Note: This is a copy of the paper published as Chapter 3 of Asynchronous Digital Circuit Design. Springer-Verlag Workshops in Computing series, April 1995, pp. 104 – 151.

Synthesizing Asynchronous Circuits: Practice and Experience

Al Davis

Department of Computer Science, University of Utah
Salt Lake City, UT 84112, USA

Abstract

Asynchronous circuit design has been a topic of study for over 40 years. Only recently has this study turned to the practical issues of building useful CAD tool support that approaches the level of functionality available in synchronous VLSI CAD systems. The existence of these asynchronous CAD systems enables today's designers to construct asynchronous VLSI devices of significant size and scope. This chapter provides a reasonably detailed description of the process of developing a set of VLSI CAD tools which permit correct controller circuits to be synthesized from finite state machine style specifications. The tools were developed in two stages. The first stage resulted in a set of tools called **MEAT**, which were used to implement a 300,000 transistor multiprocessor routing chip called the Post Office. This *Post Office experience* provided the motivation for the second stage which resulted in the **STETSON** tool set which is integrated with the Mentor VLSI tool suite and was used to synthesize a low-power infrared transceiver chip called ABCS. The chapter presents a somewhat chronological view of the development of the CAD methods and provides examples of how well these methods worked in a practical design setting.

1 Historical Motivation and Introduction

Today's digital designers are faced with a large array of stringent constraints. Design cycle times are shortening at an alarming rate. Clearly this is a side-effect of an intensely competitive marketplace but the reality is that today's designers spend their time on significantly different activities than they did a decade ago. It is often the case that the *time budget* to go from the initial design concept to working silicon can be as short as six months even for chips containing hundreds of thousands of transistors. Even multi-million transistor processor chips must be operational in slightly over a year from the time the architecture is frozen. Additional design requirements such as aggressive performance, area, and power goals exacerbate the difficulty.

The ability of the digital design field to keep up with these growing pressures has relied heavily on the capability of automated CAD tools which remove much of the low-level drudgery of the design processes in existence even a few years ago. With very few exceptions, there is simply not enough time to worry about every small detail of the design and perform hand optimized custom layouts for the implementation. The result is that designers must rely heavily on synthesis tools to automatically generate layouts from some high level hardware description such as a Verilog or VHDL program. Designs are often specified initially at the *behavioral* level to permit validation of the design's functional behavior. Once the functional behavior is considered correct then the design is further refined into a structural specification which describes the implementation details. At this point, the structural code can then be compiled into the actual implementation such as the geometry or layout in the case of a single VLSI device.

Today's synchronous designers have a large number of CAD options to choose from. Companies such as Mentor, Cadence, Synopsis, etc. vend commercial CAD tools which integrate a fairly large number of tool options, many produced by third parties. On the other hand, the asynchronous circuit design community has had very little in the way of acceptable CAD tools to support their design activities. This is in part a side effect of a primarily academic and research oriented discipline. There are also numerous asynchronous design styles. Many of these were mentioned in Chapter 1, and the other chapters in this book provide a more in-depth look at several of them. Certain asynchronous design styles do permit use of *some* of the existing synchronous CAD tools.

In particular the data path components of certain asynchronous designs and their synchronous counterparts are not significantly different. The biggest difference is that the timing of synchronous circuits is based on a periodic global clock whereas asynchronous circuits rely on a particular set of *signaling protocols* to provide appropriate control for a sequence of events. While this difference is significant for controller circuits, the difference for datapath circuits in some design styles is relatively minor. The use of *model* delays to provide asynchronous completion signals is similar to the synchronous circuit controller waiting for the specific number of clock cycles that corresponds to the completion delay of the enabled datapath component. The completion time of a synchronous datapath component is determined by performing a timing analysis on the final implementation and then picking the smallest number of clock cycles which is still greater than or equal to the modeled delay. In the asynchronous case, the request is passed through a delay element (often just a string of inverters) and then passed back to the calling module as the acknowledge.

This view that asynchronous controllers are fundamentally different but that datapath circuits are similar to synchronous circuits led to a design style has been used since 1970 when the author was first exposed to asynchronous circuit design through a series of courses taught by Chuck Seitz at the University of Utah. The initial methods were based on those described by Unger [46]. Since that time, they have been used in the design of a number of large asynchronous computer systems such as DDM1 [9], several asynchronous VLSI devices such as the ISM [7], as well as a number of synchronous, or mixed synchronous and asynchronous designs. Given this background, it is clear why the design style which subsequently evolved and the tools which support this style is similar to the synchronous design domain.

Many of today's asynchronous practitioners will argue vehemently that asynchronous circuit design is fundamentally different from synchronous design and hence it is somewhat silly to try and exploit any commonality between the two regimes. The work presented here argues the opposite point of view. Namely, if asynchronous designs are to make a practical impact, they must be capable of co-existing with synchronous designs. Furthermore if asynchronous methods are to ever be considered as a mainstream option, then synchronous designers must be educated to the point where they can exercise this option. The claim is often made in asynchronous circuit workshops and symposia that the best way to move asynchronous designs into the mainstream is to educate enough untainted fresh minds that in time they will take over. The perspective here views this as extremely unrealistic. This approach assumes that by exploiting the methodological and CAD commonalities, it is possible to expand the designer's options to the point where asynchronous components become a reality in future commercial designs rather than being either a rare commercial anomaly or merely a research topic.

Prior to 1989, the design at hand was the primary focus and no effort was spent on creating appropriate CAD support. In 1987, a group of people at HP Laboratories started the design of a scalable parallel processing system called Mayfly [10]. The Mayfly communication fabric was based on an interconnected set of fully-adaptive router chips. The performance of this router chip, called the *Post Office* was clearly going to be critical to the performance of the overall system. Furthermore it was decided that in order to avoid a number of severe global clock management costs, the Post Office would be asynchronous. Ken Stevens was the main designer of the Post Office, but some of the Post Office subsystems were designed by Bill Coates. The other Mayfly project members (other than the author) were Robin Hodgson and Ian Robinson.

At some point, probably in the summer of 1989, it became clear that our complete lack of CAD tools to support the Post Office design was becoming a nightmare for two reasons:

1. Some controller designs required so many states and had so many inputs that it was improbable that even the most conscientious hand design would be correct.

2. The rate at which the Post Office design was progressing was so slow that it was clear that we would not meet the project schedule.

The initial call for help came from Ken Stevens. In what turned out to be a rash claim, I responded with a comment to the effect that in just three weeks I could do a *quick and dirty* hack for an asynchronous logic minimizer that would get Ken out of the current bind. The Common Lisp code for the asynchronous minimizer, called *amin*, did indeed work in three weeks. The first practical test case was for a controller circuit that had been crafted by hand and was operational as a test chip. Amin ran for an entire weekend. After millions of garbage collections it was still only half way to producing a solution. We had just learned what all real CAD developers already knew, that algorithmic complexity is always important.

This somewhat humorous beginning led to a much more serious effort to develop a working tool, called MEAT, that was used extensively in the design of the Post Office chip. MEAT worked well, but not well enough. This led to a broader and more serious effort that involved numerous people from HP and from Stanford University and culminated in a second more highly evolved tool for asynchronous controller design called STETSON. MEAT was a stand alone tool which still required that the designer do the layout of the physical implementation manually. STETSON used many of the MEAT ideas, in particular it was based on a restricted multiple input change, finite state machine specification which we named *burst-mode.* A significant difference is that STETSON is not a stand alone tool. STETSON is integrated into the commercial CAD suite provided by Mentor Graphics and generates its implementation based on a library of standard cells.

The remainder of this document describes the development of the MEAT and STETSON CAD methods and discusses how well the tools worked in practical system design.

2 The Post Office

The Post Office was designed to support inter-node communication for the Mayfly parallel processing system [8]. Mayfly processing elements (PEs) communicate with each other via messages. Messages are inherently variable in length, but the physical transport of those messages is organized as a sequence of fixed length packets. The role of the Post Office is to handle all of the physical delivery aspects of packet communication. The Mayfly topology was designed to be extensible and permits an unbounded number of PEs to be interconnected. This implies that the *physical extent* of a Mayfly system is not fixed. The unknown physical extent of the system poses serious problems when considering the implied constraints on an implementation strategy which uses a common global clock. Clock skew is a possible headache for any synchronous design style and the problem is magnified as technology progresses [1]. In the case of extensible systems such as Mayfly, where the total number of boards is unbounded, the synchronous choice becomes totally intractable. We therefore chose an asynchronous design style for the Post Office implementation.

Another critical design constraint was the need for a high performance implementation. Message passing performance would clearly be critical to the success of the Mayfly system. Many proponents argue that asynchronous circuits are inherently faster since they are controlled by locally adaptive timing rather than the usual global worst-case clock frequency constraints. While we believe that this claim has merit, we feel that in general it is misleading. Asynchronous circuits require more components to implement the same function. This may result in longer wires, increased area, and reduced performance. When compared to a very well tuned synchronous design, a functionally equivalent asynchronous implementation may actually run slightly slower. The need for speed heavily influenced our particular asynchronous design style.

There are a large number of rather different design styles used by the asynchronous design community. One partition of design styles can be based on the type of asynchronous circuit target: locally clocked [35, 17, 9], delay-insensitive [25, 3, 47, 30], or various forms of *single-* and *multiple-* input change circuits [46]. Yet another distinction could be made on the nature of the control specification: graph based [36, 28, 49, 5], programming language based [25, 3, 47, 2], or finite state machine based [35, 17]. For the finite state machine based styles, there is a further distinction that can be made based on the method by which state variables are assigned [19, 44]. The design style space is large and each design style has its own set of merits and demerits. It is worthwhile to note that virtually all of the design styles focus on the design of the control path of the circuit.

All asynchronous design styles are fundamentally concerned with the synthesis of hazard free circuits. The methods which produce delay-insensitive circuits, while not perfect [26], are the most tolerant of variations in device and wire delays. This tolerance improves the probability that a properly designed circuit will continue to function under variations in supply voltage, temperature, and process parameters. We chose to slightly expand the domain of timing assumptions which must remain valid to retain hazard free implementation since this permits higher performance implementations at the expense of reduced operational tolerance. We also chose to pursue a finite state machine based style for two reasons: 1) the finite state machine concept is a familiar one for hardware designers like ourselves, and 2) the graph and programming language based models that we knew about were too slow for our purposes.

Compiled implementations based on programming language like specifications [47, 2], while elegant and robust, suffer in performance because they are presently compiled into intermediate library modules rather than into optimized transistor networks. The module of greatest concern is the C-element. C-elements are common circuit modules in asynchronous circuits and eliminating them completely is unlikely. However it has been our experience over the past decade that C-elements are similar to the proverbial GOTO statements in programming languages, i.e. too many of them are indications of serious performance problems. C-elements are latches and as such are synchronization points. Too much synchronization reduces parallelism and hence performance degrades. The finite state machine based design style does not use C-elements, although C-elements are used sparingly in interface circuits such as arbiters.

The methods developed by Martin [25, 3] are an exception in that while the initial specification of the circuit is a CSP [18] based program, the compilation process is partly interactive and results in optimized pull-up and pull-down circuits for CMOS implementation. In retrospect, we should have paid closer attention to these methods but our finite state machine bias made us somewhat blind to the capabilities of these methods since the initial specification would have required a significant stylistic change. While it is only conjecture at this point, it might make sense to create a finite-state machine front end for

Martin's tools and see how well the resulting circuits fare against those generated by MEAT and or STETSON. However as designers, we were familiar with finite state machine methods and decided to maintain this flavor in our initial specifications. Part of this bias was purely stylistic, but in terms of transferring the results of our work to other commercial designers and CAD tools, it was a justifiable choice.

In order to achieve the necessary hazard free asynchronous finite state machine (**AFSM**) implementation, it is necessary to place constraints on how their inputs are allowed to change. The most common is the *single input change* or SIC constraint [46]. SIC circuits inherently require state transitions after each input variable transition. In cases where the next interesting behavior is in response to multiple input changes, the circuit response will be artificially slow, either due to too many state transitions or due to the external arbiters required to sequence the multiple inputs. *Multiple input change* or MIC circuit design methods have been developed [46, 6] but either required input restrictions or involved implementation techniques that were unsuitable for our purposes. As a result we developed a design style that we call **burst-mode** which permits a certain style of multiple input change. Our burst-mode implementation method does not require performance inhibiting local clock generation or flip-flops.

Prior to the Post Office design, anytime the output of an AFSM controller was in response to multiple input variable changes, we used an AND gate with selectively inverted inputs to account for the desired high-to-low or low-to-high input trajectories of the individual members of the input burst. The result was a new set of *conjunctively convolved* input variables to the real AFSM. The result was that the AFSM only needed to operate under SIC constraints and hence traditional methods could be used. As this practice evolved, we noticed an increasing number of instances where the resulting circuit was suboptimal in terms of logic complexity. Since traditional methods allowed us to minimize the logic of the SIC AFSM controller but the conjunctive *burst-mode* rendezvous logic (the AND gates) was not part of this minimization process. This provided the initial motivation to investigate a direct burst-mode AFSM design style that permitted optimization of the entire AFSM design. Another benefit of this method is that it works for **both** two and four cycle signaling protocols.

As our Post Office state machines became too complex for hand synthesis, we decided to create a tool kit that was capable of automatically synthesizing the transistor level circuits from burst-mode specifications. We called this tool kit **MEAT** (described in section 5). During the development of MEAT, we were fortunate to have Steve Nowick spend two summers with us. He incorporated David Dill's verifier [12] into the tool kit, and modified the verifier to accommodate our burst-mode timing model. Steve had considerable influence on our ideas. His locally clocked design style [35] is another outcome of these earlier interactions.

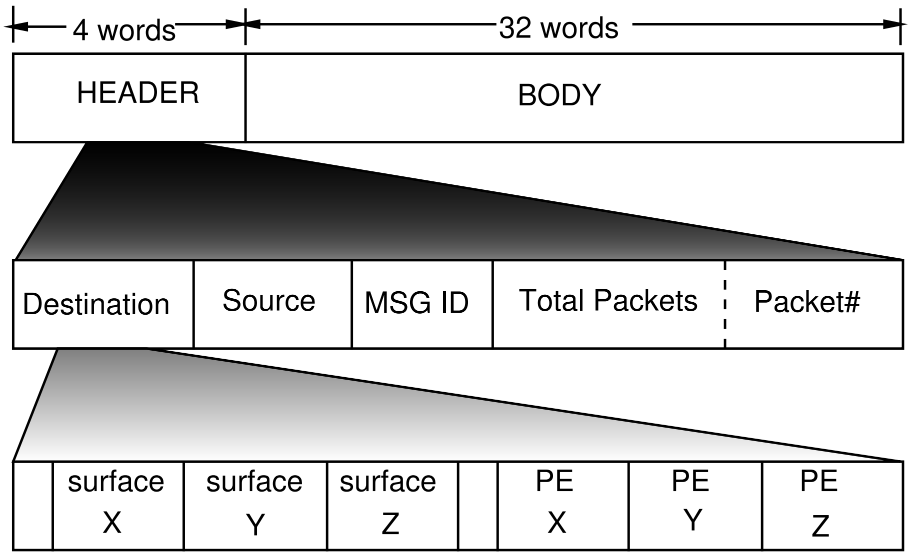


Figure 1: Packet Format

3 Post Office Behavioral Description

The Post Office [40, 41] is an autonomous packet delivery subsystem designed to be a co-processor to a controlling CPU. The interconnect topology is a wrapped hexagonal mesh used to create what we call a **surface**. Each surface has a hexagonal boundary and multiple surfaces can be interconnected by abutment. The Post Office is therefore a seven-ported device: six byte-wide external ports connect the Post Office to adjacent processing elements in the hexagonal mesh, and a 32-bit word-wide internal port provides a connection to the local CPU. The Post Office has been designed to permit all seven ports to be active concurrently.

The Post Office receives a packet from the CPU and delivers it over the communication network to a receiving Post Office, which in turn hands the packet to its CPU. Messages vary in length, but packet lengths are fixed at 36 words and have the format shown in Figure 1. The Post Office only uses the destination address field of the packet. The usage of the remaining 3 words of the header is dictated by Mayfly software convention. The Post Office contains an adaptive routing mechanism that can dynamically calculate the correct ports through which to route a packet. General communications efficiency can be significantly enhanced if the routing mechanism is capable of detecting congestion and routes packets around these congested areas. This mechanism also permits messages to be routed around nonfunctional nodes and ports.

Thus, the capability to dynamically route packets around congested or failed components implies that the order of packet arrival may vary from the order in which the packets were sent. The capability of reordering packets at their final destination is essential to ensure both deterministic behavior and to permit proper reassembly of multiple-packet messages by the destination CPU. This also implies that each packet contains a unique message identifier, the packet number, and the number of total packets in the message as shown in Figure 1. Although this slightly increases the amount of redundant information that must be transferred, it considerably reduces congestion related message delays. Congestion avoidance also increases the amount of packet traffic that can be in transit within the *postal* system before saturation effects are noticeable in terms of sharply increased packet latency times.

Messages are assumed to be delivered error-free in our current implementation and therefore no checksum or parity checking is performed. In order to avoid deadlock, a mechanism must be employed that either guarantees deadlock will not occur, or ensures that the probability of deadlock is much less than the probability of an unrecoverable component failure. The Post Office avoids physical deadlock by preventing unrestricted incremental resource claiming, thereby insuring that each Post Office will not be congested indefinitely. Note that livelock is still a possibility and there is no way that the Post Office can prevent a software deadlock situation where multiple user processes are all stalled waiting on each other. From the Post Office perspective, such a software deadlock situation looks exactly the same as the case when all PEs are busy but no messages are being sent.

For notational convenience, packets at a particular Post Office can be classified into 3 categories: transit, inbound, and outbound. Transit packets pass through a Post Office, inbound packets are destined for the local CPU, and outbound packets are created at the local node. The Post Office dynamically employs 4 types of routing algorithms: **virtual cut-through**, **best-path**, **no-farther**, and **random**. The algorithm choice depends on the specific congestion situation encountered by a particular packet. When a transit packet arrives at some external port, and the ideal external destination port of this Post Office is free, the packet will be forwarded immediately to the appropriate destination port. This method is called **virtual cut-through** [15]. The **best-path** method always takes the packet one step closer to its final destination. The **no-farther** routing neither increases nor decreases the distance of the packet from its final destination but allows it to orbit intervening congestion. The **random** method sends the packet out the first available port. In this case congestion may have the current Post Office surrounded, and it may be necessary to go farther away from the destination in order to make progress.

Virtual cut-through is only available for transit packets. If the ideal forwarding port is not available then the packet is stored in a central packet buffer pool in the Post Office. Inbound and outbound packets are always stored in the buffer pool prior to being forwarded to the local CPU or out onto the postal network. When an attempt to route a buffered packet is made, the best-path method is used. If it fails a **stagnation counter** is incremented. When the stagnation count for a particular packet exceeds its threshold then no-farther routing is also an option. Failure again increments the stagnation count. If a second threshold is exceeded then random routing is also an option. In any case

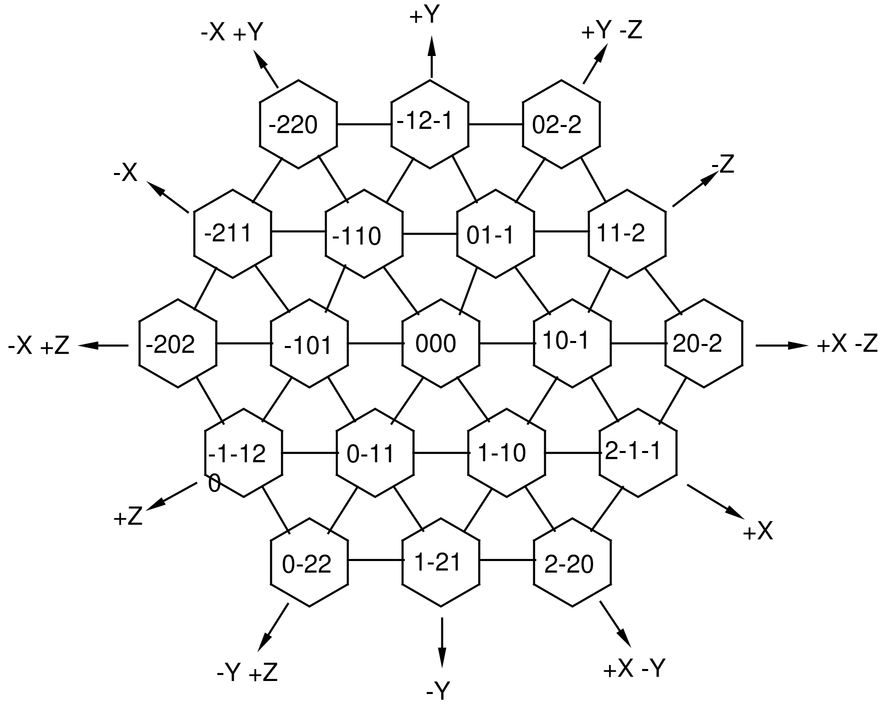


Figure 2: PE Labeling Convention

where multiple available ports are found under any routing option situation, best-path will be taken first, no-farther second, and random last.

The Post Office routing algorithm is based on a labeling of PEs in the interconnected surface topology shown in Figure 2. Each of the PEs is assigned a unique location number which is a 3-tuple that corresponds to an <x,y,z> location in the surface. Note in Figure 2 that the X, Y, and Z axes correspond to the edges of the surface rather than the edges of the PE which are connected by the communication links. The reason for this choice is that it simplifies the routing algorithm. Each link traversed by a packet in either direction crosses 2 of the axes, one in the positive direction and another in the negative direction. Also note that the sum of the 3 location components is always equal to 0. This feature is used to detect illegal addresses. If a message is to be sent from the center PE to any other PE in the surface it is easy to understand how to make the decision as to which Post Office port should be used. For example, suppose a message is being sent from <0,0,0> to <1,-2,1>. The relative difference is <1,-2,1>. Since the y component is the largest and the value is negative, the **best-path** must be to send the message out either port that heads in the $-Y$ direction. The **no-farther** paths are the two paths which do not cross the Y axis.

The proper routing decision is much less obvious when the source PE is at the periphery of the surface, since a wrap link is a possibility. E.g. the $-X, +Y$ port of PE <-1,2,-1> is wrapped to the $+X, -Y$ port of node <1,-2,1>. The relative difference between these two paths is <-2,4,-2> which does not correspond to the possible single $-X, +Y$ move. One option would be to use a table based routing scheme; however the table size is quadratically dependent on the number of PEs and would be prohibitive. Therefore the Post Office computes the routing decision using a method described below. The result is a Post Office element which is fast, general, and scalable. The only configuration specific parameter that must be loaded into a Post Office register is a single integer n indicating a surface of size E–n, where n is the number of PEs on the surface edge. Figure 2 shows an E–3 surface.

The Post Office routing method removes the potential complexity imposed by the wrap lines by normalizing addresses to *center-based* coordinates. The method is:

1. Subtract the current packet location (the local PE label) whose value is stored at initialization time in a Post Office control register from the destination address. The result is an **unnormalized relative address.** For an E–n surface, a wrap line will be needed if one of the components of the resultant value is greater than n.

2. In order to convert the unnormalized relative address to a **normalized** form, which makes the current PE appear to be in position <0,0,0>, it is necessary to use a normalization vector. This vector has the value <2n-1, n-1, n>. For an E–3 surface, this value is <5,2,3>.

3. This normalization vector must then be aligned with the unnormalized relative address. The relative positions in the normalization vector must be preserved in the alignment, so a right end-around shift is used. There are two alignment conditions:

   (a) The 2n-1 component in the normalization vector is aligned with the component of largest magnitude in the relative address when no component of that address is zero.

   (b) Otherwise the n-1 component is aligned with the zero component in the unnormalized address.

4. The sign of the 2n-1 term of the normalization vector is set to match the sign of its aligned component. The signs of the n-1 and n components are the negation of the 2n-1 component's sign.

5. The normalization vector is subtracted from the unnormalized relative address to yield the normalized relative address.

6. At this point the routing decision is simple. The **best-paths** must reduce the largest magnitude component of the normalized relative address since this magnitude indicates the present distance of the packet to its final destination. In cases where there are no zeros in the normalized relative

address and when there are two possible ports, one of which does not result in a zero relative address component at the next stage, the selection is made which does not result in a zero relative address component. A zero component in the normalized relative address indicates that there is only one **best-path** route. The goal is to keep multiple **best paths** available whenever possible.

4 High-Level Implementation

4.1 Functional Blocks

The Post Office consists of a set of asynchronous control logic, communication links, and datapath circuitry. Figure 3 shows the major logic blocks in the Post Office. There are five major components; the external port interface, adaptive routing and forwarding, bus arbitration block, central buffer controllers, and the internal PE interface. Each logic block is an independent agent, and can execute concurrently with the other components. The static buffer controller consists of 19 independent controllers, each with an associated packet buffer, seven of which can execute concurrently.

Submodules in the Post Office communicate via the *spine bus* using a four-cycle hand-shaking protocol. Each module must obtain mastership of the spine bus before its transaction can occur. There are 29 logic blocks which can obtain bus mastership. The Bus Arbiter block centrally services bus mastership requests. Whenever any module requires bus operations, it will assert its *Bus-Request* signal. When the requesting module receives the *Bus-Acknowledge* signal, it may drive the control and data signals on the bus. The arbitration logic is centralized as it results in faster servicing of bus requests when compared to distributed schemes. However, centralization requires a significantly higher wiring complexity.

Once a logic block has mastership of the bus, it may execute one or more transactions. The spine bus signals include three self-timed control lines ($\overline{Req}$, $\overline{Ack}$, and $\overline{Nak}$), a three-bit opcode, an eight-bit address, a three-bit sender address, a six-bit routing result addresses, and 128 bits of packet data. Typical transactions on the spine bus are to allocate a port for packet delivery, or transmit a 128 bit **line** of packet data between two blocks.

The CPU communicates with the Post Office via a 32-bit addressable register interface. The Post Office can run in either a polling or interrupt mode for either arriving and/or outgoing packets. A set of control registers can mask and disable the interrupts. There are also FIFO registers for loading and unloading packets, plus several control, status, and debugging registers.

Upon power-up or initialization, the current surface size and processor address must be loaded into control registers before packets can be correctly routed to their destination. The CPU must also write the normalization vector into the routing register. Data is normally transferred as an entire 144 byte packet. The only exception to this is on the PE interface which permits the CPU to quit reading or writing early by issuing a *done* command. This

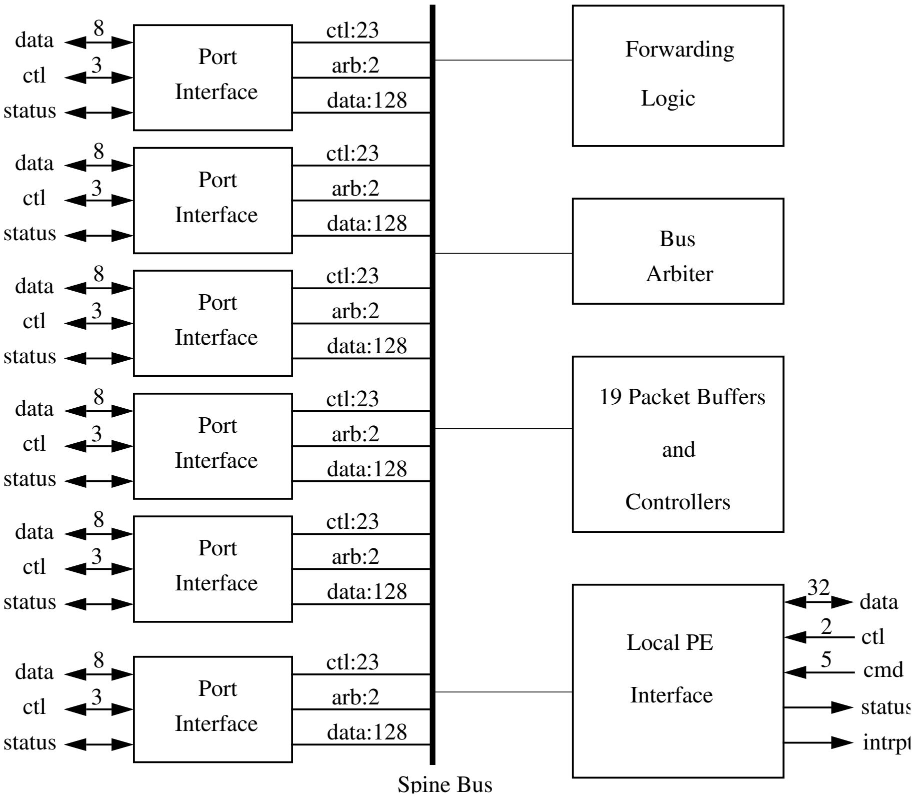


Figure 3: Post Office Block Diagram

provides additional software protocol flexibility and increased performance.

All registers and memory in the PE interface are static. Although the Mayfly architecture is designed to service message traffic quickly [11], an inbound or outbound packet can remain unread or partially loaded in the buffers indefinitely since the CPU's software protocol will determine the timing of packet reads and writes.

The *packet buffer controllers* and *forwarding logic* blocks cooperate to centrally buffer and forward packets between the external ports. The forwarding logic is responsible for finding a free port for packet delivery using adaptive best-path, no-farther, and random routing. The buffer controllers snoop spine bus transactions between the forwarding logic and external ports to determine when to actively deliver and latch a packet to/from a port. Packet buffer storage requests are verified by the forwarding logic block. If there are insufficient buffers to guarantee deadlock free operation, the request will be rejected. Otherwise, the best-path address for the packet will be latched and a free packet buffer address will be returned to the requester.

Reliability requires that at least one Post Office node contain a valid copy of every packet being delivered. The PE interface loads each outbound packet into a central packet buffer. The source Post Office is responsible for keeping a valid copy of the message until it has been accepted and placed into the packet buffers of another Post Office (which could be several hops away due to cut-through). The first buffer location can then be freed for storing another packet. Centrally buffering outbound packets also increases bandwidth utilization of the external ports as packets are not forwarded until they are entirely received. The pipelined, cut-through transmissions can always proceed at full port-interface speed, independent of the PE interface logic. The speed of the PE interface may vary depending on the CPU speed, the amount of data to be transmitted, the drivers, or CPU state. The requirement that packets be stored in the buffers at the source and destination nodes increases the latency in lightly loaded networks. However, average latency will be much lower in cases where either local or global congestion is present.

The port interface controls transfers across an eight-bit bidirectional link connecting two Post Office chips. One interface becomes the sender, while the other becomes the receiver. Interface Hand-shaking uses three control wires and a two-cycle protocol. The port interface will receive a service request to forward a packet to the adjacent chip. If the interface is busy, either sending or receiving a packet, the delivery request will be rejected. When a delivery request is made to an idle interface, the interface must arbitrate for control of the external link. If this interface wins the arbitration, then the delivery is accepted and this interface becomes the sender and must reject the pending delivery request.

The first four bytes loaded by a receiving port interface contain the destination address of the packet. After latching the address the routing logic calculates the delivery port(s). The packet can then be immediately forwarded using virtual cut-through to the destination port, or to the packet buffers if this node is the final destination. When cut-through forwarding through the shortest path(s) cannot occur, the interface will attempt to store the packet in the packet buffers. If there is no room in the buffers due to deadlock prevention, the delivery to this node will be rejected.

4.2 Design Style and Circuit Elements

The Post Office consists of four types of components: finite state machines, RAM, arbiters, and some datapath logic such as subtracters, comparators, counters, registers, etc. All of these components were designed as self-timed modules, which were interconnected to form the larger elements in a hierarchical fashion. The largest self-timed element is the entire Post Office chip.

All state machines were designed from burst-mode AFSM descriptions. We assume arbitrary logic delay and insignificant wire delay. The environment is assumed to enforce burst-mode operation where all outputs will be driven before the next input burst arrives. The speed-independent model includes the isochronous fork assumption [48], and also applies to the verifier used to check our designs. Filtering or *triggering* of slowly rising input signals is used to ensure that this assumption doesn't create problems within state machines due to logic threshold variances.

Performance can be greatly enhanced if asynchronous *library* components such as C-elements, toggles, selectors, etc. are not used. The physical operation of sets of library components can usually be collapsed into asynchronous state machines, which are both smaller and faster than the library components. These AFSMs exhibit concurrency when designed using burst-mode. However, AFSMs cannot be designed where there are nondeterministic races between input signals. Hence there are only two library components which are required in our AFSM based design; the mutual exclusion or (ME) element and the C-element. Arbiters used in the Post Office consist of one or more ME and a finite state machine.

4.3 Datapath Logic

Data is transferred between elements using a two- and four-cycle self-timed bundled-data protocol [42]. This is a weaker model than the speed-independent model used for control signals inside state machines [27]. Also, some datapath circuits are controlled in a *clocked* domain, rather than in a self-timed fashion. These datapath components either contain a stoppable clock or are clocked by state machine outputs. The stoppable or raw clock signals are usually burst-mode generated in parallel with an associated set of asynchronous hand-shake signals. The minimum delay of these hand-shake signals must be greater than the maximum delay required by the clocked circuitry.

An example can be found in the port interfaces. The number of bytes which have been transmitted is recorded by a shift register. A state machine is signaled when the last byte has been transmitted, indicating the completion of the data delivery phase. The counter consists of an eight transistor dynamic shift register (plus one transistor for initialization). It is clocked by a state machine that latches and enables the packet data onto the external bus. The easily verifiable timing assumption is that the shift register *done* signal will arrive before the completion of the latch/enable cycle.

Making timing assumptions for datapath logic has two consequences. First, the logic is smaller and simpler because no completion signals are generated. Second, there is an increased potential for errors, as correct operation is now dependent upon physical circuit layout, process, and environmental parameters. Hence this is rarely used except for easily verifiable datapath components where completion signal generation is too costly in terms of speed or area.

Each RAM block is the size of a packet, 1152 bits. They are configured in 128×9 arrays, so it is fairly efficient to detect when a word line has been driven. Data validity and precharge is sensed on a single bit line pair, with the assumption that all bit lines will exhibit similar delay.

Many resources in the Post Office are accessible by other components. Accesses to these resources are made in a nondeterministic but mutually exclusive fashion. For example, two port interfaces may attempt to transfer data along the spine bus. Also, port interfaces A and B may both request forwarding a

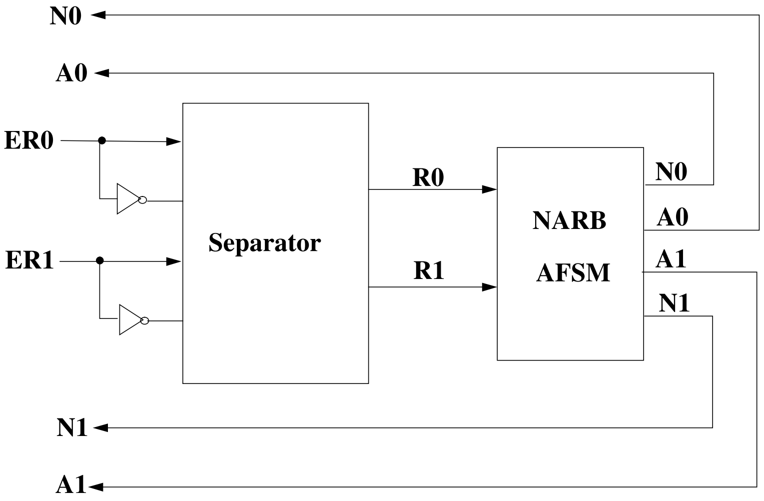


Figure 4: Nacking Arbiter Block Diagram

packet through port interface C. These two cases require different arbitration circuits. In the first case, the two accesses will be serialized on the spine bus; the loser of the arbitration will be blocked until the resource is freed. In the second case, the accesses should *not* be serialized! The winner will transfer its packet through port C, but the loser should not be blocked until the winner's transfer completes. Rather, it should abort the operation and be allowed to move on to other operations. Standard arbitration will serialize accesses for the first example, but a *naking* or *nonblocking arbiter* is required for the operation specified in the second example.

There are 13 naking arbiters in the Post Office. We found nonblocking arbiters to be a simple yet intriguing asynchronous circuit and posted it as a design and implementation exercise to our peers who are designing asynchronous tools and circuits. Our solution consists of a "separator" (which is built out of ME elements and a few NAND gates) and a state machine synthesized by MEAT. The block diagram of this circuit is shown in Figure 4. The circuit for the separator is shown in Figure 5 and the state machine is shown in Figure 6.

5 MEAT

The MEAT tools are sufficiently fast that alternative design options can be explored. The designer is freed from the task of understanding the underlying transformations required to produce hazard-free asynchronous circuits. Asynchronous circuits are specified for MEAT as burst-mode Mealy state machines. This style of specification provides a powerful way of encapsulating concurrency, communication, and synchronization in an accurate and easily understood form. The input specification is compiled into a set of CMOS com-

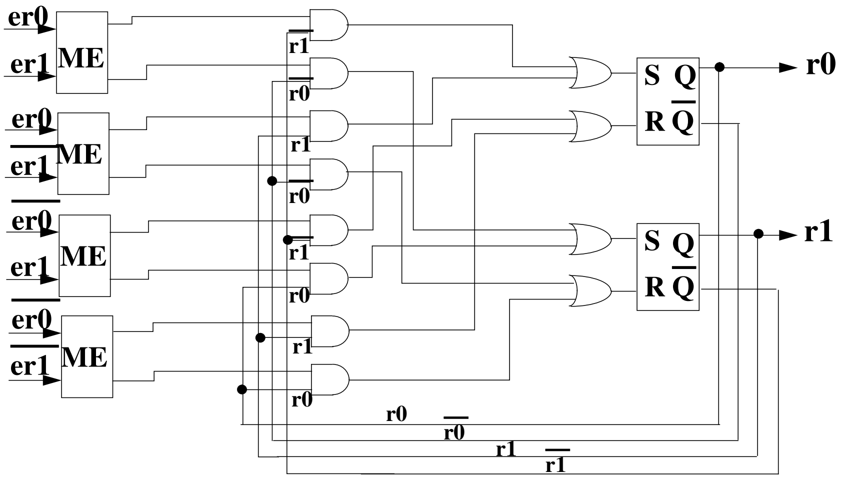


Figure 5: Nacking Arbiter Separator

plex gates. The result is an implementation which is efficient in both speed and area.

State flow diagrams are used to model the behavior of state machines implemented using MEAT. They provide an intuitive method for defining control functionality, and are similar to the flow charts and state diagrams that are commonly taught in multiple disciplines today [13]. A finite state machine is modeled as a directed graph, where the nodes represent states and arcs represent transitions between states. Each arc is labeled with the set of input firings which trigger the transition and an associated set of output firings. These state diagrams can easily represent parallelism and synchronization, and are reasonably compact when compared to other graphical specification methods.

MEAT state diagrams allow a constrained form of MIC operation, which we refer to as **burst-mode**. When a state change is triggered by a conjunction of input signal transitions (an input burst), these signals are allowed to change in any order and at any time. Allowing MIC operation simplifies the definition of synchronization operations and tends to more closely match the designer's mental model of the hardware. Presently MEAT does not contain a state graph editor so a textual specification format is used. The more natural graphical state machine description may be trivially mapped to the textual version: each arc in the state diagram is mapped to a single statement in the text file, which indicates the source and destination states along with the associated input and output bursts.

Burst-mode state diagrams are reasonably compact when compared to petri-nets, m-nets, STG's, and other graphical representations. These diagrams work well for transition (2 cycle) or level-mode (4 cycle) signaling protocols. Figure 7 shows an example of an STG (a), enhanced STG (b), and burst-mode state diagram (c) for an asynchronous D flip-flop. In this paper we assume positive logic, hence $a\uparrow$ corresponds to a high transition on signal a. In the

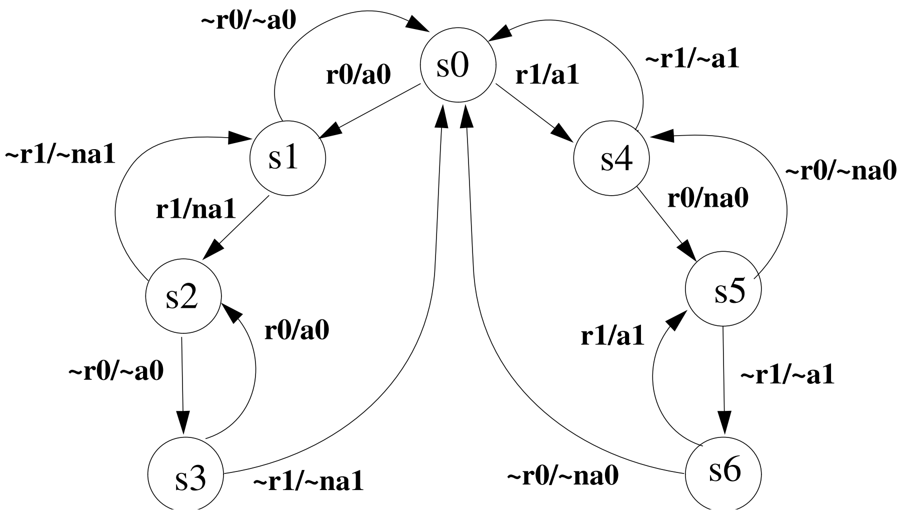


Figure 6: Nacking Arbiter State Machine

textual version $a\uparrow$ is represented simply as a and $a\downarrow$ as $a\sim$. The corresponding textual entry version for MEAT is:

```
:fsm Asynch-Flip-Flop          ;name of FSM for documentation.
:in  (D Clk)                   ;list of input variables.
:out (Q)                       ;list of output variables.
:init-in  ()                   ;initial value of inputs, default zero.
:init-out ()                   ;initial value of outputs, default zero.
:init-state 0                  ;initial state, default is State 0.
:state 0 (Clk~) 0 ()          ;specification of state transitions:
:state 0 (D * Clk) 1 (Q)      ;  format is <current state> <input burst>
:state 1 (Clk~) 1 ()          ;            <next state> <output burst>
:state 1 (D~ * Clk) 0 (Q~)
```

The first automated task performed by MEAT is to generate a primitive flow table [46] from the textual FSM specification. This is a two-dimensional array structure which captures the behavior represented by the state diagram in a more detailed form. Each row of this table represents a node in the state diagram; each column represents a unique combination of input signals. Each entry in the table thus represents a position in the possible state space of the FSM.

For each entry, the value of the output signals and the desired next state may be specified. If a next-state value is the same as that of the current row, the state machine is said to be in a *stable state*. If the next-state value specifies a different row, the table entry represents an *unstable state*. A simple way of understanding the flow table is to note that horizontal movement within a row represents changes in the values of input signals, while vertical movement within a column represents a *state transition*. All of our specifications are given

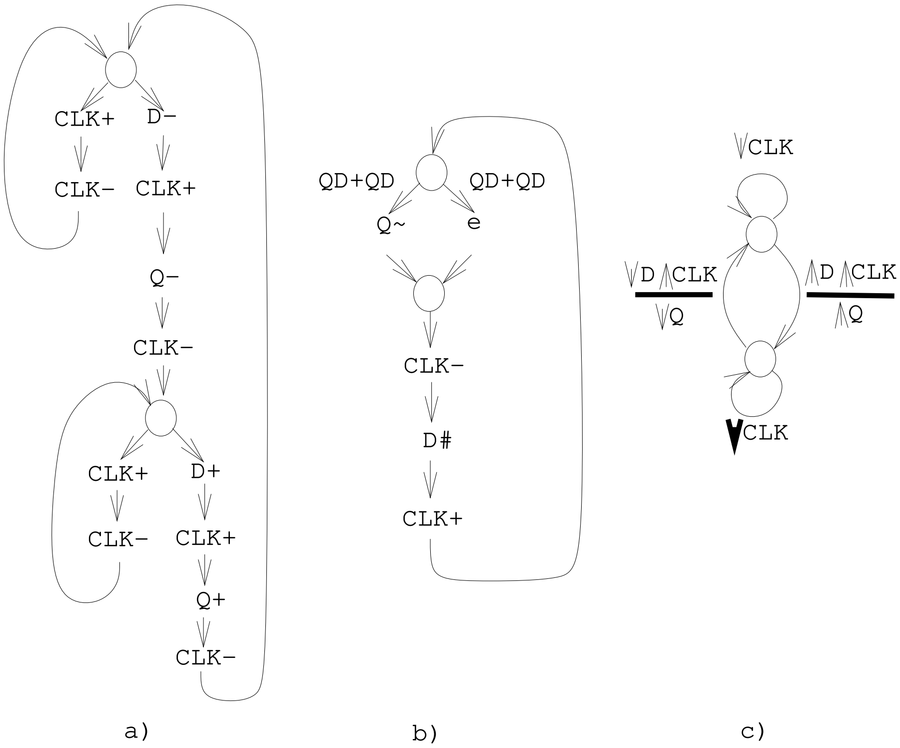


Figure 7: Sample Flip-Flop Specifications

in *normal form*, that is, each unstable entry in the table must lead directly to a stable state.

Each allowed input burst will result in a particular path through the FSM state-space, starting at the stable entry where the burst begins. Other entries in the same row may be visited during the course of the input burst. In order for MIC behavior to be correctly represented, it must be guaranteed that the circuit will remain stable in the initial row until the input burst is complete. This is an important point and is a cornerstone of the burst-mode methodology. In essence, any minterm formed from input variables which can be reached during the course of an input burst must be *covered* by a stable entry in the flow table. The minterm defined by the completion of the burst will correspond to an unstable state which will cause a transition to the target row and fire the output burst.

The output burst, if any, may occur concurrently with the state change, or may be constrained to happen *after* the state change has occurred. To allow the flexibility for the later synthesis stages to choose either option, signals in the output burst are labeled as don't cares in the unstable exit state of the flow table. Since all state transitions are STT *Single Transition Time*, the monotonicity of output voltage changes is guaranteed, regardless of whether the value of a given output transition in an unstable entry is mapped to logic level zero or one.

Any entry in the flow table not reachable by any allowed sequence of input bursts is labeled as a *don't care* and can take on any value for the outputs or next-state values. As in the case of output bursts discussed above, it is not immediately evident which values will lead to the simplest circuit. Therefore, the assignment of specific values to the don't care entries is deferred for as long as possible. The inclusion of these don't cares can significantly simplify state reduction and boolean minimization, and also lead to more compact circuits.

The next step in the design process is to attempt to reduce the number of rows in the flow table by merging selected sets of two or more rows while retaining the specified behavior. This involves first calculating the set of *maximal compatible* states. The set of maximal compatibles consists of the largest sets of state rows which can be merged, which are not subsets of any other such set. There may be various valid combinations of the maximal compatibles that can be chosen to produce a reduced table with the same behavior.

This is essentially the well-known state-reduction problem; unfortunately complications are introduced due to the MIC nature of the input bursts. "Traditional" methods normally apply only to SIC circuits, and when used for our burst-mode specifications may produce hazards in the final implementation.

Nowick et. al. [32] have developed the modifications necessary to the state-reduction and subsequent synthesis steps to guarantee that the resulting implementation will be hazard-free under burst-mode conditions. These modifications are not presently incorporated into MEAT. Currently we use a verifier [12] on the synthesized implementation. The verifier has been modified to operate with explicit timing assumptions. Hazards detected in the implementation are then reviewed to see if the circuit would exhibit correct behavior under reasonable delay assumptions. If these assumptions fall within acceptable bounds of fabrication and operational constraints then the timing assumption is entered into the verifier. If an unacceptable assumption is required then the circuit is fixed either by manual repair or by modifying the state-machine specification. The manual repair usually involves the addition of an appropriate inverter chain to delay the critical path race.

The final choice of minimized states is an example of the *binate covering* problem. There are three constraints on this choice. First, and obviously, only compatible states may combined (*compatibility* constraint). Second, each state in the original design must be contained in at least one of the reduced states (*completeness* constraint). Third, selecting certain sets of states to be merged may imply that other states must also be merged (*closure* constraint). Grasselli and Luccio [16] have developed a tabular method for determining a closed cover of states, which is also in the process of being incorporated into MEAT. At present, MEAT requires the user to manually determine and enter a state covering. If any of the necessary constraints are not satisfied, MEAT will inform the user that the covering is invalid.

A new flow table representing the behavior of the minimized FSM is then generated by merging the specified rows of the original flow table. It should be noted that it is not always true that minimizing the number of states will simplify the hardware or increase performance. However, a reduced state machine can result in fewer state variables which in most cases does indeed result in a smaller and faster implementation.

A set of state variables must then be assigned to uniquely identify each row of the reduced flow table. These state variables are used as feedback signals in the final circuit. In contrast to synchronous control logic design, state codes may not be randomly assigned, but must be carefully chosen to prevent races. The MEAT state assignment algorithm is based on a method developed by Tracey [44]. The Tracey algorithm has the advantage that it produces STT state assignments which minimize delay in the implementation. In cases where two or more state variables must change value when transitioning to a new state, all variables involved are allowed to change concurrently, or *race*. However, it must be guaranteed that the outcome of the race is independent of the order in which the state variables actually transition in order to produce a *non-critical* race which exhibits the correct asynchronous operation. Several valid assignments may be produced, and each will be passed to the next stage for evaluation. Each state assignment will result in a unique implementation.

After state codes are assigned, the next synthesis stage computes a canonical sum of products boolean expression for each output and state variable. A modified Quine-McCluskey minimization algorithm is used. The resulting expression includes all essential prime implicants, and possibly other prime implicants and additional terms necessary to produce a covering free of logic hazards. It may be possible for each output or state variable to be specified using several other minimal equations. The large number of don't care entries typically present in the flow table causes the standard algorithm to be rather inefficient and increases the likelihood that more than one minimal expression will be found. The MEAT implementation contains optimizations for don't care dominant functions. Each possible solution is given a heuristic "weight" that indicates the expected speed and area cost of implementation using complex CMOS gates. When multiple state assignments have been produced in the previous step, the total weight of each unique SOP (sum of products) equation is then used to choose between the various instantiations.

The minimized equations produced in the previous step are then used to automatically produce transistor netlists, suitable for simulation, representing complex CMOS gates. An interface to the Electric [37] design system is used to generate a schematic diagram to help guide the layout process, which unfortunately has not yet been automated. The complementary nature of CMOS n-type and p-type devices is exploited to generate a single, complex, static gate through simple function preserving transformations. These transformations can increase performance while reducing the area and device count. As a SOP equation is *folded* into a complex gate, the number of logic levels required to generate the output can be reduced. If the function is too large to be implemented as a single module, it can easily be broken up into a tree of complex gates with 2 or more logic levels, but better overall performance [43]. Typical state machine implementations have response times between 3 and 5 2-input NAND gate delays.

Our complex gate design generates negative logic outputs (low voltage levels for asserted signals). A convention of positive logic levels is assumed for all signals external to the state machine, requiring that the outputs be inverted. This is a feature for performance reasons as the gain of the inverter can be used as a driver to increase signal strength and reduce rise and fall times. When outputs need to drive a large load, a buffer tree can be used.

All state machines also require a reset signal to place the storage logic into the correct initial state. Storage in these state machines is implemented via the state variables. If a single complex gate is used to generate the output, the state storage is reset by NOR-ing the output with the reset line. For complex gate trees, a resettable NAND gate is used. Although the performance of the NOR gate is not optimal, the load on the feedback lines is local to the state machine and typically small so a large gain is not required.

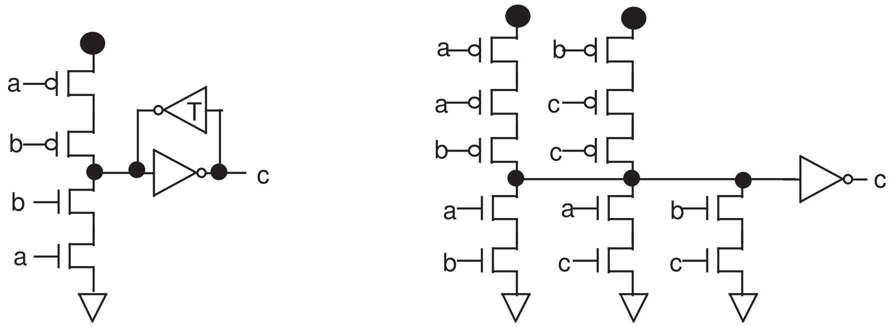
6 Design Issues and Examples

Figure 7 essentially shows how AFSM designs are specified using MEAT. Rather than presenting a series of more complex designs which will show roughly the same thing, we will present a number of design vignettes which illustrate interesting points in the design space, and an example of MEAT usage.

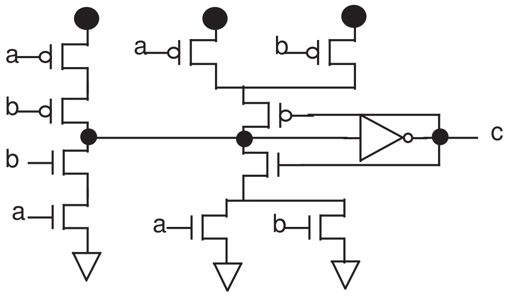
6.1 A Story about C-element Design

During a 1986 course on asynchronous circuits taught by Ivan Sutherland and Bob Sproull, the discussion turned to the design of the common C-element. At that time, the standard C-element consisted of a 2-high stack, followed by an inverter. This element had the problem that it was a *dynamic* gate. If the two inputs remained at different voltage levels for long enough the C-element's state would be lost and cause an invalid output transition. This was clearly unacceptable for general asynchronous applications. Static versions of the circuit were created by including a weak "trickle charge" inverter to maintain correct voltage on the internal node in the absence of it being directly driven by the 2-high stack.

The trickle charge inverter was a problem for several reasons. First, it reduced the performance of the circuit. When the internal node $\overline{c}$ (in Figure 8a) needed to be flipped to a different voltage, the trickle inverter would be actively driving the circuit one way, while the 2-high stack was actively driving it another way. The 2-high stack needed to charge the node, as well as dissipate the current supplied from the trickle inverter. This caused increased power consumption due to the existence of a DC path between the power rails during a state change. Secondly, the inherent gain of an inverter is greater than the gain of a 2-high stack. This design requires the 2-high stack to overpower the inverter to flip the state of the device. Unless the drive of the 2-high stack is significantly greater than the inverter, the node becomes susceptible



a) Trickle inverter C-element b) Complex gate for c = ab + ac + bc



c) Optimized complex gate C-element

Figure 8: C-Elements, Hand Optimized Matched by MEAT

to noise problems which could result in hazards. This gain difference can only be overcome by reducing the size of the inverter and increasing the size of the 2-high stack. Hence the sizing of the components becomes critical. Increasing the size of the 2-high stack slows the circuit by requiring additional input drive. Decreasing the width and increasing the length of the inverter reduces the reliability of the inverter and the portability to other processes.

After the day's discussion, we spent several hours attempting to come up with a better C-element design which eliminated the trickle inverter, yet did not add significant complexity to the component. Ultimately a design was found which was compact and efficient. This design has been widely used in a number of sites. This design required 4 more transistors than the trickle charge design. However, the 2-high stack could be of optimally sized transistors and there was no fight to drive the internal node $\overline{c}$. Although this circuit was larger, and the inputs drive twice the number of devices, it was significantly faster than the original design and avoided the power consumption, noise, portability, and function problems of the old design.

Several years later, curiosity lead us to see what MEAT would produce for a C-element. The exact same circuit was produced from MEAT in an instant. MEAT generated equations for the circuit shown in Figure 8b and the back-end schematic generated the equivalent but optimized version shown in Figure 8c.

6.2 Using Burst-Mode to Increase Performance

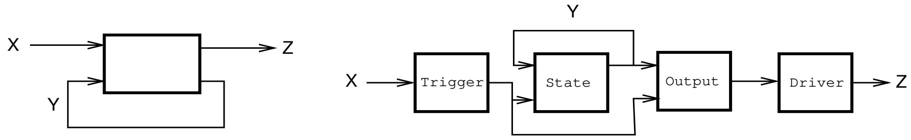
Burst-mode assumes that inputs and outputs are generated as discrete sets, or bursts. In general, this violates delay-insensitive and speed-independent assumptions. For example, assume that an input burst has completed, and the resulting output burst causes several outputs to be generated. One of the outputs could be generated before the others. This output can be received by a destination module which could in turn generate an output which is fed back as an input to the original module even *before* the rest of the outputs have been generated. This violates burst-mode operation as the next input burst has occurred before the previous output burst has completed. Burst-mode assumes that all outputs in the burst must be generated before the environment can respond to the output burst or computation interference may occur. The cases where computation interference can occur can be flagged and checked by circuit timing analysis.

MEAT's burst-mode MIC model is similar to the fundamental mode assumptions for traditional SIC AFSM designs. Namely we assume that once an input burst has arrived the AFSM will settle in a stable state before the next burst can arrive. If this assumption cannot be met then external arbitration will be required to enforce the assumption.

If an input burst changes an internal state variable, speed-independent operation will generally require the state variable to stabilize before the output can be changed. Performance can be improved if outputs can change concurrently with state changes. MEAT accomplishes this by making the transitioning output a don't care in the unstable exit point of a row in the flow table. This places a priority on logic minimization, but usually will produce a circuit which can generate an output concurrent with state changes. The fundamental mode assumption guarantees that the AFSM is ready to accept the next input burst when it arrives, as the state variable transition has completed and the logic has stabilized. Unger has shown that it is possible to weaken this fundamental mode assumption [45], although his method is not presently incorporated into MEAT.

6.3 When Speed-Independent Circuits Fail: The Isochronous Fork

Ideally all asynchronous circuits should be designed as delay-insensitive modules. However, performance requirements may force one to make weakening assumptions about circuit behavior. Many of these assumptions are realistic, as physical devices and wires do not require unbounded delays to generate and propagate signals. However, care must be used to assure that the circuit complies to these assumptions under all operating conditions or the design will be



State Machine

MEAT State Machine Logic Blocks

Figure 9: State Machine Generation

unsafe and costly failures may occur.

Simplifying assumptions are best exploited when they are constrained to a fixed extent physical domain as is the case with AFSM modules. Hierarchical composition of these modules can then proceed conforming to delay-insensitive rules since all of the external interfaces can be designed to avoid timing assumptions. Inside an AFSM, the relative delay of wires and gates can be more easily controlled, analyzed, and modified as the constraints are all local. When these timing assumptions apply outside an individual module then the entire system must be analyzed to assure compliance with the timing assumption set. At this point there is little to distinguish the circuit from a synchronous one.

A common performance and synthesis assumption made by many asynchronous circuit designers is that of speed-independence. The assumption that wire delay is zero leads to the **isochronous fork assumption**. This implies that multiple devices driven by a single component react to the signal change at very nearly the same time. This model works well for situations where the transistors are slow and the paths are fast. Unfortunately this model becomes less valid as IC technology progresses and is suspect even today.

Furthermore, whenever the rise or fall time of an isochronous fork is greater than the switching delay of any physical device, failure may occur due to variances in switching thresholds. Noise, long wires, and high-capacitance paths exacerbate the problem. Within a particular AFSM module, this problem can be managed successfully but it is difficult between modules. Martin [24] and Van Berkel [48] have both described circuit failures due to paths which did not behave in an isochronous fashion. Both failures were the result of using C-elements in module interfaces. C-elements inherently contain an isochronous fork, since the output of the C-element will be an output of the module as well as being fed back locally to maintain the C-element's state.

The philosophy we have used in the MEAT tool and in the design of our circuits is to remove isochronous forks from external interfaces. MEAT state machines are broken into the partitions shown in Figure 9. Our philosophy is that we would gladly increase the cost and difficulty of designing modules if it simplifies the composition of systems. Timing assumptions are always easier to analyze and fix in a small, local cell rather than across a series of modules.

The trigger box has two functions. First, high capacitance inputs (inputs with a slow rise time) will be passed through an inverter or Schmitt trigger. This will reduce the load on the input line, which can increase circuit performance. It also results in crisp rise and fall times of signals internal to the

AFSM. Secondly when an unasserted input signal is required by the state or output boxes, the trigger box will invert that signal. Each input will have its inverted and uninverted signal shared by all function blocks in the state machine to eliminate hazards and create a smaller implementation. The isochronous forks created by sharing the inverters are easily controlled within the local domain of the AFSM since components within a particular AFSM are physically close. Hence wire delays of the internal signals and the trigger box delay are normally insignificant.

The driver block is used to generate positive output voltage levels and to increase the signal strength when the output is heavily loaded. Circuit performance is enhanced since it is sized to drive its output load appropriately. Isochronous forks in a MEAT design will only exist when a state variable is used directly as an output. In such cases, the output can be buffered by one or two inverters to assure that the fork is isolated within the AFSM. While this decreases the performance of the circuit, the module can function in a delay-insensitive manner and can be used safely without analyzing its load in a broader context.

This design style has been tested continuously over the last five years. We have designed several large asynchronous circuits which have usually worked the first time, using simulators to verify correct composition of the modules. The result of this experience has led to a high confidence factor in the practical value of the method.

6.4 An AFSM example

In order to illustrate exactly what MEAT does, we will transcribe an actual synthesis run using MEAT to create a Post Office state machine called the SBUF-SEND-CTL. The behavior is initially specified as a burst-mode AFSM as shown in Figure 10. This example is taken from the suite of Post Office state machines publicly available for use by other researchers [21, 35].

The specification of sbuf-send-ctl from Figure 10 is textually entered for MEAT as follows:

```
:fsm sbuf-send-ctl
:in  (Deliver Begin-Send Ack-Send) ;list of input variables
:out (Latch-Addr IdleBAR Send-Pkt) ;list of output variables
:state  0 (Deliver)
        1 (IdleBAR * Latch-Addr)
:state  1 (Deliver~)
        2 ()
:state  2 (Begin-Send)
        3 (Latch-Addr~)
:state  3 (Begin-Send~)
        4 (Send-Pkt)
:state  4 (Ack-Send)
        5 (Send-Pkt~)
```

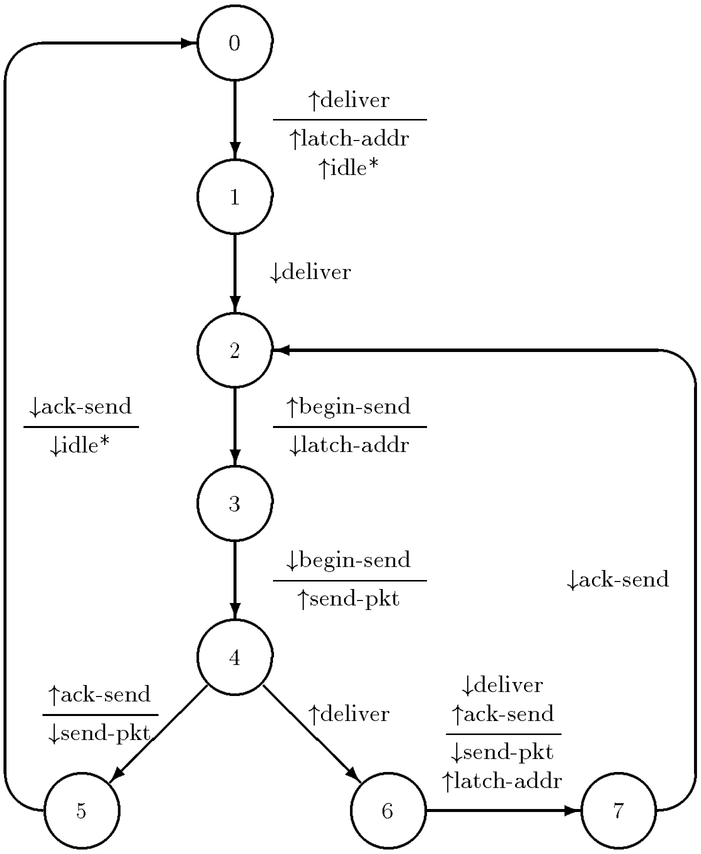


Figure 10: Sbuf-send-ctl State Machine

```
:state  5 (Ack-Send~)
        0 (IdleBAR~)
:state  4 (Deliver)
        6 ()
:state  6 (Deliver~ * Ack-Send)
        7 (Send-Pkt~ * Latch-Addr)
:state  7 (Ack-Send~)
        2 ()
```

The following is a transcript from a MEAT session. The specification resulted in a single implementation with two state variables.

```
> (meat "sbuf-send-ctl.data")

Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
Enter State set: '((0 5) (1 2 7) (3 4) (6))
```

```
SOP for "Y1":
  18: DELIVER + Y1*BEGIN-SEND~
SOP for "Y0":
  28: BEGIN-SEND + Y0*ACK-SEND~ + Y0*DELIVER
SOP for LATCH-ADDR:
  12: Y1*Y0~
SOP for IDLEBAR:
  30: ACK-SEND + BEGIN-SEND + Y0 + Y1
SOP for SEND-PKT:
  12: Y0*BEGIN-SEND~
 HEURISTIC TOTAL FOR THIS ASSIGNMENT: 100
```

The implementation can then be checked for hazard-free operation by the verifier. The verifier reads the specification and implementation. For this example, the state variables and outputs generated by MEAT are implemented as two-level AND/OR logic. Each signal is generated independently of the others. Only direct inputs are shared, so the same inverted signal in different output logic blocks will use separate inverters. Separate inverters will result in verification errors in the burst-mode speed-independent analysis. In this example, the $\overline{\textit{begin-send}}$ signal is shared by *Y1* and *send-pkt*. The two inverters are merged and the output is forked to both logic blocks. The fresh implementation is then checked. The verifier points out a d-trio hazard [46] which is removed by adding an inverter to change the sequencing of begin-send into the *Y0* logic. The final implementation is then verified as hazard free as follows:

```
> (verifier-read-fsm "sbuf-send-ctl.data")

Max Compatibles: ((0 5) (1 2 7) (3 4) (6))
Enter State set: '((0 5) (1 2 7) (3 4) (6))

> (setq *impl* (merge-gates '(1 11) *impl*))
> (verify-module *impl* *spec*)
10 20 30 40 50
Error:  Implementation produces illegal output.

> (setq *impl* (connect-inverter 10 6 *impl*))
> (verify-module *impl* *spec*)
10 20 30 40 50 60 70 79 states.
T
```

The canonical SOP equations generated by MEAT are then transformed into complex gates for implementation. The CMOS circuit for *Y0* is shown in Figure 11. The complex gates are then manually implemented using the Electric [37] layout editor. The physical layout is then simulated with COSMOS [4] to check for layout errors. Cooperating sets of state machine cells are interconnected to form larger modules, integrating clocked datapath logic when necessary.

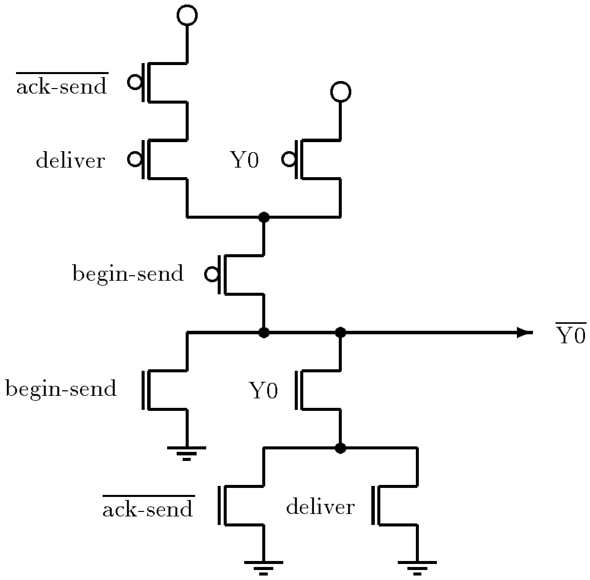


Figure 11: Complex CMOS Gate for sbuf-send-ctl Y0

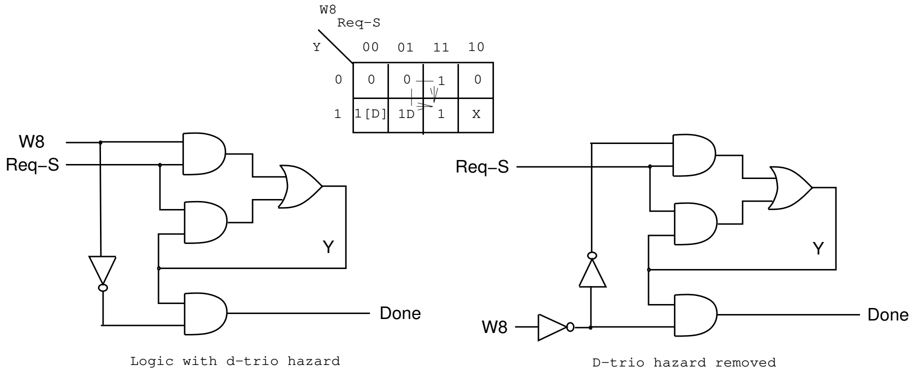
6.5 D-Trio Hazards, Assumptions, and Possible Elimination

Figure 12 shows a static **d-trio** or nonessential function hazard which is found in some of the state machines produced by MEAT. D-trio hazards are fundamental and cannot be removed in every case, but they will be detected by the verifier In this cases the hazard occurs because the input burst resulted in an internal state change while the output burst contained no transition for the *Done* signal. The d-trio hazard in this example can produce a static 1-hazard on the *Done* signal. The input burst is perceived by the *Done* output logic *after* the state change burst thereby creating the hazard.

The $W8$ signal of the logic with the d-trio also contains an isochronous fork. If we ignore the potential threshold deviations then timing analysis shows that the physical behavior will not exhibit the hazard. However, this circuit cannot be included in a system without analyzing the driver, load, and stray capacitance on the $W8$ input or errors will result.

By modifying the trigger logic in the Sendr-Done state machine shown in Figure 12, we can both eliminate the d-trio hazard and the external isochronous fork. This incurs *no* performance penalty. The $\overline{W8}$ signal to the *Done* logic remains delayed by a single inverter, while the $W8$ signal to the state logic becomes doubly inverted rather than fed directly into the logic from the input.

The double inversion has the effect enforcing correct *sequencing* of the order of arrival of the $\overline{W8}$ signal to the *Done* logic. Transitions on $\overline{W8}$ will always be



Logic with d-trio hazard

D-trio hazard removed

Figure 12: Hazard removal from "Sendr-Done" state machine

perceived by the *Done* logic before changes in the state variable, resulting in hazard-free circuit operation. Transitions are ordered such that the assertion of the state variable is not critical to the performance of the circuit, so the double inversion of $W8$ into the state logic does not negatively impact performance.

7 MEAT and the Post Office in Retrospect

The first silicon version of the Post Office chip was only partly operational. It was also approximately only half as fast as expected. As the causes for the performance and functional problems were uncovered several methodological errors became painfully clear. In particular our focus and CAD development (e.g. MEAT) had centered on controller circuits since the Post Office is a control intensive chip. There are 94 different AFSM's, many of which are replicated six times, once for each of the six external ports. To some extent the modular composability of asynchronous circuit modules had led us astray. Our focus should have included the overall design. The complex communicating AFSM control path of the Post Office required proper coordination of multiple AFSM controller modules. We discovered some functional errors in the first silicon that were the result of design errors involving more than 2 AFSM controllers. There was no problem with pairwise AFSM combinations. The difficulty came with the need to sustain a high level of internal concurrency involving multiple AFSM's.

The other problem was the somewhat cavalier attitude taken towards datapath circuits. Since MEAT could only generate optimized transistor schematics for controller circuits, all the datapath design and all the controller layout were done manually. This was a time consuming exercise and the resulting floor plan was the major culprit in terms of the factor of two *performance surprise.* In particular, the spine bus was so long that it became the major performance bottleneck in the chip. Since modification of the design to improve the floor plan would require significant manual layout time, and the project deadline was fast approaching, we had no option but to fix the functional problems and live with less performance.

While these problems were certainly real, there were several encouraging aspects of the design and the methodology. Since the Post Office design instigated the development of the MEAT tools, approximately a fifth of the Post Office control path design was done manually prior to the development of MEAT. The remaining four-fifths was implemented using the MEAT tools. The automated part of the design took one quarter of the total design time and was virtually error free. The nature of these errors are the basis for the subsequent sections of this chapter. Since it is based on traditional finite state machine control, the design style has proven to be a very natural transition for existing hardware designers. The MEAT synthesis techniques have generated compact high-performance circuits that work, and the complexity of the synthesis algorithms have proven to be tractable for large designs.

The present status of the Post Office is that second silicon is *adequately* operational. In testing the second silicon we were able to generate enough concurrency in the chip to create power starvation in the core which can cause failure. The cause of this problem was inadequate power distribution wire sizes. Power analysis had been performed for a 3 micron process and we had naively assumed that a fresh analysis was unnecessary when the process scaled to 1.2 microns. Fortunately a simple software modification prevents this extremely rare situation from occuring in the Mayfly systems. Packets can still flow concurrently on all seven ports of the Post Office. Each of the external ports permits a byte of information to be transmitted every 50 nanoseconds. Each packet is 144 bytes and hence takes 7.2 microseconds to transmit. The internal port is a 32-bit word-wide data path which is capable of sending or receiving a word every 50 nanoseconds. The current CPU that we use is only a 16 MHz device using a multiplexed address and data bus. The CPU cannot keep up with the Post Office and therefore actual packet loads and unloads take approximately 10 microseconds.

The Post Office was fabricated using MOSIS revision 6 design rules in a 1.2 micron CMOS process. The circuit contains 300,000 transistors and has an area of 11 × 8.3 mm. Although this is a research circuit, it is of sufficient complexity to demonstrate the validity of the design style for realistic systems component design.

The goal in the development of the MEAT tool was to generate fast, compact, efficient circuits. Showing the excellent performance that can be achieved with asynchronous designs is an important part of forwarding this technology to the general circuit design community. While experienced asynchronous designers understand that there are more benefits in the asynchronous approach than speed, it is clear that the dominant metric in evaluating circuit design styles in the commercial arena is performance. Our Post Office design was no exception; as long as the circuit was fast nobody cared how we did it except us. We view this as a sad reality, since it restricts the impact of the conceptual elegance of asynchronous circuits to the academic community.

The problems that we encountered in the Post Office design proved that

MEAT, like any CAD tool, is incomplete. The back-end only produces schematics. Manual layout is prohibitively time consuming. Some form of automatic layout is necessary unless we abandon the complex gate approach in order take advantage of standard cell and technology mapping approaches. However, automatic layout is a difficult task and should also include automatically sized transistors for the performance needs of the design. Using standard cells will result in some lost performance but makes the synthesis task easier. It was also painfully clear that we needed to make a connection to existing CAD tool frameworks to permit synthesis of datapath as well as control path circuits, and to allow behavioral simulation to be performed on the entire design.

These realizations plus the fact that the verifier occasionally found hazards in the design were the motivation for a subsequent asynchronous synthesis CAD effort called **STETSON**. Steve Nowick's efforts to modify David Dill's asynchronous verifier [12] created a realization that our flow table based state and logic minimization methods in MEAT were not quite correct for burst-mode operation. Changing from SIC to even the restricted MIC behavior of burst-mode required further thought. In order to pursue this work, a research project at the HP Stanford Science Center was set up to provide a collaborative effort between the asynchronous research efforts at Stanford and those in the Mayfly group at HP laboratories. HP had recently made a corporate decision to move to the Mentor GDT IC tool platform and since HP was sponsoring the project, the obvious choice was to improve on the MEAT ideas in a way that was both correct and could be integrated into the Mentor GDT framework which is called Falcon.

The STETSON work was primarily done by five people. Bill Coates, one of the Mayfly project members, did the work on integrating the tools into the Mentor GDT framework and developed an improved version of the state machine minimization routines. Steve Nowick developed a set of criteria for minimizing burst-mode AFSM's which permitted correct hazard free logic minimization for unclocked burst-mode AFSM's as well as an alternative method which uses state holding latches for AFSM implementation. The latter formed the basis for his Ph.D. dissertation [34] called the *locally-clocked* method. Ken Yun's dissertation [50] developed a third synthesis path called the *3D* method. Since the problem of generating a quality custom layout automatically was extremely difficult and since our corporate sponsors were increasingly moving towards standard cell and gate array design methods, we decided to develop the STETSON methods to generate standard cell implementations. A key problem in doing this was the ability to do hazard non-increasing technology mapping onto the standard cell library. The solution to this problem became the Ph.D. dissertation of Polly Siegel [39].

The project which drove the development of the STETSON tool kit was a chip which we called **ABCS**. ABCS is a low-power infrared communications receiver intended for hand held digital assistant type devices. Alan Marshall, an HP laboratories member from Bristol England, came to Stanford for two years to participate in the effort. He was the main designer of the ABCS chip and his efforts catalyzed much of the STETSON development. The author and

Prof. David Dill of Stanford were certainly involved, but the day to day efforts and the bulk of the credit for STETSON belongs to Bill Coates, Steve Nowick, Alan Marshall, Polly Siegel, and Ken Yun.

The remainder of this chapter is devoted to the description of their efforts.

8 STETSON AFSM Synthesis

The information in this section is based on a more detailed paper by Bill Coates and Steve Nowick [31]. The goal of this effort was to produce hazard free gate level implementations for burst-mode AFSM specifications that did not require the post synthesis verification step required by designs synthesized by the MEAT tools.

While the STETSON tool set also permits the 3D [50] and locally clocked [34] options, the unclocked method described here has several advantages. Unlike the locally clocked method, there is no need for local clock generation logic or transparent latches. Unlike the 3D method, a state transition requires no multi-stepping through intermediate states. Additionally this method does not rely on the addition of delays to the critical path as in other MIC methods [22, 14]. The result is improved performance.

The AFSM is specified by the same burst-mode AFSM description used in MEAT, although the syntax is slightly different for STETSON. There are two restrictions on this specification:

- **Maximal Set Property:** no exit burst from a state can be a proper subset of an exit burst from the same state.
- **Unique Entry Point Property:** A given state must always be entered with the same set of input values.

These two properties were also true of MEAT AFSM's and are necessary to prevent ambiguity in the AFSM's behavior. If the maximal set property were not present then the AFSM would be presented with the need to react to part of a burst rather than waiting for a larger burst. Such a distinction is silly in an asynchronous environment. If the unique entry point were not observed then sometimes a state would be waiting for input variable A to go high, while another time it would wait for it to go low.

The first step of the synthesis is to transform the burst-mode specification into an unminimized flow table. For example the specification shown in Figure 13 is transformed into the flow table shown in Figure 14.

An output of nil in the specification indicates no change to the current outputs. For inputs a + after a variable name indicates a low to high transition for that input. Similarly a – indicates a high to low transition. Since the specification tracks every transition of every input variable, the choice of four or two cycle signaling protocols does not influence the methodology and becomes simply a designer choice. Hence mixing two and four cycle protocols is a simple task, and the method of choice can be suited to the constraints of the design. It is sometimes the case that two-cycle protocols are faster, and it is often the

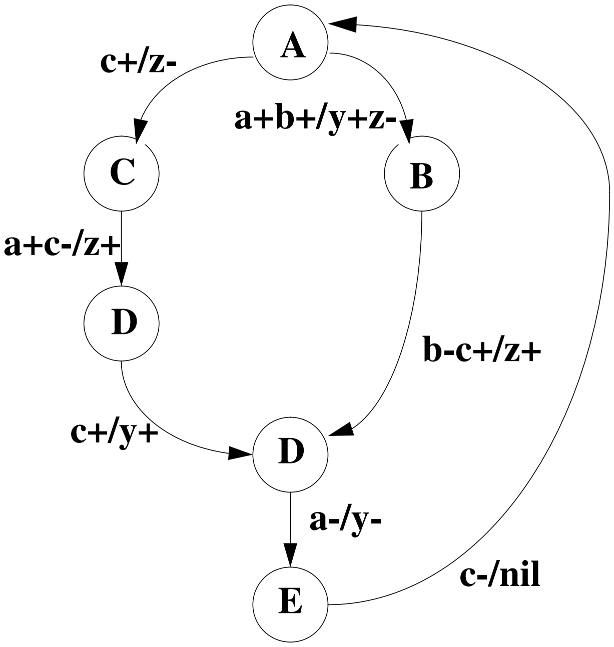


Figure 13: Sample Burst-Mode Specification

case that four-cycle protocols require less logic to implement. The result, as was true for the Post Office, is that it is often appropriate to mix the two styles.

In the flow table, horizontal movement indicates a change in the input state and a vertical movement indicates a change in the state of the AFSM. The combination of the two is the total state of the machine. Note that one aspect of the burst-mode flow table is the covering of all possible *trajectories* from an initial state to a final state target of some burst change. This is illustrated from state A where two exit possibilities exist. The **c+** exit is a simple SIC transition, but the **a+b+** exit represents a true burst where input concurrency can occur. This means that either a can go high and then b, or vice versa, or they can go high with so little separation as to appear to be changing concurrently from the perspective of the circuit. Until the burst is complete, the flow table indicates that no state change will occur.

In classical methods such as those used by MEAT, states are then merged based on compatibility relations in the columns of the flow table. States are compatible if they are output compatible, and if they are output compatible then their next state sets must be compatible as well. These methods are well described in Unger [46]. The problem is that certain legal state minimizations can result in a reduced flow table which cannot be implemented as a hazard free logic circuit. Nowick and Coates [31] have shown that for the unminimized flow table there is a hazard free logic implementation, and that using the classical methods certain state combinations may create a reduced flow table for

Next State, Outputs=yz

inputs = abc

Current State	000	001	011	010	110	111	101	100
A	A, 01	C, 00	—	A, 01	B, 10	—	—	A, 01
B	—	—	—	—	B, 10	B, 10	E, 11	B, 10
C	C, 00	C, 00	—	—	—	—	C, 00	D, 01
D	—	—	—	—	—	—	E, 11	D, 01
E	—	F, 01	—	—	—	—	E, 11	—
F	A, 01	F, 01	—	—	—	—	—	—

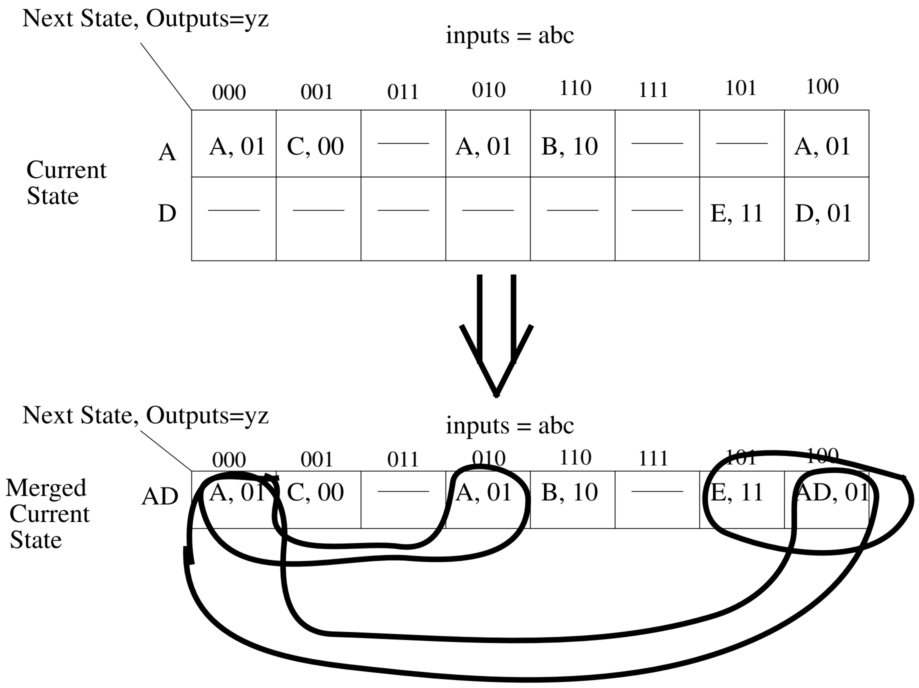
Figure 14: Sample Unminimized Flow Table

which no hazard free cover exists. These situations caused the need for post minimization verification in the MEAT tools.

As an example, consider rows A and D in Figure 14. All columns are output compatible, and the only implied next state compatibility is in column labeled **100** which indicates that A and D are compatible if A and D are compatible. This is always true and hence at first glance states A and D could be merged to reduce the AFSM and result in a column shown in Figure 15.

The Karnaugh map covers for the z output variable are shown in the resulting merged state row for AD. Output z has a 1 to 0 transition after input burst **a+b+** and a 1 to 1 transition for input burst **c+**. The result is that any product term which intersects the **c+** transition also intersects the **a+b+** product term. Hence there is a logic hazard for the z output. The problem becomes even more complex for the next state logic. This is a curious result in that although a hazard free solution exists prior to minimization no solution may exist after classical minimization.

Nowick studied the effects of burst-mode trajectories and came up with a set of state merging constraints that must be added to the classical output and next state compatibility constraints. The result is a new state algorithm (called dhf-compatibility) which guarantees that a hazard free two-level logic sum of products implementation exists. The key to this algorithm is to realize that burst-mode trajectories from initial sub-cube to the final destination fill a binary n-cube area of the boolean space. This area is easy to visualize using the traditional Karnaugh map models. In short if two spaces (call them S1 and S2) intersect and are dhf-compatible then they must satisfy the following constraints:



	000	001	011	010	110	111	101	100
A	A, 01	C, 00	——	A, 01	B, 10	——	——	A, 01
D	——	——	——	——	——	——	E, 11	D, 01

	000	001	011	010	110	111	101	100
AD	A, 01	C, 00	——	A, 01	B, 10	——	E, 11	AD, 01

Figure 15: Sample Unminimized Flow Table

- If S1 and S2 are both 1 to 0 transitions, then they must share a common initial point, OR they must intersect at their destination points, OR they must not intersect at any point.

- If S1 is a 1 to 0 transition, and S2 is a 1 to 1 transitions, then S2's initial point may lie in the S1 space but the S1 destination point may not lie in the S2 space.

Detailed arguments of why these constraints are necessary can be found in [31]. Experimental results indicate that the output latency and the logic complexity of the unclocked method are comparable to that of the 3D method, but the next-state latency of the unclocked method is an improvement over the inherent multi-stepping to the next state model inherent to the 3D method. For applications where cycle time is critical the unclocked methods have an advantage, whereas there is little difference in the unclocked and 3D methods for applications whose performance is dominated by output latency.

9 Hazard Non-Increasing Technology Mapping

Once an AFSM has been specified and transformed into a set of equations which implement the specification, we essentially have a netlist for the implementation. The next step, given the standard cell bias for the STETSON tools, is to map this implementation onto the standard cell library. A detailed account of this approach can be found in [39]. Automatic technology mapping techniques have been employed for synchronous design styles [20, 38, 23]. These algorithms allow translation of a technology-independent logic description into a technology-dependent implementation. However, these techniques by themselves are not suitable for asynchronous design styles since they do not take hazards into account.

In particular we were interested in a *generalized fundamental-mode* asynchronous design style [33], since it separates the combinational portions of the design from the sequential portions, similar to many synchronous design styles. Each step of algorithmic technology mapping is examined for its influence on the hazard behavior of the modified network. Our approach was to modify an existing synchronous technology mapper, CERES, to adapt it to work for generalized fundamental-mode designs.

Our burst-mode design style extends the fundamental mode assumption imposed by Mealy and Moore machines to allow multiple-input change bursts in a particular state. The synthesis methods that incorporate this burst-mode or *generalized* fundamental-mode design style produce logic under the assumption that a burst of input changes can occur, in any order, and the outputs and feedback variables of the combinational portion will settle before the next set of input changes are applied. As in the case of single-input change fundamental-mode, no hazards can be tolerated during the input bursts. However, both single-input change hazards and multi-input change hazards must be considered for the technology mapping step.

There are two basic classes of hazards: *function* and *logic* hazards. Function hazards are a property of the logic function and can only be eliminated through appropriate placement of delay elements, whereas logic hazards are purely a property of the implementation. If a network has a function hazard for a given transition, then it cannot have a corresponding logic hazard for that same transition.

Within the two classes of combinational hazards, we can have *static* or *dynamic* hazards. Given that a transition is being made between two points I_1 and I_2 in the input space $\{0,1\}^n$, then static hazards apply to transitions where $f(I_1) = f(I_2)$, and dynamic hazards apply to cases where $f(I_1) \neq f(I_2)$. Within the class of logic hazards, we will further distinguish between *single-input change* (s.i.c.) hazards and *multi-input change* (m.i.c.) hazards.

Static logic hazards will occur whenever the transition is not properly covered by a single gate; that is, whenever the implementation does not contain a single gate that maintains the output value throughout the input transition. Dynamic hazards are applicable to both single-input change and multi-input change situations. A dynamic hazard occurs when, during an expected $0 \rightarrow 1$ ($1 \rightarrow 0$) transition of the output, a $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ ($1 \rightarrow 0 \rightarrow 1 \rightarrow 0$) transition occurs. For single-input change conditions, this corresponds to a situation where a literal and its complement fan-out to several paths. For multi-input change conditions, a dynamic hazard can occur when a single gate is turned on momentarily during the transition. For the generalized fundamental-mode asynchronous design style, all logic hazards are of interest, and we must make sure that new logic hazards are not introduced during the technology mapping operation.

The technology mapping approach taken by algorithmic technology mappers typically divide the problem into three major steps: decomposition, partitioning, and matching and covering. First, the initial network, which is represented as a directed acyclic graph (DAG), is decomposed into a multi-level network composed of canonical two-input, single-output gates. Next, the circuit is partitioned into sets of single-output *cones* of logic, where a cone of logic represents the gate-level network as traced from a given output to the inputs. Each cone is then treated independently by the mapper. All possible matches to library elements are then found for subnetworks within each logic cone. Finally, an optimal set of matching library elements is selected from the set of matches to realize the network.

The decomposition step partitions the network into an equivalent network composed of 2-input, 1-output base functions. This process can be performed by recursively applying DeMorgan's rule and the associative law to the network. Both operations have been shown to be hazard-preserving for all logic hazards [46]. Thus, the modified network composed of 2-input, 1-output gates has identical hazard behavior to that of the original network.

Partitioning breaks the decomposed network at points of multiple fanout into single-output cones of logic. This is a heuristic simplification that is required to convert a multi-output logic network into a collection of single-output cones of logic, so that simpler algorithms can be employed to find the best cover for the network [20, 23]. Each subnetwork is then mapped separately by the matching and covering algorithm described in the next subsection. Given that we start with a hazard free network and preserve this behavior (within the partitions) in the covering step, the partitioning step cannot alter the hazard behavior of the network.

The matching and covering steps involve identifying equivalence between a subnetwork and a library element, and replacing that subnetwork with the equivalent library element. This step must not introduce any new hazards into the design.

Different algorithms are used during the matching and covering step. Keutzer and Rudell [20, 38] use tree pattern matching techniques for matching elements within the library to portions of the circuit. In MIS, a complete set of patterns composed of different decompositions of two-input, one-output gates is generated for each library element. Standard pattern matching techniques are used to compare the library element with a portion of the network to be mapped. As long as the patterns themselves do not have hazards, these techniques will work to preserve the hazard behavior of the circuit, provided that the initial pattern decomposition is hazard-preserving and that the library elements do not have hazards. However, if a hazardous library element is selected, then the hazard behavior of the subcircuit must be examined before the match is accepted.

Some mappers, such as CERES, use Boolean techniques to detect equivalent networks. These techniques decouple the structure of the subnetwork from the matching process, which means we cannot reason about the transformations of one subnetwork into the other. However, we can build on theorems presented by Unger in [46] to show that if we replace a portion of the circuit with an equivalent circuit with similar hazard behavior, the resulting circuit is still hazard free for the transitions of interest.

Siegel [39] has shown that if we start with a set of hazard free library elements, the covering step will not introduce new hazards and the resulting cover will introduce no new hazards. Additionally, if we have library elements that contain hazards, we need to make sure that the hazards they contain are a subset of those in the portion of the network that is being matched.

Since the STETSON synthesizers (unclocked, locally clocked, and 3D) all generate hazard free implementations for burst-mode inputs it is then the mappers job not to introduce any new hazards. In order to do this, the library must be augmented with hazard information. The matching step must then check to see if the hazards contained in a candidate library cell are subset of the hazards contained in the logic equations. If this is true then a match succeeds and the final implementation has does not contain added hazards and hence is hazard free for burst-mode transitions of interest.

The functionality of each library element is expressed in Boolean factored form (BFF). We use the BFF expression as an accurate and convenient representation for both the functionality and structure of the particular library element. This BFF expression is then analyzed for logic hazards when the library is read into the mapper, and the logic hazard behavior of each library element is added as an annotation to the library element for later use during the matching phase.

The matching algorithm must now be modified to take the logic hazard information into account. If a hazardous library element is selected as a match for a particular subcircuit, then the subcircuit of interest must be examined to see if the same logic hazards exist. Note that we are not interested in the function hazard behavior of either the library element or the subnetwork, since function hazards are purely a property of the function itself and thus are independent of implementation.

The procedure for doing the comparison is much easier than the initial computation of the hazards for each library element, since we already know the transitions of interest. For each logic hazard in the library element, we must look at the subcircuit to see if the same logic hazard exists. As soon as we find a hazardous transition in the library element that is not in the subcircuit, we can stop, since that library element cannot be used safely. At this point the library element is eliminated from consideration for this match.

We examined a typical commercial standard cell library to see how many elements contained logic hazards. Only the multiplexers, which represented a small fraction of the library, contained hazards. So most matching elements will be logic hazard-free and the normal synchronous algorithms will be used with negligible overhead.

Table 1 shows the (logic) hazardous elements that are present in several libraries. The LSI and CMOS3 libraries are commercial ASIC libraries. The GDT library is a standard-cell library produced specifically for a particular chip, and includes many complex and-or-invert gates. (Note that many of these elements had several instances with different drive capability but only one is shown for brevity.) For these libraries, the only library elements with logic hazards were multiplexors.

Lib	**Hazardous Elements**	**#**	**Total Elements**	**% Hazardous Elements**
LSI9K	Muxes	12	86	14%
CMOS3	Muxes	1	30	3%
GDT	none	0	72	0%

Table 1: Libraries and their hazardous elements

The remaining problems we must solve then are how to efficiently characterize the logic hazard behavior of the library elements, and how to easily determine whether the subcircuit has the same logic hazards. We address these problems in the next section.

Table 2 shows mapping results and mapping runtimes for some asynchronous benchmark circuits. The area costs are relative to the particular library.

The SCSI design was mapped from logic equations produced by the locally clocked method [32]. The ABCS design is a part of the control logic for the infrared communications chip which is and was mapped from equations generated using the 3D method.

Design	**Library**					
	CMOS3			**LSI**		
	CPU	**Delay**	**Area**	**CPU**	**Delay**	**Area**
chu-ad-opt	.6s	24ns	152	3.7s	2.1ns	9
dme-fast-opt	.8s	11.8ns	232	3.8s	1.7ns	13
dme-fast	.6s	11ns	136	3.7s	1.7ns	9
dme-opt	.7s	22.4ns	184	3.9s	2ns	10
dme	.6s	17.8ns	168	3.7s	1.9ns	10
oscsi-ctrl	10.6s	96.5ns	3552	16.1s	7.3ns	172
pe-send-ifc	2.3s	47.2ns	864	5.3s	3.5ns	40
vanbek-opt	.6s	19.5ns	144	3.7s	2.6ns	9
dean-ctrl	33.6s	126ns	11320	43.5s	10.3ns	565
scsi	20.7s	95ns	6888	27.9s	7.2ns	330
abcs	9s	74.7ns	3288	13.3s	6.8ns	168

Table 2: Mapping results for synchronous vs asynchronous mapper (Depth of 5). Benchmarks were run on a DEC 5000.

10 STETSON and ABCS

10.1 Control Path Synthesis

Figure 16 shows STETSON's tool flow options for synthesizing controller circuits. The oval shape at the beginning of the process is implemented by a program written by Bill Coates called **bm-check**. This program takes a textual burst-mode controller specification and checks it to insure that the specification meets the requirements for a legal burst-mode specification, namely adheres to the maximal set and unique entry point properties. There are a number of switches that can be set to control which output files should be generated. Bm-check can generate behavioral VHDL or Verilog code, an extensive set of burst-mode test vectors to exercise all of the paths in the AFSM, or input files for any of the three synthesis methods.

The next step is to create minimized state machines from any or all of the AFSM synthesizers, namely the unclocked, locally clocked or 3D methods. The result of this step is a set of hazard free but unminimized logic equations for the implementation. The asynchronous logic minimizer is then used to produce hazard free minimal equations. The asynchronous logic minimizer a constrained variant of the familiar Quine-McCluskey algorithm which has also been optimized for the don't care dominant nature of burst-mode specified AFSM's. The logic produced by the asynchronous minimizer is typically larger than that produced by an equivalent synchronous minimizer due to the need to provide redundant covers in the logic in order to avoid hazards. However in the large number of cases that we have examined, the overhead with respect to the number of product terms has been no more than 6%. For a large system chip, the area spent on control circuitry is typically a small fraction of the total chip area. We view this 6% area overhead for control circuits as being a negligible cost given the modular composition, circuit reuse, and incremental improvement benefits of asynchronous systems.

The technology mapper then creates a technology dependent netlist using the cells in the cell library. At this point the Mentor AutoCells package performs the place and routing chores to produce the final implementation. Switch level simulation and VHDL simulation results can then be compared using the same set of test vectors to validate that the implementation and the specified behavior are consistent. This comparison step was useful in early designs when we did not trust our toolset implementation to be correct. However, for the last year, we have not encountered a single instance where the comparison did not indicate a match. However, considering the cost in terms of time and dollars spent per processing run, we continue to run the match path.

10.2 Datapath Synthesis

Our design style views datapath circuits as they are viewed in synchronous designs. We typically use bundled data and either two or four cycle signaling protocols and use model delays associated with the datapath elements to provide completion signals. Hence datapath synthesis is performed by writing

VHDL models, which are then synthesized into standard cell circuit blocks by the Mentor VHDL compiler. Behavioral VHDL models can be used in concert with the synthesized AFSM models generated by bm-check to run simulations on system wide behavior of multiple control and data path modules.

Model delays are determined using standard timing analysis tools. The margins that we use are typically smaller than those employed by synchronous designers since issues like clock skew do not have to be accounted for in our asynchronous design style which uses point to point control signals from controlling AFSM modules to their respective controlled datapath modules. Most companies build their own set of datapath generators which are used to generate datapath blocks like memories, register files, ALU's, etc. and these tools can be used inside the STETSON framework. All that is needed is to run the timing analyzer on the result in order to determine the model delay and then generate an inverter chain that corresponds to this delay into the synthesized module.

10.3 ABCS

The STETSON toolset was used to synthesize a low-power infrared receiver chip suitable for hand held digital assistant style applications. Prior to submission of the ABCS design, several small MOSIS *tinychip* test fragments were designed and submitted to check the validity of our synthesis tools. They all worked as expected on first silicon. Given our Post Office experience this came as a pleasant surprise.

The biggest problems in the development of the ABCS design came from two basic sources:

1. The design and the tool developments were concurrent activities. Hence much of the design testing and modeling had to wait or even drive some of the tool development activities which significantly extended the design cycle time.

2. The desire to produce the ABCS design in a mainstream fashion meant that we needed to produce a VHDL specification of the data and control path elements. The Mentor VHDL compiler has some serious performance flaws. The result was an extremely slow and painful development and modeling cycle.

Nonetheless, the ABCS chip was submitted and ran on first silicon.

11 Conclusions

The work reported in this chapter was done over a five or six year period. We started this effort somewhat naively thinking that automating what we had been doing by hand for decades would be reasonably straight-forward task. We were totally wrong. The evolution of the burst-mode idea, initially important from the standpoint of performance, eventually required that we rethink the entire methodology. MEAT was the first step, and while it drastically cut the design cycle time, the need for post verification of the synthesis and the need for manual layout were critical flaws. Another critical flaw was that while control path circuits are often the most complex and error prone aspects of a design, the datapath logic consumes the most area on the final chip, and MEAT had no support for datapath synthesis. The final problem with MEAT was that we still had no way to perform behavioral analysis on the total design.

The STETSON effort was the logical next step. It was fortunate that we were able to conduct this effort in an academic setting. The mere fact that multiple students were involved provided options that would not have been pursued otherwise. The result is a toolset that can be used for large scale designs, uses rather than ignores commercial CAD capabilities, and permits synchronous designers to maintain most of their familiar design methods.

As with any CAD effort, the STETSON design has many aspects which can be improved. There is considerable designer art required to decompose a large control path specification into a set of communicating AFSM's which achieve a reasonable balance of cost and performance. Similarly, our method of always using model delays for the datapath elements rather than using more directly asynchronous methods is not appropriate for designs which require more robust behavior. We view these areas as opportunities for future research.

12 Acknowledgments

It should be clear from the text of this chapter that the author is merely acting as a scribe for the efforts of a number of people who made the MEAT and STETSON tools a reality. The same is true for the Post Office and ABCS chips. The author started the MEAT effort and produced the logic minimization methods that were used in MEAT but these were subsequently modified by Steve Nowick. Bill Coates did all of the work on the state machine minimization part of MEAT and continued this effort with help from Steve Nowick to produce the unclocked state machine synthesis method in the STETSON tools. Ken Yun provided the 3D option in STETSON and Steve Nowick provided the locally clocked option. Polly Siegel developed the methods and implementation for doing the hazard non-increasing technology mapper for STETSON. Ken Stevens was the primary designer of the Post Office with help from Bill Coates. Alan Marshall was the designer of the ABCS chip. The entire STETSON effort is indebted to the guidance and insights of David Dill.

The activities reported here were truly a group effort. The author played only a small part, however as scribe for these efforts he assumes all responsibility for errors in the transcription.

Finally, we owe a great deal of gratitude to Hewlett Packard Laboratories, and in particular Dick Lampman and Stephen Rosenberg, whose support made all of this possible.

References

[1] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[2] Erik Brunvand and Robert Sproull. Translating Concurrent Programs into Delay-Insensitive Circuits. In *IEEE International Conference on Computer Aided Design: Digest of Technical Papers*, pages 262–265. IEEE Computer Society Press, 1989.

[3] Steven M. Burns and Alain J. Martin. *The Fusion of Hardware Design and Verification*, chapter Synthesis of Self-Timed Circuits by Program Transformation, pages 99–116. Elsevier Science Publishers, 1988.

[4] Carnegie-Mellon University. *User's Guide to COSMOS.*

[5] Tam-Anh Chu. On the models for designing VLSI asynchronous digital systems. Technical Report MIT-LCS-TR-393, MIT, 1987.

[6] Henry Y. H. Chuang and Santanu Das. Synthesis of multiple-input change asynchronous machines using controlled excitation and flip-flops. *IEEE Transactions on Computers*, C-22(12):1103–1109, December 1973.

[7] William S. Coates. "The Design of an Instruction Stream Memory Subsystem". Master's thesis, University of Calgary, December 1985.

[8] A. Davis. "The Mayfly Parallel Processing System". Technical Report HPL-SAL-89-22, Hewlett-Packard Company, March 1989.

[9] A. L. Davis. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. Technical Report UUCS-77-113, University of Utah, Computer Science Dept, 1977.

[10] A. L. Davis. Mayfly: A General-Purpose, Scalable, Parallel Processing Architecture. *Lisp and Symbolic Computation*, 5(1/2):7–47, May 1992.

[11] A. L. Davis, B. Coates, R. Hodgson, R. Schediwy, and K. Stevens. Mayfly System Hardware. Technical Report HPL-SAL-89-23, Hewlett-Packard Laboratories, April 1989.

[12] David Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. An ACM Distinguished Dissertation.* MIT Press, 1989.

[13] William I. Fletcher. *An Engineering Approach to Digital Design.* Prentice-Hall, 1980.

[14] A. D. Friedman and P. R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6):559–566, June 1968.

[15] R. M. Fujimoto. *VLSI Communication Components for Multicomputer Networks.* PhD thesis, Univ. of California at Berkeley, August 1983.

[16] A Grasselli and F. Luccio. A Method for Minimizing the Number of Internal States of Incompletely Specified Sequential Networks. *IEEE TEC*, June 1965.

[17] A. B. Hayes. Stored State Asynchronous Sequential Circuits. *IEEE Transactions on Computers*, C-30(8), August 1981.

[18] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, Aug 1978.

[19] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[20] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *24th Design Automation Conference*, pages 341–347. IEEE/ACM, 1987.

[21] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of Verifiably Hazard-Free Asynchronous Control Circuits. In Carlo H. Sequin, editor, *Proceedings of the 1991 UC Santa Cruz Conference on Advanced Research in VLSI.* MIT Press, 1991.

[22] G. Mago. Realization methods for asynchronous sequential circuits. *IEEE Transactions on Computers*, C-20(3):290–297, March 1971.

[23] F. Mailhot. *Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Optimizations.* PhD thesis, Stanford University, 1991.

[24] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The Design of an Asynchronous Microprocessor. In C.L. Seitz, editor, *Advanced Reserach in VLSI: Proceeedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[25] Alain Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(1):226–234, 1986.

[26] Alain Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[27] C. Mead and L. Conway. *Introduction to VLSI Systems.* McGraw-Hill, 1979. Chapter 7.

[28] Teresa Meng. *Synchronization Design for Digital Systems.* Kluwer Academic, 1990.

[29] R. E. Miller. *Switching Theory, II: Sequential circuits and machines.* Wiley, 1965. Chapter 10.

[30] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of Delay-Insensitive Modules. In Henry Fuchs, editor, *Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[31] S. M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 434–441, October 1994.

[32] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society, 1991.

[33] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society, 1992.

[34] Steve Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993.

[35] Steven M. Nowick and David L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *1991 IEEE International Conference on Computer-Aided Design*. IEEE Computer Society, 1991.

[36] S. S. Patil. Coordination of asynchronous events. Technical Report TR-72, MIT Project MAC, June 1970.

[37] Steven M. Rubin. *Computer Aids for VLSI Design*. VLSI Systems. Addison-Wesley, 1987.

[38] R. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, U. C. Berkeley, April 1989. Memorandum UCB/ERL M89/49.

[39] Polly Siegel. *Technology Mapping for Asynchronous Designs*. PhD thesis, Stanford, January 1995.

[40] Kenneth S. Stevens. The Communications Framework for a Distributed Ensemble Architecture. AI 47, Schlumberger Palo Alto Research, February 1986.

[41] Kenneth S. Stevens, Shane V Robison, and A.L. Davis. "The Post Office – Communication Support for Distributed Ensemble Architectures". In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 160 – 166, May 1986.

[42] I. E. Sutherland, R. F. Sproull, C. E. Molnar, and E. H. Frank. Asynchronous Systems, Volume I. Technical report, Sutherland Sproull and Associates, Palo Alto, CA, January 1985.

[43] Ivan E. Sutherland and Robert F. Sproull. Logical effort: Designing for speed on the back of an envelope. In Carlo H. Sequin, editor, *Proceedings of the 13th Conference on Advanced Research in VLSI*, pages 1–16. UC Santa Cruz, March 1991.

[44] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15:551–560, August 1966.

[45] S. H. Unger. A Building Block Approach to Unclocked Systems. In *Proceedings of the 26th HICSS Conference*, January 1993. To appear.

[46] S.H. Unger. *Asynchronous sequential switching circuits.* Wiley-Interscience, 1969.

[47] C. H. (Kees) van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI.* PhD thesis, Technical University of Eindhoven, May 1992.

[48] Kees van Berkel. Beware the Isochronic Fork. *Integration, the VLSI journal*, 13(2):103–128, 1990.

[49] Peter Vanbekbergen, Francky Catthoor, Gert Goossens, and Hugo De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *International Conference on Computer-Aided Design*. IEEE Computer Society Press, 1990.

[50] K. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the International Workshop on Computer-Aided Design*, pages 576–580, November 1992.

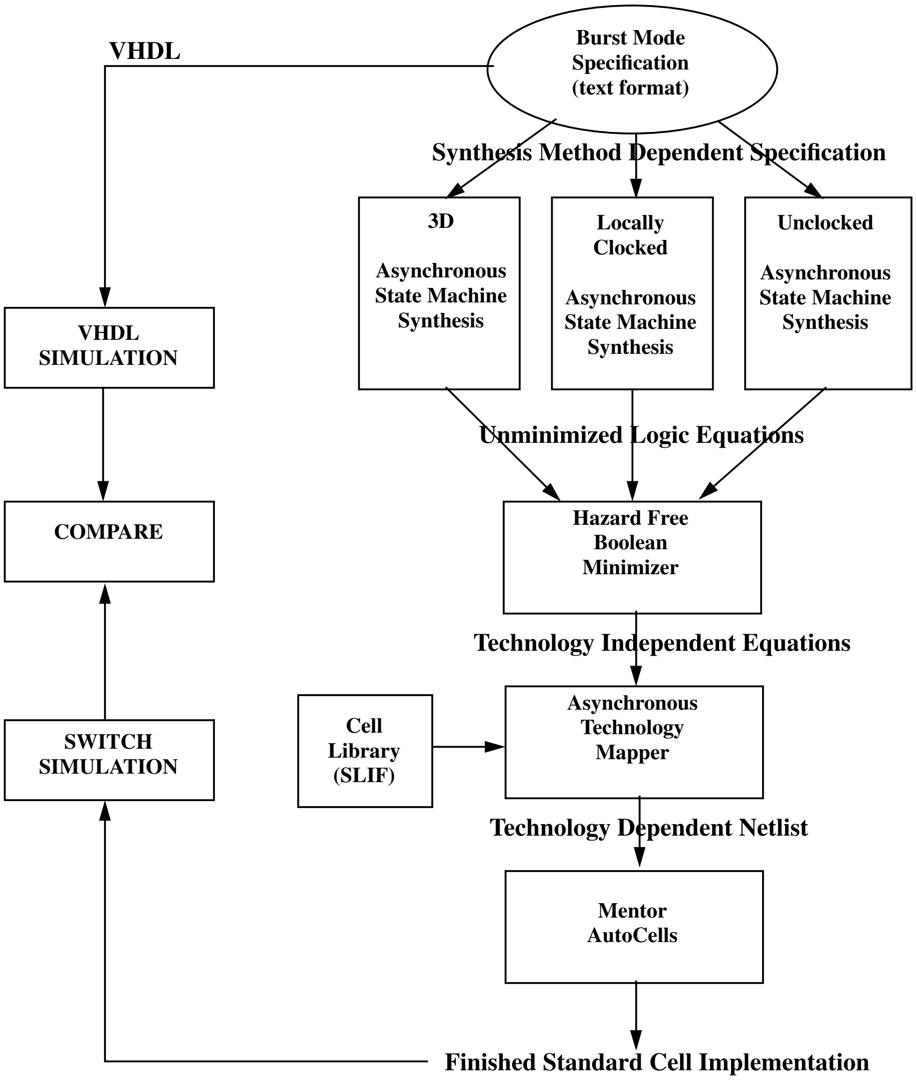


Figure 16: STETSON Tool Flow