Automatic Synthesis of 3D Asynchronous State Machines

Kenneth Y. Yun David L. Dill

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

We describe a new *automatic synthesis tool* (3D) for designing asynchronous controllers from burst-mode specifications, a class of specifications allowing multiple input change fundamental mode operation. We present an algorithm for constructing a *three-dimensional* next-state table, a heuristic for encoding states, and a procedure for generating necessary constraints for exact logic minimization. We demonstrate the effectiveness of the 3D implementation and the synthesis procedure on numerous designs including a large realistic example (Asynchronous Data Transfer Protocol of the SCSI Bus Controller). We estimate the *latency* (input to output delay) and the *cycle time* (time required for the circuit to stabilize after the excitation) for all benchmark designs using a 0.8μm CMOS standard cell library.

1 Introduction

As the digital systems become more complex, it is increasingly attractive to use components operating at different clock rates or components not requiring clocks at all, such as asynchronous FIFOs and bus interface units. In such systems, the asynchronous designs are much better suited for interface circuits and controllers than the synchronous ones. The synchronous components require resynchronization of signals originating from modules operating at different clock rates. Upgrading one component may require the redesign of the entire system; however, the asynchronous components are inherently modular. The synchronous components must be designed for the worst case timing over all possible variations in power supply voltage, operating temperature and fabrication process, but the asynchronous components are robust under various environmental assumptions. The setup and hold time requirements coupled with the clock skew (exacerbated due to the complexity in global clock distribution) becomes a significant fraction of a clock cycle in the synchronous design; however, there is no such overhead in the asynchronous design.

In our previous paper [16], we have introduced a new asynchronous controller, called the *3D asynchronous state machine*, and its synthesis method. This design style uses *burst-mode* specifications, a class of specifications allowing multiple input change fundamental mode operation. Our implementation uses *standard combinational logic*, generates *low latency outputs* and guarantees*freedomfrom hazard at the gate level*. Unlike locally clocked burst-mode machines [10], it requires *no locally-synthesized clock* and *no explicit storage elements*. In addition, primary outputs as well as additional state variables are used as feedback variables. Furthermore, the 3D machines do not require state bits to be encoded in the original specification before synthesis can begin; cf. USC/CSC property [3, 7, 8, 5, 9, 15].

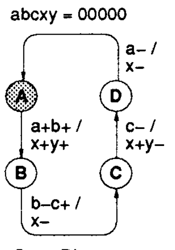
In this paper, we present an *automatic synthesis procedure* for the 3D asynchronous state machines; in particular, we describe an algorithm for constructing a three-dimensional next-state table, a simple but efficient state encoding heuristic, and a procedure for generating constraints for exact logic minimization [11]. Finally, we demonstrate the effectiveness of the 3D implementation and the synthesis procedure using benchmark designs including a large realistic example (Asynchronous Data Transfer Protocol of the SCSI Bus Controller).

2 Overview

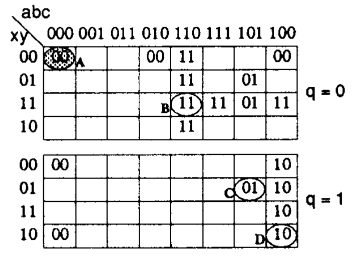
2.1 Specification

An asynchronous state machine allowing multiple-input changes is specified by a state diagram [10, 16]. A state diagram contains a finite set of states, a set of labelled arcs connecting pairs of states, and a start state. Arcs are labelled with possible transitions from one state to another. Each transition consists of a *non-empty* set of inputs (an *input burst*) and a set of outputs (an *output burst*). Note that every input burst must be non-empty; if no inputs change, the machine is stable.

In a given state, when all the inputs in the specified input burst have changed value, the machine generates the corresponding output burst and moves to a new state. Only specified input changes may occur, and input transitions may arrive in arbitrary order; however, the next set of input transitions (the next input burst) may not arrive until the machine is stabilized (*fundamental mode environmental assumption*). There is an implicit restriction to such *burst-mode* specification — no input burst in a given state can be a subset of another. An example of a burst-mode specification is shown in figure 1. This specification describes a simple controller having 3 inputs (a, b, c) and 2 outputs (x, y). s^+ and s^- denote 0–1 and 1–0 transitions of the signal s.



State Diagram



q	xy \ abc	000	001	011	010	110	111	101	100
q = 0	00	00 A			00	11			00
	01					11		01	
	11					B 11	11	01	11
	10					11			
q = 1	00	00							10
	01							C 01	10
	11								10
	10	00							D 10

Next–State Table for outputs xy

Figure 1: Example (Specification and Next-state Table).

2.2 Implementation

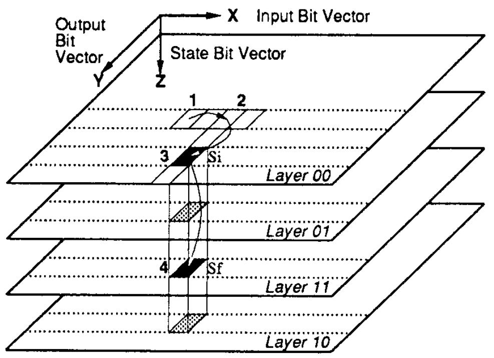
Formally, a 3D asynchronous finite state machine can be defined as a 4-tuple (X, Y, Z, δ) where

- X is a set of primary input symbols;
- Y is a set of primary output symbols;
- Z is a (possibly empty) set of internal state variable symbols;
- $\delta : X \times Y \times Z \rightarrow Y \times Z$ is a *next-state function.*

The hardware implementation of the 3D state machine is a two-level AND-OR network where outputs (and additional state variables when necessary) are fed back as inputs to the network. There are no explicit storage elements such as latches, flip-flops or C-elements in a 3D machine; only static feedback is used to maintain memory.

The 3D implementation of the burst-mode specification is obtained from the 3-dimensional function map called the ***next-state table***, a 3-dimensional tabular representation of the ***next-state function*** δ (see figure 1). In general, the next-state table is ***incompletely specified***; however, the next state of every "reachable" state must be completely specified.

The operation of the 3D state machine is similar to a Mealy-mode synchronous state machine (see figure 2). A machine cycle consists of 3 *phases* (input burst followed by output burst followed by state burst). During the idle state, the machine waits for an input burst to occur. When the last input transition of the input burst arrives, an output burst takes place. The state burst, if required, immediately follows the output burst, completing the 3-phase machine cycle.



1 □ Start of Input Burst
2 □ Start of Output Burst (End of Input Burst)
3 ■ Start of State Burst (End of Output Burst)
4 ■ New Stable State (End of State Burst)
▨ *Critical Race exists if different from 3 and 4*

Figure 2: 3D Next-state Table.

2.3 Considerations for Hazards

We can classify all hazards in asynchronous circuits into two categories: *function hazard* and *logic hazard*. Function hazards are due to the incompletely or incorrectly specified function during multiple input changes.[1] Logic hazards arise due to the delay variations of the physical gates [16, 10, 1, 2] despite the correct function. We can further classify logic hazards into combinational and sequential logic hazards.

In 3D machines, we preclude the presence of *function hazards* by correctly specifying the next-state of every "reachable" state during each of the bursts [16]. The requirements to insure hazard-free combinational logic during each of the bursts are presented in [16]. The following is a summary of the covering requirements for output logic (Similar requirements exist for state logic).

- For a 0–1 transition of output:

  The output burst must be covered by a single cube.

- For a 1–1 transition of output:

  The input burst must be covered by a single cube; the output burst must be covered by a single cube.

- For a 1–0 transition of output:

  Suppose n input transitions constitute an input burst enabling 1–0 transition of an output. The input burst must be covered by n cubes, each of which contains exactly one literal that corresponds to a unique input in the input burst. Thus, each cube changes monotonically from 1 to 0 as the corresponding input transition fires.

  In addition, we require that any cube that intersects ***transient states*** of an input burst B_{in} (states traversed during B_{in} preceding the last transition of B_{in} not including the stable start state) must also include the start state of B_{in} if B_{in} enables a 1–0 transition of an output, for otherwise one such cube may glitch (0–1–0) and the glitch may propagate to the output (1–0–1–0 dynamic hazard).

[1]In a sequential network, feedback variables as well as primary inputs must be considered as inputs to the network.

The smallest cube that covers a 1–1 transition of an output is called *essential*. Similarly, a 1–0 transition of an output is covered by a *set* of essential cubes — the minimal set consists of n essential cubes iff n input transitions enable the 1–0 transition of output. A logic hazard is present in an on-set cover if an essential cube is excluded from it. The *essential cover* for a logic function is a set of essential cubes.

If a transition between layers (state burst) requires multiple state bit changes (see figure 2), the machine traverses intermediate layers before it settles down to the final stable state. In 3D machines, a *critical race* is present if the transient states during a layer transition have different next-states from the final stable state (see figure 2). We insure that the machine is free of critical races by encoding layers such that no input or output burst intersects the transient states of layer transitions and by forcing all transient states during a layer transition to have the same next-states as the final stable state of the transition.

It has been assumed up to this point that output changes are not fed back until an input burst is assimilated by the machine; likewise, no state variable changes are fed back until the preceding output burst is absorbed. However, if feedback delays are short, there may be situations in which one or more fed-back outputs (at the network input) change before an enabled output transition fires. The hazard that arises due to the race between the arrivals of input transitions and output (or state variable) transitions at the network input is called the *essential hazard*. Essential hazards can be circumvented by inserting sufficient delays in the feedback paths. However, it is desirable to minimize feedback delays since the delays in the feedback paths impose an additional constraint on when the next set of primary input transitions can arrive at the 3D machine inputs without causing circuit malfunctions. In 3D machines, We minimize the feedback delays with a simple set of one-sided timing constraints [16].

3 Synthesis Procedure

The synthesis procedure consists of the following three steps:

1. A 3D next-state table is constructed from the burst-mode specification.
2. A layer diagram, which represents connectivities and encoding restrictions amongst the layers, is generated; a critical-race-free layer encoding is performed.
3. A set of on-set and off-set covers as well constraints for logic minimization is formed for each output and state variable; logic minimization is carried out.

3.1 Next-State Table Construction

Definition 1 *Let S_{IO} be the set of symbols representing the input-output states reachable by executing the burst-mode specification, and S_O be the set of output symbols. A burst-mode FSM specification is said to have the* **unique next-state code** *(UNC) property iff there exists a next-state function which maps S_{IO} to S_O.*

A burst-mode specification has the UNC property iff the next-state table can be built in one layer without conflicts — no additional state variable is required. The UNC property, however, is

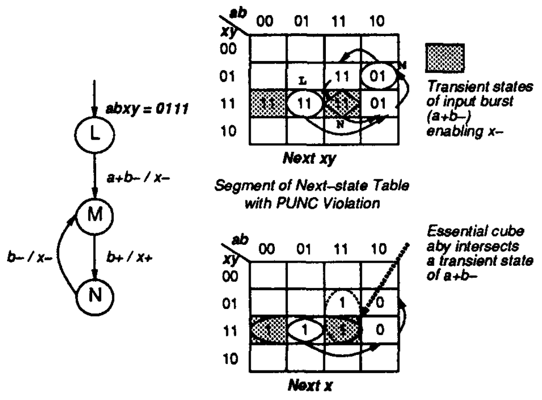

abxy = 0111
L
a+b- / x-
M
b- / x-
b+ / x+
N
ab
xy
00 01 11 10
Next xy
Segment of Next-state Table with PUNC Violation
Transient states of input burst (a+b-) enabling x-
Essential cube aby intersects a transient state of a+b-
Next x

Figure 3: PUNC Violation.

not sufficient to guarantee that a hazard-free "one layer" implementation of the specification exists. Suppose an essential cube of an on-set cover of an output, say x, intersects a transient state of an input burst B_{in} enabling a 1–0 transition of x. If the essential cube is not expanded to "include" the start state of B_{in} in the final logic equation, then the implementation has a dynamic logic hazard. The *top* next-state table segment of figure 3 does not violate UNC; nevertheless, if the essential cube aby, which intersects a transient state ($abxy = 1111$) of the input burst (a^+b^-), is not expanded to include the start state ($abxy = 0111$) of the input burst, a dynamic logic hazard is present in the implementation of x (during the input burst a^+b^-, the term aby may glitch (0–1–0), and this glitch may propagate to the output). This observation leads to the notion of the *proper unique next-state code* (PUNC) property — the specification with the UNC property has the PUNC property iff every essential cube of an output, that intersects a *transient state* of the input burst enabling a 1–0 transition of the output, also intersects the *start state* of the input burst. The *essential layer cover* for a layer L of the next-state table for a logic function F includes all the essential cubes in the layer L and no others.

Definition 2 *A layer L of the next-state table is said to have the* **proper unique next-state code** *(PUNC) iff the essential layer covers for the outputs (state variables) do not include any cube that intersects the transient states of a burst B enabling a $1-0$ transition of an output (state variable) but does not include the start state of B.*

In general, burst-mode specifications do not satisfy the PUNC property. We overcome this difficulty by building multiple layers of the next-state table. Each layer contains the next-states of the "path" traversed from one specification state to another via zero or more specification states and each satisfies the PUNC property.

We build a layer of the next-state table by assigning a next-state to each reachable state. Reachable states are traversed by "executing" the burst mode state diagram (in depth-first search manner). A node of the state diagram corresponds to a *specification-state*. Whenever a PUNC violation is detected[2], we back up to the last specification-state and start building a new layer from that node. When traversing a new branch of the state diagram, we start from the layer of the parent node. This process of traversing the state diagram and building layers of next-state table continues until all the nodes of the state diagram have been processed (see figure 4).

[2]There may be two different states in the original specification with identical input and output values. If the stable state reached at the conclusion of an output burst had been traversed (thus its next-state is already specified) and corresponds to a different specification-state, we need to back up to the last specification-state and start building a new layer even if the PUNC is not violated.

```
specification-state = initial specification-state;
layer = Initial layer;
repeat
  Execute input/output burst;
  if PUNC not violated then begin
    Enter next-states of the path traversed
      in the current layer of the next-state table;
    if next specification-state not processed then
      specification-state = next specification-state
    else begin
      Go back to the nearest ancestor
        with unprocessed children nodes;
      state = state of next unprocessed child node
    end;
    layer = layer of parent node
  end else begin
    Back up to the last specification-state;
    layer = new layer
  end
until every node is processed.
```

Figure 4: Next-State Table Construction Algorithm.

3.2 Layer Encoding

Once the layers of the next-state table are built, the layer diagram (see figure 6) is formed by grouping specification-states into layers by traversing the state diagram again. Whenever there is a transition (state burst) from layer A to layer B, an undirected edge is drawn between A and B. Let us denote the initial and final state of the state burst (layer transition from A to B) by s_i and s_f. The next-state table entries of the states with the same xy-position (see figure 2) as s_i and s_f are checked for possible conflicts. If the next-state of a state with the same xy-position as s_i and s_f has already been specified, then the layer containing that state, say C, is considered a potential cause for a conflict in assigning codes to A and B,[3] henceforth called a *potential-conflict layer*. The edge between A and B is then labelled with C.

Formally, the *layer diagram* is defined as an undirected graph. Each edge e_i is labelled with a (possibly empty) set of vertices. The vertices represent layers of the next-state table, the edges represent transitions between the layers, and the labels on the edges correspond to *potential-conflict layers* for the transitions edges represent.

The critical-race-free layer encoding is done using this graph. The objective of the layer encoding is to generate a critical-race-free layer assignment that requires a small number of state bits. It has been shown elsewhere [14, 4, 13] that a universal one-shot state encoding (using *state splitting*), with a Hamming distance of 1 between any two state codes, exists. However, the universal state encoding is very costly to implement for a large number of states; furthermore, requiring a Hamming distance of 1 between any two layers is unnecessary for our 3D implementation. Instead, we propose to use a simple heuristic.

Let the codes assigned to layers A and B be c_a and c_b respectively. The potential-conflict layer C, if assigned the code c_c, is said to *obstruct* the transition from A to B (or from B to A) iff

$$c_c + (c_a \oplus c_b) = c_a + (c_a \oplus c_b)$$

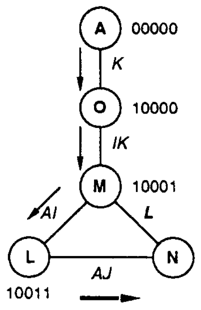
where $+$ and $\oplus$ denote *bitwise OR* and *XOR* respectively. For example, the potential-conflict layer C obstructs the transition from A ($c_a = 001$) to B ($c_b = 010$), if c_c is 000 or 011, but does not, if c_c is 100. Our goal is to encode the layers in such a manner that no layer obstructs the transitions (edges) on which it is labelled as a potential-conflict layer.

[3]Avoiding this conflict is simply a sufficient condition to avoid introducing dynamic logic hazards.

The heuristic layer assignment begins by trying to use state bits of length $\lceil \log_2 m \rceil$ bits where m is a number of layers. If the layer encoding using $\lceil \log_2 m \rceil$ bits fails, the heuristic retries using longer codes. The layer A, which contains the initial specification-state, is always assigned the code 0. Each layer is assigned the next available code (defined below) as the layer diagram is traversed in depth-first search manner starting from A. As we assign a code to layer l, we need to check whether the potential-conflict layers labelled on the edges between l and all of its neighbors with assigned codes obstruct the corresponding transitions. However, if a potential-conflict layer l_{pc} for an edge, say (l, l_j), has not been assigned a code, we need to postpone checking for possible conflicts (caused by l_{pc}) until a code is assigned to it. Meanwhile, we must record each edge on which l_{pc} is labelled as an *unresolved* potential-conflict for l_{pc}. Furthermore, we need to check whether l itself obstructs the transitions on which l is labelled as a potential-conflict layer.

The *next available code* c for layer l is a code with the shortest Hamming distance from the code of its predecessor node that satisfies the following conditions:

- c is not already assigned to another layer.
- The potential-conflict layers labelled on the edge between the layer l and its neighbor l_j do not obstruct the transition between l and l_j when c is assigned to l.
- Every unresolved potential-conflict for l becomes resolved — no edge, on which l is labelled as a potential-conflict layer, is obstructed by l — when c is assigned to l.



J not assigned before this segment

Attempting to assign N

	N	Hamming Distance from L	
1	10010	1	L obstructs (N,M)
2	10000	1	Assigned to O
3	10111	1	L obstructs (N,M)
4	11011	1	L obstructs (N,M)
5	00011	1	L obstructs (N,M)
6	10110	2	L obstructs (N,M)
7	10101	2	***Success***

Unresolved–Potential–Conflict

J	(L,N)

Figure 5: Code Assignment.

Figure 5 illustrates the code assignment on a segment of a layer diagram. Suppose layer J has not been traversed (thus not assigned a code). 10000, 10001 and 10011 are assigned to layers O, M and L. In attempting to encode layer N, we find that layer L obstructs the edge (N, M) if 10010, 10111, 11011, 00011 or 10110 is assigned to N. The code for layer N with the minimum Hamming distance from layer L that does not induce any conflict is 10101. Since a potential-conflict layer J for the edge (L, N) has not been assigned yet, we must record the fact that the edge (L, N) must be checked when a code is assigned to layer J.

If a code that meets the conditions above is not available when assigning a layer, the "code space" is enlarged by adding a code bit, and the entire encoding process is repeated. It is theoretically possible that adding code bits alone is not enough to guarantee that a critical-race-free layer assignment exists. Note that we can always fix the problem by adding redundant layers (state splitting). Nevertheless, in our extensive experiments, we have not encountered any such example; thus, our current implementation does not have a provision for the state splitting.

3.3 Logic Minimization

Definition 3 *A* **privileged cube** *is an essential cube that partially covers a burst enabling a* $1-0$ *transition of an output or a state variable. The* **privileged pair** (p, s) *is an ordered pair of a privileged cube* p *and the start state* s *of the burst enabling a* $1-0$ *transition of an output or a state variable, which* p *partially covers.*

Logic minimization is performed using exact algorithms for hazard-free logic, implemented in an automated logic minimizer [11]. This hazard-free logic minimizer, using a variation of Quine-McCluskey algorithm, attempts to find an optimum cover of *essential cubes* using *logical prime implicants*, the implicants that do not illegally intersect *privileged cubes*. Essential cubes, off-set cubes and privileged pairs are generated by our *3D synthesis tool*, and the prime implicants are produced by *espresso*.

4 Experimental Results

The synthesis procedure is completely automated (coded in C). Numerous experiments have shown that the synthesis tool produces results that are efficient in terms of both the area and the latency. The *latency* is a delay from the last input transition of an input burst to the last transition of the resultant output burst. Another useful measure is the minimum delay from the last input transition of an input burst to the first input transition of the next input burst without causing circuit malfunction, called the *cycle time*. Experimental results are shown in table 2. The latencies and the cycle times are evaluated using a 0.8μm CMOS standard cell library, developed for the Verilog simulator by the Torch group at Stanford University [6]. The library cells were characterized using the SPICE simulator under military worst-case conditions (4.5V power supply, 125°C) and derated for the nominal case (5V, 25°C).

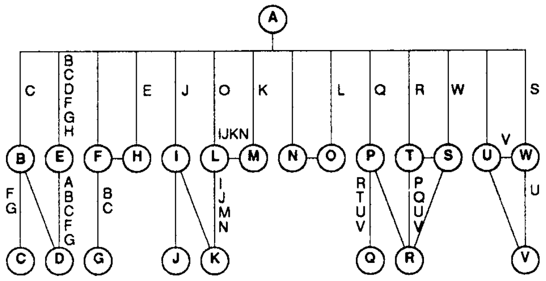


Figure 6: PSCSI Layer Diagram.

Layer	Code	Layer	Code	Layer	Code
A	00000	I	01000	Q	01011
B	00001	J	01101	R	01110
C	00011	K	10011	S	01100
D	00110	L	10001	T	01010
E	00010	M	10000	U	10110
F	00111	N	10100	V	11000
G	01111	O	00101	W	10010
H	00100	P	01001		

Table 1: Critical-Race-Free Layer Assignment of PSCSI.

We use a large specification called the *Pipelined SCSI Bus Controller (Asynchronous Data Transfer Protocol)* (similar to the one presented in [12]) to demonstrate the effectiveness of the 3D implementation and the synthesis procedure. The Asynchronous Data Transfer Protocol of the Pipelined SCSI Bus Controller is specified in 45 original states and 62 transitions; 10 primary inputs and 5 primary outputs are used. The 3D synthesis tool transforms the burst mode specification into the next-state table, derives a layer diagram (see figure 6), performs a critical-race-free layer assignment (see table 1), and generates essential covers, off-set covers and privileged pair sets for outputs and state variables.

	Specification				Implementation						
	States / Transitions		Primary		State	Product Terms		Literals			Cycle
			In	Out	Vars	*Output*	*Total*	*Output*	*Total*	Latency	Time
chu-ad-opt	4	4	3	3	0	4	4	11	11	1.2ns	1.2ns
vanbek-ad-opt	3	3	3	3	0	4	4	9	9	1.3ns	1.3ns
dme	8	10	3	3	2	6	11	18	29	2.0ns	3.1ns
dme-fast	8	10	3	3	2	7	12	19	29	1.7ns	2.9ns
alloc-outbound	8	9	4	3	2	6	12	16	27	1.8ns	3.0ns
mp-forward-pkt	4	4	3	4	0	6	6	14	14	1.4ns	1.4ns
nak-pa	6	6	4	5	1	7	10	12	17	1.7ns	2.5ns
pe-send-ifc	11	14	5	3	2	15	21	45	60	2.3ns	3.7ns
rcv-setup	6	8	3	2	0	3	3	8	8	1.4ns	1.4ns
sbuf-read-ctl	7	8	3	3	1	5	8	12	17	1.5ns	2.6ns
sbuf-send-ctl	8	9	3	3	2	9	14	21	32	2.1ns	3.3ns
sendr-done	3	3	2	1	1	1	4	3	8	1.0ns	2.4ns
sic-example	6	12	2	1	1	2	6	6	13	1.5ns	2.5ns
dram-controller	12	14	7	6	1	17	20	40	46	2.2ns	2.2ns
scsi-tsend-bm	11	13	5	4	2	19	27	38	58	2.3ns	3.8ns
scsi-trcv-bm	10	12	5	4	2	19	24	40	55	2.3ns	3.4ns
scsi-isend-bm	10	12	5	4	2	20	25	47	62	2.5ns	3.9ns
scsi-tsend-csm	10	11	5	4	2	20	24	34	44	2.2ns	3.3ns
scsi-trcv-csm	8	9	5	4	2	18	23	30	42	2.3ns	3.6ns
scsi-isend-csm	8	9	5	4	2	19	24	30	42	1.9ns	3.4ns
pscsi-isend	9	11	4	3	3	15	28	44	80	2.9ns	4.4ns
pscsi-ircv	6	7	4	3	2	9	14	19	31	1.7ns	3.2ns
pscsi-tsend	10	12	4	3	3	13	26	34	70	2.2ns	4.3ns
pscsi-trcv	6	7	4	3	1	12	14	21	25	2.2ns	2.6ns
pscsi-tsend-bm	10	12	4	4	3	11	23	29	60	2.0ns	3.7ns
pscsi-trcv-bm	7	9	4	4	2	15	21	32	47	2.0ns	3.8ns
pscsi	45	62	10	5	5	51	108	162	378	3.3ns	6.1ns

Table 2: Experimental Results.

The logic minimization is performed by the exact logic minimizer described in the previous section.

In the future research, we plan to extend the burst mode specifications to allow *don't care* inputs in the input bursts and provide the capability to handle the input choices based on "level-sensitive" conditional signals.

5 Acknowledgement

The authors would like to thank Steve Nowick of Stanford University for generously providing the logic minimizer and many helpful discussions.

References

[1] Jon G. Bredeson. On multiple input change hazard-free combinational switching circuits without feedback. In *14th Annual Symposium on Switching Theory, Iowa City, Iowa*, October 1973.

[2] Jon G. Bredeson and Paul T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. *Information and Control*, 20:114–224, 1972.

[3] T.-A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, 1987.

[4] A. D. Friedman, R. L. Graham, and J. D. Ullman. Universal single transition time asynchronous state assignments. *IEEE TOC*, C-18(6):541–547, 1969.

[5] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *DAC-91*.

[6] J. Maneatis and D. Ramsey, 1992. Private communication.

[7] T. H. Meng. *Synchronization Design for Digital Systems*. Kluwer Academic, 1990.

[8] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. CSP, Inc., 1985.

[9] C.W. Moon, P.R. Stephan, and R.K. Brayton. Specification, synthesis, and verification of hazard-free asynchronous circuits. In *ICCAD-91*.

[10] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *ICCD-91*.

[11] S. M. Nowick and D. L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *ICCAD-92*.

[12] S. M. Nowick, K. Y. Yun and D. L. Dill. Practical asynchronous controller design. In *ICCD-92*.

[13] Gabriële Saucier. Encoding of asynchronous sequential networks. *IEEE TEC*, EC-16(6):365–369, 1967.

[14] S. H. Unger. *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969.

[15] P. Vanbekbergen, F. Catthoor, G. Goossens and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *ICCAD-90*.

[16] K. Y. Yun, D. L. Dill and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *ICCD-92*.