Unifying Synchronous/Asynchronous State Machine Synthesis *

Kenneth Y. Yun David L. Dill

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Abstract

We present a design style and synthesis algorithm that encompasses both asynchronous and synchronous state machines. Our proposed design style not only supports generalized "burst-mode" multiple-input change asynchronous designs [21], but also allows the automatic synthesis of *any synchronous Moore machine* using only basic gates (and no state-holding elements). Moreover, the synthesis method covers many circuit styles in the range between burst-mode and fully synchronous. We can easily specify and synthesize sequential circuits which change state on both rising and falling clock edges, have multiple-phase clocks, etc., and mixed synchronous/asynchronous designs, subject only to setup and hold-time constraints. To demonstrate the effectiveness of the design style and the synthesis tool, we present a modified version of a previously published large practical controller design — the SCSI data transfer controller [14] redesigned to improve performance and to eliminate preprocessing circuit for converting "level-sensitive" signals to "edge-sensitive" signals, often a cumbersome manual design process, by interfacing directly with "level-sensitive" signals.

1 Introduction

Over the last few years, many new design styles and synthesis methods for asynchronous control and interface circuits have been proposed [2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 17, 18, 20, 21]. The high degree of interest in this work stems from designer concerns such as dealing with clock distribution, obtaining high performance, and dealing with interfaces between synchronous and asynchronous devices.

There are three loosely-defined categories of asynchronous design styles and synthesis methods available today: transformations from HDL descriptions [1, 3, 7, 11], synthesis from signal transition graphs (STG) and state graphs (SG) [2, 4, 5, 8, 10, 17, 18], and multiple-input change (burst-mode) asynchronous state machines [12, 20, 21].

All of these design styles are highly constrained, because strict requirements are necessary for hazard-free synthesis. It remains to be seen that actual applications fall naturally within these constraints. When interface applications have been designed using these techniques, it has been clear that loosening or changing the constraints would simplify the design and make the results more efficient.

In a recent paper [21], we identified some additional features of asynchronous specifications which would make interface design simpler, namely *conditionals* and *directed don't cares*. Conditionals allow control flow to depend on the logical value of an input signal, and directed don't cares allow an input signal to change (monotonically) concurrently with output signals. Both of these additions are significant extensions over existing design styles. We also described an automatic synthesis method for the new specifications.

* This work was supported by the Semiconductor Research Corporation, Contract no. 92-DJ-205, and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.

The addition of conditionals and directed-don't-cares moved our existing asynchronous design style in the direction of synchronous circuits: In the limit case where all but one signal is a conditional signal, the remaining signal controls when state changes occur, and when the conditional signals are sampled. In other words, the remaining signal is a clock, and the circuit is synchronous almost.

There is only one reason why arbitrary synchronous circuits cannot be implemented in this framework: the conditional signals must change monotonically. In a true synchronous circuit, input signals can vary freely except during a setup/hold-time interval.

In this work, we describe how to eliminate the requirement of monotonicity in conditional signals from the synthesis method. Hence, the resulting design style not only supports generalized "burst-mode" multiple-input change fundamental-mode asynchronous designs, also allows the automatic synthesis of *any synchronous Moore machine*, in which the synchronous inputs are represented as conditional signals, and the clock is the only non-conditional signal. Moreover, the synthesis method covers many circuit styles in the range between burst-mode and fully synchronous. We can easily specify and synthesize sequential circuits which change state on both rising and falling clock edges, have multiple-phase clocks, etc., and mixed synchronous/asynchronous designs, subject only to setup and hold-time constraints.

The resulting circuits use only combinational logic with feedback – there are no explicit latches. In effect, on purely synchronous designs the method automatically synthesizes the latches at the gate level.

Although we believe that unifying synchronous and asynchronous design represents a breakthrough, the synthesis method is not competitive in its current form with existing synchronous synthesis methods for purely synchronous applications. However, it is highly efficient for asynchronous designs, and we believe it will be very attractive for applications that are "mostly" asynchronous, with some synchronous aspects. We demonstrate the last point using a modified version of a previously published large practical controller design — the SCSI data transfer controller [14] redesigned to improve performance and to eliminate the preprocessing circuit for converting "level-sensitive" signals to "edge-sensitive" signals, often a cumbersome manual design process, by interfacing directly with "level-sensitive" signals.

This paper is organized as follows: Section 2 describes the extended-burst-mode specification and the proposed change in the interpretation of the specification to handle synchronous designs. Section 3 describes the 3D synthesis method: in particular, the combinational logic covering requirements in the presence of non-monotonically changing signals and their effects on the synthesis algorithm. Section 4 describes the SCSI controller specified in the extended-burst-mode and implemented in the 3D design style.

2 Specification Style

In this section, we describe the extended-burst-mode specification as defined in [21] and propose lifting some restrictions we imposed in [21] to handle purely synchronous design.

2.1 Notations and Terminology

Definition 1 *Signal transitions:*

- *s^+ denotes that s changes monotonically from 0 to 1 (if s initially 0) or remains 1 (if s initially 1); s^- denotes that s changes monotonically from 1 to 0 (if s initially 1) or remains 0 (if s initially 0). A signal transition s^+ or s^- is said to be* **terminating**.
- *$s^\#$ denotes that s remains 0, monotonically changes from 0 to 1, or remains 1; $s^\sim$ denotes that s remains 1, monotonically changes from 1 to 0, or remains 0. A signal transition $s^\#$ or $s^\sim$ is said to be a* **directed-don't-care**.
- *A terminating transition is* **compulsory** *iff the last preceding transition of the same signal is also terminating.*

The *terminating* transition of a signal means that the signal changes to the specified level, if it is not already at the specified level, and the *directed-don't-care* transition of a signal means that the signal *eventually* changes to the level specified by the next *terminating* transition. Note that the directed-don't-care does *not* mean that the signal can oscillate freely between 1 and 0 (as in a true don't care).

Definition 2 *Conditionals:*

- *$\langle s^+ \rangle$ and $\langle s^- \rangle$ denote* **conditional clauses** *"if input signal $s = 1$" and "if input signal $s = 0$" respectively.*
- *The signals specified in conditional clauses are said to be* **conditional signals**.

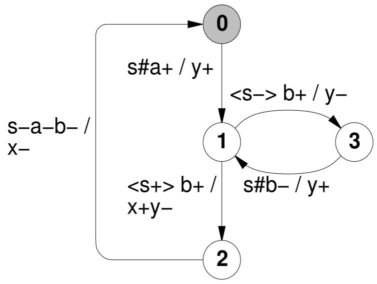


Figure 1: Extended-burst-mode Specification.

2.2 Extended-burst-mode Specification

An *extended-burst-mode* asynchronous finite state machine [21] is specified by a state diagram (see figure 1), which consists of a finite number of states, a set of labelled arcs connecting pairs of states, and a start state. Each arc is labelled with two sets of possible signal transitions: an input burst and an output burst. An *output burst* is a set of output transitions, and an *input burst* is a non-empty set of input transitions (terminating or directed-don't-care), at least one of which must be a *compulsory* transition. A *conditional input burst* is an input burst restricted by a *conditional clause*: i.e.,

- $\langle s^+ \rangle a^+ / x^+$ denotes "if $s = 1$ when a^+ fires, then x^+ is enabled to fire."
- $\langle s^- \rangle a^+ / y^-$ denotes "if $s = 0$ when a^+ fires, then y^- is enabled to fire."

In a given state, when all the conditional signals have the specified values (if the input burst is conditional) and when all the specified *terminating* transitions in the input burst have fired, the machine generates the corresponding output burst and moves to a new state. Specified transitions in the input burst may arrive in arbitrary order as long as the conditional signals are *set up* (*environmental constraint*) before any compulsory transition in the input burst fires. The *setup* and *hold* time of conditional signals with respect to *sampling* transitions depend on the implementation.

There are several restrictions to extended-burst-mode specifications:

- If an input burst in a state is conditional, all other input bursts in the same state must also be conditional.
- The polarities of terminating transitions of a signal must alternate. For example, the firing sequences, $a^+ a^-$, $a^+ a^\sim a^-$ and $a^- a^\# a^\# a^+$, are valid, but $a^+ a^+$ is not.
- The set of possible *entry points into a state* (input and output values entering a state) from every predecessor state must be identical — the *unique-entry-set* requirement. For example, the set of entry points to S_1 from S_0 is $\{sabxy|01001, 11001\}$; the set of entry points to S_1 from S_3 is the same.
- The *distinguishability constraint* exists to disambiguate multiple input bursts from a single state. Let T_i^{min} be the set of *compulsory* transitions in the input burst T_i and T_i^{max} be the set of all possible transitions in T_i. The distinguishability constraint for the extended-burst-mode is: *for every pair of input bursts i and j from the same state, either the conditional clauses are mutually exclusive, or $T_i^{min} \not\subseteq T_j^{max}$.*

  For instance, the input bursts from S_0 in figure 2a are legal because s^+ and s^- are mutually exclusive. However, the input bursts from S_0 in figure 2c are illegal because the conditional clauses are not mutually exclusive and $\{b^+\} \subset \{a^+, b^+\}$. Likewise, the input bursts from S_0 in figure 2b are illegal because the set of all possible transitions for the unconditional input burst $a^+ b^\#$ is $\{a^+, b^+\}$ and $\{b^+\} \subset \{a^+, b^+\}$.

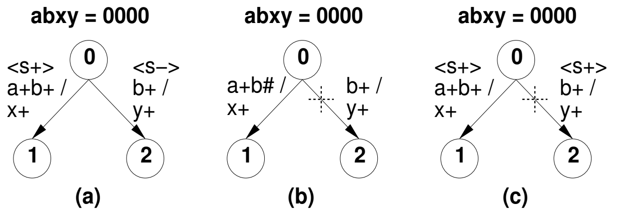


Figure 2: Distinguishability Constraints.

Difference from Previous Extended-burst-mode Specification

In [21], two constraints on the conditional signals were imposed:

- The conditional signals must change monotonically even outside of the setup/hold window (unlike data signals in synchronous circuits). The conditional signals are treated as event signals elsewhere in the specification.
- The number of conditional signals per conditional clause was restricted to one.

Here, we remove these restrictions. First, we allow *an arbitrary number of conditional signals per conditional clause.* Second, values of conditional signals are *undefined outside of setup and hold time windows of the sampling (compulsory) signals* — this implies that conditional signals *may not be used as event signals.*

3 3D Implementation

3.1 Overview

A 3D asynchronous finite state machine is a 4-tuple (X, Y, Z, δ) where X is a set of primary input symbols, Y a set of primary output symbols, Z a (possibly empty) set of internal state variable symbols, and $\delta : X \times Y \times Z \rightarrow Y \times Z$ 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 extended-burst-mode specification is obtained from the 3-dimensional function map called the *next-state table*, a 3-dimensional tabular representation of δ. The next-state of every reachable state must be specified; the remaining entries of the next-state table are don't-cares.

The operation of the 3D state machine is similar to a Mealy-mode synchronous state machine. A machine cycle consists of 3 *phases*: an input burst followed by an output burst followed by a state burst (or an input burst followed by a state burst followed by an output burst). After completion of the previous output/state burst, the machine waits for an input burst to occur. The input burst may be conditional or unconditional. The conditional signals must be set up before the *first* compulsory transition arrives. When the last terminating transition of the input burst arrives, an output burst (a state burst) takes place. The state burst (output burst) immediately follows the output burst (state burst), completing the 3-phase machine cycle. The next set of compulsory transitions may not arrive until the machine is stabilized (*fundamental-mode environmental constraint*). The conditional signals must remain stable (*hold time requirement*) until the machine is stabilized as well.

3.2 Synthesis Method

The synthesis procedure described below follows the same steps presented in [19] with some modifications in the next-state table generation step. We describe how the combinational logic covering requirements in the presence of non-monotonically changing conditional signals alter the next-state table construction algorithm.

The synthesis procedure begins with building the *next-state table* (see figure 6) by assigning a next-state to each reachable state. If the specification does not have a *unique next-state code* for each reachable state, new *layers* of the next-state table are added so that the final construction has the unique next-state codes. Note that the next-states are assigned so that every output/state variable transition is monotonic, i.e., the presence of any *function hazard* is precluded.

Once the next-state table is built, the codes are assigned to the layers of the table using a Tracey-like encoding heuristic [16]. In 3D machines, a critical race is present if the transient states during a layer transition have different next-state codes from the final state of the transition. The layer encoding heuristic insures that the machine is free of critical races by assigning codes to the source and destination layers of each layer transition such that the next-states of the transient states traversed during layer transitions have not been specified (thus can be assigned the same next-state as the final stable states).

The next step is to eliminate sequential hazards. In 3D machines, the outputs (state variables) change in response to the input bursts and remain constant while the fed-back outputs and state variables change. Likewise, the state variables (outputs) change due to the output bursts (state burst) and remain constant while the fed-back state variables (outputs) change. Assuming sufficient delays in the feedback paths for outputs and state variables, we can disregard the interactions between inputs and feedback variables (i.e., there are no *essential hazards* [20]).

The following are the timing requirements imposed by the synthesis method to guarantee correctness of the implementation.

- *fundamental-mode environmental constraint* : no new input burst may arrive until the machine is stabilized;
- *feedback delay requirement*: feedback variable transitions are not fed-back until the output/state burst is complete;
- *setup time requirement*: all conditional signals must be stable before any compulsory (sampling) transitions arrive.

Assuming these timing constraints are met, we can analyze the hazards in the combinational circuit that results from cutting feedback paths. We can then view each burst (input, output or state) as a *multiple-input change* to the output (state variable) combinational logic.

3.2.1 Requirements for Combinational Logic Synthesis

The presence of freely-varying signals makes hazard-free combinational logic synthesis difficult. Here we develop a set of hazard-free covering requirements for the 2-level AND-OR implementation of a logic function during a *multiple-input change* with some inputs undergoing non-monotonic transitions. The hazard-free combinational logic synthesis for multiple *monotonic* input changes is described in [13, 21]. The *new* results presented here are simple extensions of [13]. We apply these results to the 3D machine combinational logic synthesis.

Delay Model

The *bounded wire delay model* best represents the Huffman-mode asynchronous state machines, such as the 3D machines, in the current generation of VLSI technology (ever-decreasing feature size and non-trivial wire delays compared to gate delays). However, it requires complex timing analysis to detect the presence of hazards. We use the conservative *unbounded wire delay model* — any connection between a gate output and a gate input can have unbounded but finite delay — for the combinational logic to simplify timing analysis.

Combinational Logic Hazards in the Presence of Undirected-don't-care

We consider the combinational logic hazards *when at least one input change is monotonic and compulsory and the rest can be monotonic but non-compulsory or non-monotonic (don't-care).*

A *generalized transition cube* is a cube with a *start-cube* and an *end-cube*. The generalized transition cube $[A, B]$ contains all minterms that can be reached during a transition from any point in start-cube A to any point in end-cube B. The start-cube A (end-cube B) is a maximal subcube of $[A, B]$ with all compulsory transition signals at pre-firing (post-firing) levels. The *start-subcube* (*end-subcube*) A' (B') is a maximal subcube of A (B) with all directed-don't-cares and non-compulsory terminating transitions at pre-firing (post-firing) levels. An *open generalized transition cube* $[A, B)$ is $[A, B] - B$. A *multiple-input change* from A to B includes all compulsory input transitions and may contain some non-compulsory transitions. Figure 3a illustrates a transition cube $[A, B]$ with start-cube A, end-cube B, compulsory input transitions a^+ and b^+, and undirected-don't-care s^*. In figure 3b, s^* is replaced with $s^\#$.

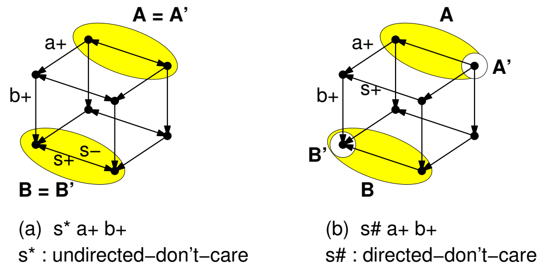


Figure 3: Transition Cube $[A, B]$.

Let $[A, B]$ be a transition cube that describes a *function-hazard-free* transition (the output changes monotonically during multiple-input change) from a set of input states A (start-cube) to B (end-cube) for a combinational logic function f. Assume that a cover C for f is implemented in a 2-level AND-OR.

Lemma 1 *If f has a $0-0$ transition in $[A, B]$, the implementation is free of logic hazards.*

Lemma 2 *If f has a $1-1$ transition in $[A, B]$, the implementation is free of logic hazards iff $[A, B]$ is contained in cover C.*

Lemma 3 *If f has a $0-1$ ($1-0$) transition in $[A, B]$, the implementation is free of logic hazards iff no cube c in C intersects $[A, B]$ unless c also contains B' (A').*

From Lemmas 1- 3, a set of hazard-free on-set covering requirements for the output (state variable) transition enabled by a burst is derived.

1. For $1-1$ transition:

   $[A, B]$ must be covered by a single cube.

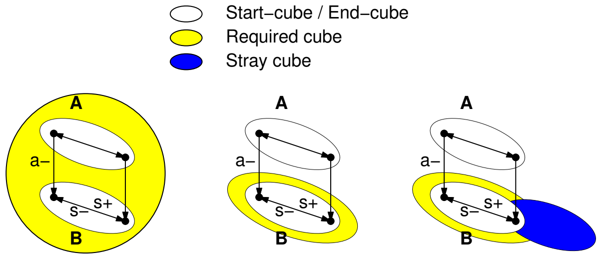
2. For $0-1$ transition:

   The minterms in $[A, B)$ belong to off-set, and B must be covered by a single on-set cube. Any on-set cube that intersects B must also contain the end-subcube B'.

3. For $1-0$ transition:

   Each maximal subcube of $[A, B)$ must be covered by a single on-set cube, and the minterms in B belong to off-set. Any on-set cube that intersects $[A, B)$ must also contain the start-subcube A'.

Once the required on-set cubes for each output and state variable function during each burst are found and it is determined that no required cube violates the covering requirement 2 or 3, we simply OR the product terms corresponding to those required cubes to get a hazard-free cover. Hazard-free covers for $1-1$ and $0-1$ transitions are illustrated in figure 4ab. Figure 4c shows a violation of Lemma 3.



(a) 1–1 transition (no hazard)   (b) 0–1 transition (no hazard)   (c) 0–1 transition (hazard)

Figure 4: Covering Requirements for Transition Cube $[A, B]$.

3.2.2 Effects of Undirected-don't-cares on Next-state Table Construction

The 3D synthesis algorithm for the specifications without non-monotonically changing conditional signals is follows: The algorithm builds the *next-state table* by assigning a next-state to each reachable state. If the specification does not have the *unique next-state code* for each reachable state or a violation of the covering requirement 2 or 3 is detected, new layers of the next-state table are added so that the final construction has the *proper unique next-state code* (PUNC) property [19] (every specified entry in the next-state table *after state encoding* has the unique next-state code and no required cube violates the covering requirement 2 or 3).

In [19, 21], we showed that it is always possible to find a hazard-free cover for every output and state variable function if the next-state table construction (after state encoding) satisfies the *proper unique next-state coding* property. Furthermore, in [21], we showed that it is always possible to construct a next-state table which has the PUNC property for any extended-burst-mode specification with monotonically changing conditional signals (allowed to change *at most once* during a transition from a specification-state to the next specification-state).

Now we must examine whether it is still possible to construct the next-state table that has the PUNC property when we allow conditional signals to change freely outside of setup and hold time windows of the sampling (compulsory) signals. We begin by analyzing an example.

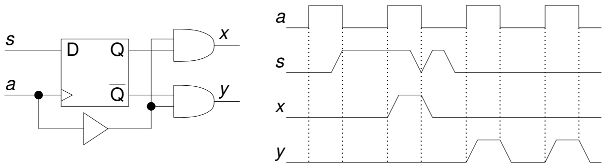


Figure 5: Example (Synchronous Implementation).

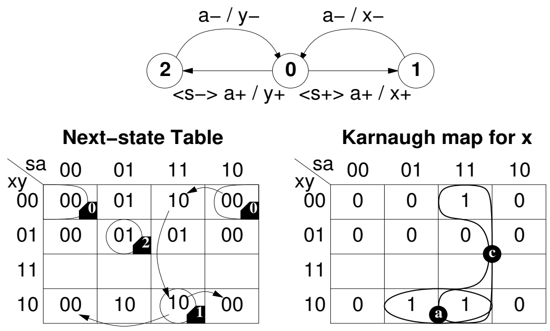


Figure 6: Example (State Diagram and Next-state Table).

Example

Figure 6 shows a specification of a circuit which works as follows and figure 5 shows one possible synchronous implementation and the timing diagram.

- If the mode bit s sampled at the rising edge of the clock a is 1, the output x follows the clock for that cycle and the output y remains 0. Otherwise, y follows the clock and x remains 0.

Consider the input burst a^- in S_1, which enables the output transition x^-. The covering requirement 3 states that no cube may intersect the transition cube $[A, B]$ (A = x110 and B = x010) unless it also contains A' (A). However, cube c (required to cover the output burst x^+ in S_0) shown in figure 6 intersects the cube a (part of the transition cube $[A, B]$), but it cannot be expanded to cover A — there is a dynamic hazard.

Solution

Our solution to avoid this dynamic hazard that results from violating the covering requirement 3 is to add a new layer (state) and to move to it (state burst) before enabling outputs to change *if the next input burst enables* $1-0$ *transition of an output.*

Figure 7 illustrates our solution. If $s = 1$ when a^+ fires, we move to a new layer (state burst p^+) before enabling x^+ to fire. Thus the next x entry for $saxy$ = 1100 (in the $p = 0$ part of the table) is specified to be 0. In the $p = 1$ part of the table, the next x entry for $saxy$ = 11x0 is 1. When output x stabilizes to 1, the machine is in S_1. The next x for $saxy$ = x110 ($p = 1$) is specified to be 1 so that output x remains unchanged until the compulsory transition a^- fires. In the new layer, we specify the next-states of the output burst *as if* s were allowed to change (possible because we are in a new layer) although the environment is not allowed to change s until x is stable (*hold time requirement*). Thus the start-cube of this output burst is $psaxy$ = 1x100, and the end-cube $psaxy$ = 1x110. This new required cube ($psaxy$ = 1x1x0) for the output burst x^+ now contains the *start-cube* of the next input burst a^- — no violation of covering requirements.

Now let's examine the required cubes for state variable p we added. We require one cube ($psaxy$ = x1100) to cover the state burst p^+ enabled by the conditional input burst $\langle s^+ \rangle a^+$ and another cube ($psaxy$ = 1x1x0) to cover the output burst x^+. Since the required cube ($psaxy =$ x1100) does not intersect the start cube of the next input burst a^-, the covering requirements are not violated.

Figure 7 shows our implementation which require 42 CMOS transistors (10.5 equivalent-gates). The synchronous implementation in figure 5 requires 10 equivalent-gates (7 for D-FF and 3 for 2 ANDs).

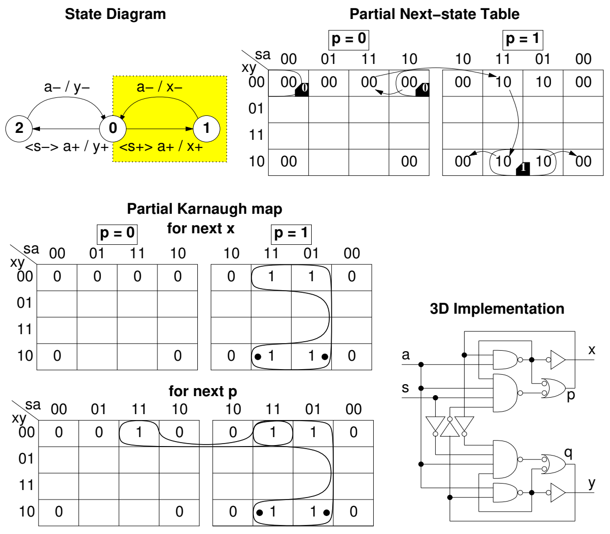


Figure 7: Example (Partial State Table with PUNC).

3.2.3 Next State Table Construction Algorithm

The algorithm builds the next-state table by assigning a next-state to each reachable state. Reachable states are determined by traversing the extended-burst-mode state diagram depth-first and, at each specification-state, "executing" the specified input/output bursts. At each specification-state, if the next-states of the reachable states do not induce a PUNC violation, the current layer of the next-state table is updated with new entries. However, if a potential PUNC violation is detected, the current specification-state is assigned to a new layer, and the next-states of the reachable states are recorded in the new layer of the next-state table (If the path leading to the current specification-state is merged with a path to another specification-state that had already been traversed, the algorithm backtracks to the specification-state preceding the merging and assigns that specification-state to a new layer). Moreover, *if the input burst following a conditional burst enables a* $1-0$ *transition of an output, the algorithm inserts a state burst immediately following the conditional input burst and specifies the next-states of the output burst in the new layer. If the output burst is empty, the algorithm adds a "dummy" output burst.*

Using this algorithm, it is always possible to construct a next-state table which has the PUNC property for any extended-burst-mode specification, including ones with *non-monotonically changing* conditional signals. Thus it is *always possible to find a hazard-free cover* for every output and state variable function.

4 Example (SCSI Controller)

In this section, we describe a modified version of the SCSI data transfer controller presented in [14]. The purpose of this exercise is to demonstrate that the extended-burst-mode specification indeed provides more powerful mechanism to specify the controller behavior when dealing with existing synchronous interfaces and improves performance by allowing concurrent input/output transitions.

4.1 Overview

The SCSI data transfer controller communicates with two interfaces: the SCSI device's local DMA bus and the SCSI bus. The controller regulates the flow of data between two buses. Our implementation (see figure 8) assumes that the DMA bus is a 680x0-type bus controlled by an M68450 compatible DMA controller.

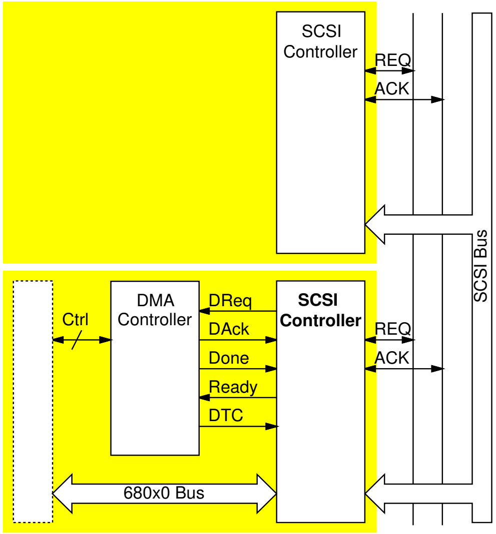


Figure 8: A Simple Configuration of SCSI Bus.

A SCSI device is configured in one of four operating modes: *Target-Send*, *Target-Receive*, *Initiator-Send*, and *Initiator-Receive*. The initiator originates the data transfer operation by requesting the target to begin a handshaking protocol. The sender moves data from the local bus to the SCSI bus; the receiver moves data from the SCSI bus to the local bus. M68450 compatible DMA controllers can access the 680x0 bus in two modes: *cycle steal* and *burst*. In *burst* mode, the DMA controller maintains the control of the bus until a block transfer is complete whereas, in *cycle steal* mode, the DMA controller relinquishes the bus after each transfer and acquires the bus through arbitration for each data transfer. Our implementation supports all eight data transfer modes.

4.2 Implementation

Our new implementation has two major improvements, *directly attributed to the extension of the burst-mode specification*, over the one presented in [14].

- In [14], a special decoding circuitry which converts M68450 signals to the form that suited the original burst-mode specification style was needed. Our new implementation interfaces directly with M68450 signals eliminating the decoding circuitry because we can specify the desired behavior (distinguishing the last transfer from the preceding ones) directly using *conditional constructs* and *directed-don't-cares*.
- In [14], there were just two 8 bit registers for temporary data storage: one for input and the other for output, which meant that the local bus bandwidth was not fully utilized (up to 24 bits wasted for each transfer cycle), lowering the data transfer throughput. In our new implementation both the input and output data registers can hold up to 32 bits of data and function as FIFO buffers. The conditional burst construct is particularly useful in handling "loops" in the control flow, as required in reading/writing data from/to the FIFO. *Moreover, our new implementation works correctly whether the conditional signals used to terminate the loops are glitchy or not.* Our new work allows the designers to use predesigned (generally synchronous) macrocells that may produce glitches.

We describe below one of the operating modes (*Target-Send / burst mode*) in detail (see figures 9 and 10).

In the *Target-Send* mode, a data transfer operation begins when the SCSI controller makes a data transfer request to the DMA controller by asserting DReq. Upon detecting DReq asserted, the

DMA controller requests the bus mastership. Once the bus mastership is granted, the DMA controller notifies the SCSI controller that a transfer is to take place by asserting DAck (Note that, in the *burst* (or block) mode, DReq must remain asserted until the last transfer is acknowledged). The SCSI controller, in turn, asserts Ready, signalling the DMA controller that it is ready to receive data. The DMA controller asserts DTC (Data Transfer Complete) when the data is valid on the bus. The SCSI controller, latches the 32 bit data in the output data register using DTC and negates Ready. Note the synchronous nature of the DMA controller operation — it negates DTC and DAck one clock cycle after DTC is asserted and asserts DAck again, initiating the next transfer cycle (if DReq remains asserted), one clock cycle after DTC and DAck are negated without an explicit acknowledgment from the SCSI controller.

Once the data is latched in the output data register, the least significant byte is immediately driven onto the SCSI bus. Note that the SCSI bus protocol mandates that data must be valid on the bus before REQ is asserted and remain valid until ACK is asserted. The (target device) SCSI controller asserts REQ after DAck and DTC are negated, and the initiator acknowledges the fact that it has received the data by asserting ACK. The (target device) SCSI controller then shifts the next byte onto the SCSI bus, updates the transfer byte counter and negates REQ, and the initiator, in turn, negates ACK. The (target device) SCSI controller *samples* the output data register Empty flag to determine whether all 4 bytes have been transferred when ACK is negated (Note that the Empty flag is updated when ACK is asserted and sampled when ACK is negated). This 4 phase handshaking continues until all 4 bytes are transferred to the initiator. When the most significant byte transfer is acknowledged, the (target device) SCSI controller asserts Ready if DAck had been asserted.

Note that the Empty flag can be glitchy if it is a ripple-carry output of a synchronous counter. Our implementation will work correctly whether the Empty flag is glitchy (*non-monotonic conditional signal*) or not.

To terminate the DMA transfer, the DMA controller asserts Done and DAck simultaneously. The SCSI controller samples Done flag when DTC is asserted. Done, DAck and DTC are negated all at the same time. In [14], DAckLast and DAckNorm signals (served as DAck for the last transfer and for all the other ones respectively) were decoded from DAck and Done. In our new implementation, this decoding is unnecessary because the controller is capable of sampling the *level* conditional signals.

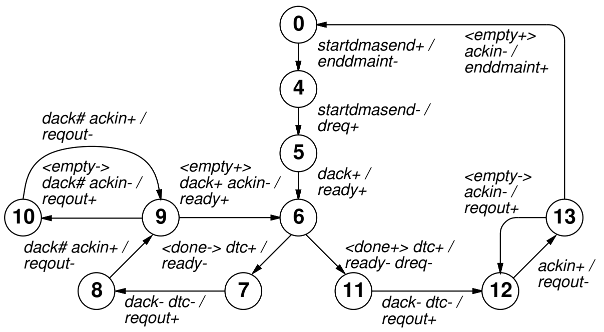


Figure 9: SCSI Target-Send / Burst Mode Specification.

4.3 Results

The SCSI data transfer controller, which supports all 8 operating modes, was specified in 74 states and 104 transitions. 11 primary inputs and 5 primary outputs were used; 6 state variables were added by the 3D synthesis tool. The two-level AND-OR implementation produced by the 3D tool had 784 literals. The two-level logic equation, subsequently, was mapped to the 0.8μm CMOS standard cell library, developed for the Verilog simulator by the Torch group at Stanford University [6], using the hazard-nonincreasing technology mapper [15]. 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).

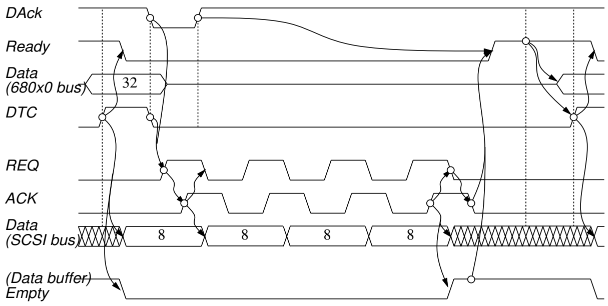


Figure 10: SCSI Target-Send / Burst Mode Timing.

The implementation has 710 equivalent-gates (2838 CMOS transistors) compared to 498.5 equivalent-gates for [14], which does not include the decoding circuitry. It has 2.9ns *latency*, the delay from the last input transition of an input burst to the last transition of the resultant output burst, and 4.5ns *cycle time*, the delay from the last input transition of an input burst to the first input transition of the next input burst.

References

[1] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *ICCAD-92*.

[2] P. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *ICCAD-92*.

[3] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *ICCAD-89*.

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

[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] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive VLSI circuits. In C. A. R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*, Addison-Wesley, 1990.

[8] Teresa H.-Y. Meng. *Synchronization Design for Digital Systems*. Kluwer Academic, 1990.

[9] 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.

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

[11] C. Myers and T. Meng. Synthesis of timed asynchronous circuits. In *ICCD-92*.

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

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

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

[15] P. Siegel, G. De Micheli, and D. Dill. Automatic technology mapping for generalized fundamental-mode asynchronous designs. In *DAC-93*.

[16] J. M. Tracey. Internal state assignment for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15(4):551-560, August 1966.

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

[18] C. Ykman-Couvreur, B. Lin, G. Goossens, and H. De Man. Synthesis and optimization of asynchronous controllers based on extended lock graph theory. In *EDAC-93*.

[19] Kenneth Y. Yun and David L. Dill. Automatic synthesis of 3D asynchronous state machines. In *ICCAD-92*.

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

[21] K. Y. Yun, D. L. Dill, and S. M. Nowick. Practical Generalizations of Asynchronous State Machines. In *EDAC-93*.