an infinite amount of time when the wine making machine is broken. It resets its handshakes after 5 to 7 time units. The patron lives close to the shop so it can accept a new bottle of wine in 2 to 3 minutes after being called, and it resets after 2 to 3 minutes. The shop always responds within 1 to 2 minutes. Using this delay information, the circuit can be optimized as shown in Figure 10(b).

IV. POSET Timing

In order to synthesize and verify timed circuits, it is necessary to efficiently find all reachable timed states. Approaches based on regions or discrete time rapidly explode. Zones can do better, but explode for highly concurrent systems. We developed POSET timing which performs analysis on partially ordered sets of events rather than linear sequences [11, 10, 3, 1]. This eliminates false causality, and it can be orders of magnitude more efficient. The runtimes for the verification of various sizes of a stari circuit, a self-timed FIFO, are shown in Figure 11. It has been shown that a region based tool, `timed COSPAN`, runs out of 1 GByte of memory for 3 stages.

POSET timing still suffers from state explosion for modest size designs. Therefore, we are developing techniques that use the hierarchical information to decompose the design into components for individual analysis. We have formally proven that the result of these synthesis or verification runs produce correct but conservative results [13]. Our preliminary analysis shows that automatic abstraction can be substantially more efficient in both memory and time. Results for the verification of a timed FIFO designed at SUN are shown in Figure 12. While traditional methods can only verify 4 stages, we can easily verify 100 stages in about 20 minutes.

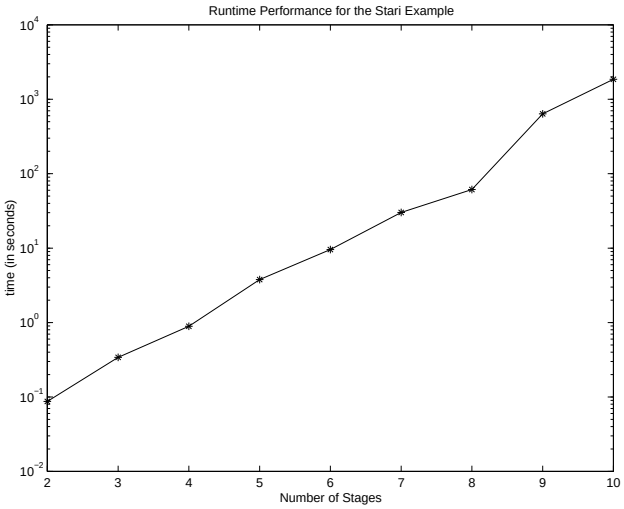

Fig. 11. POSET timing results for stari example.

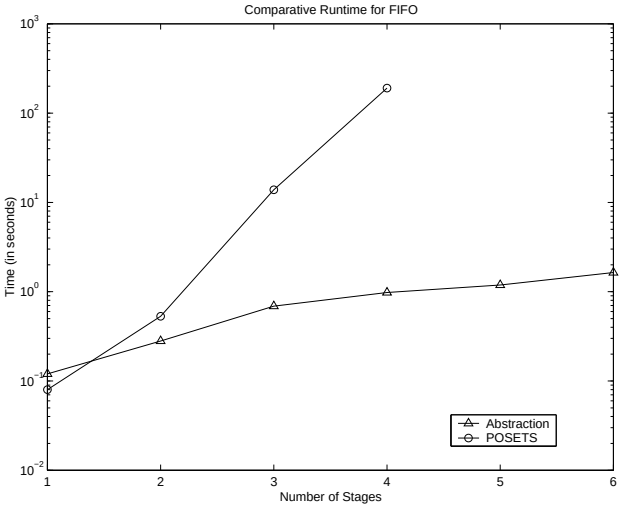

Fig. 12. Automatic abstraction results for SUN FIFO.

V. Conclusions

This paper describes the importance of timed circuits through several industrial-scale examples which have been designed or verified using ATACS. This paper has also given an overview of our timed circuit design methodology. Much more work though is needed to make timed circuit design practical. First, we need to develop automatic translation techniques from the channel level model to the handshaking level model. We need to continue to develop our automatic abstraction techniques for synthesis and verification. Even with good abstraction, we still need ever more efficient POSET techniques for verification when components are large. Since for verification the entire state space is not necessary, we are developing partial order techniques which when combined with POSET timing can efficiently yield a result. Finally, more design examples are needed to test the timed circuit methodology.

Acknowledgements

We would like to thank the many other people that have been involved in the ATACS project: Brandon Bachman, Jeff Cuthbert, Jie Dai, Hans Jacobson, Sung-tae Jung, Chris Krieger, Scott Little, Curt Nelson, Allen Sjogren, and Robert Thacker.

References

[1] W. Belluomini and C. J. Myers. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided-Design*, May 2000.

[2] W. Belluomini, C. J. Myers, and H. P. Hofstee. Verification of delayed-reset domino circuits using ATACS. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, April 1999.

[3] W. Belluomini and C.J. Myers. Verification of timed systems using posets. In *International Conference on Computer Aided Verification*. Springer-Verlag, 1998.

[4] R. Ginosar, R. Kol, K. Stevens, P. Beerel, K. Yun, C. Myers, and S. Rotem. Apparatus and method for self-timed marking of variable length instructions having length-affecting prefix bytes. Patent granted September 7, 1999, patent number 5,948,096.

[5] R. Ginosar, R. Kol, K. Stevens, P. Beerel, K. Yun, C. Myers, and S. Rotem. Branch instruction handling in a self-timed marking system. Patent granted August 3, 1999, patent number 5,931,944.

[6] R. Ginosar, R. Kol, K. Stevens, P. Beerel, K. Yun, C. Myers, and S. Rotem. Efficient self-timed marking of lengthy variable length instructions. Patent granted August 24, 1999, patent number 5,941,982.

[7] H. P. Hofstee, S. H. Dhong, D. Meltzer, K. J. Nowka, J. A. Silberman, J. L. Burns, S. D. Posluszny, and O. Takahashi. Designing for a gigahertz. *IEEE MICRO*, May-June 1998.

[8] K. C. Killpack, E. Mercer, and C. J. Myers. A standard-cell self-timed multiplier for power and area critical synchronous systems. In *Advanced Research in VLSI*. CS Press, 2001.

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

[10] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. POSET timing and its application to the synthesis and verification of gate-level timed circuits. *IEEE Transactions on Computer-Aided Design*, 18(6):769–786, June 1999.

[11] T. G. Rokicki and C. J. Myers. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.

[12] Shai Rotem, Ken Stevens, Ran Ginosar, Peter Beerel, Chris Myers, Kenneth Yun, Rakefet Kol, Charles Dike, Marly Roncken, and Boris Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, April 1999.

[13] H. Zheng and C. J. Myers. Automatic abstraction for synthesis and verification of deterministic timed systems. In collection of papers from TAU'00.