Note: Published in Integration Vol. 15, No. 3, November 1993, pp. 341-366

The Post Office Experience: designing a large asynchronous chip

by

Bill Coates
Al Davis
Ken Stevens [1]
Hewlett-Packard Laboratories

September 7, 1995

ABSTRACT

The Post Office is an asynchronous, 300,000 transistor, full-custom CMOS chip designed as the communication co-processor for the Mayfly scalable parallel processor. The Post Office is responsible for all of the physical packet delivery duties: dynamic adaptive routing, congestion avoidance, buffering, and physical deadlock prevention. Since communication latency is a critical influence on message passing multiprocessor system performance, the Post Office had to be designed both for high speed operation, and to function in machine configurations containing widely varying numbers of processing elements. The design consists of a relatively large number of asynchronous finite state machines which operate in parallel to control the various datapath components. Performance requirements led to the development of a design style which permits the design of sequential circuits which operate under a restricted form of multiple input change signalling called *burst-mode*. The Post Office complexity forced us to develop a set of design tools capable of correctly synthesizing transistor circuits from state machine and equation specifications, and capable of verifying the correctness of the resultant circuitry using implementation specific timing assumptions. The paper provides a case study of this design experience.

[1] Presently with the University of Calgary Computer Science Dept.

1 Introduction

The Post Office was designed to support inter-node communication for the Mayfly parallel processing system [7]. 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. This unknown physical extent aspect 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. 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 in today's asynchronous design community. One partition of design styles can be based on the type of asynchronous circuit target: locally clocked [21, 13, 8], delay-insensitive [16, 3, 30, 20], or various forms of *single-* and *multiple-* input change circuits [29]. Yet another distinction could be made on the nature of the control specification: graph based [22, 19, 32, 5], programming language based [16, 3, 30, 2], or finite state machine based [21, 13]. 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 [14, 28]. 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. There is presently little to distinguish the asynchronous and synchronous datapath design styles.

All asynchronous design styles are fundamentally concerned with the synthesis of hazard free circuits. The methods which produce delay-insensitive circuits, while not perfect [17], 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 [16, 3, 30, 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 trouble. C-elements are latches and as such are synchronization points. Too much synchronization reduces parallelism and performance. The finite state machine based design style does not use C-elements, although C-elements are used sparingly in interface circuits such as arbiters.

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 [29]. 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 [29, 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.

As our Post Office state machines got 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 call this tool kit **MEAT** and is described in section 4. During the development of MEAT, we were fortunate to have Steve Nowick spend two summers with us. He incorporated David Dill's verifier [10] into the tool kit, and modified the verifier to accommodate our burst-mode timing model. Steve had considerable influence on our ideas and his locally clocked design style [21] is another outcome of these earlier interactions. We are indebted to Steve for his influence on our design style.

The remainder of this paper presents a functional description of the Post Office chip, a description of the high-level implementation decisions, a brief synopsis of the MEAT tools, a specific AFSM synthesis

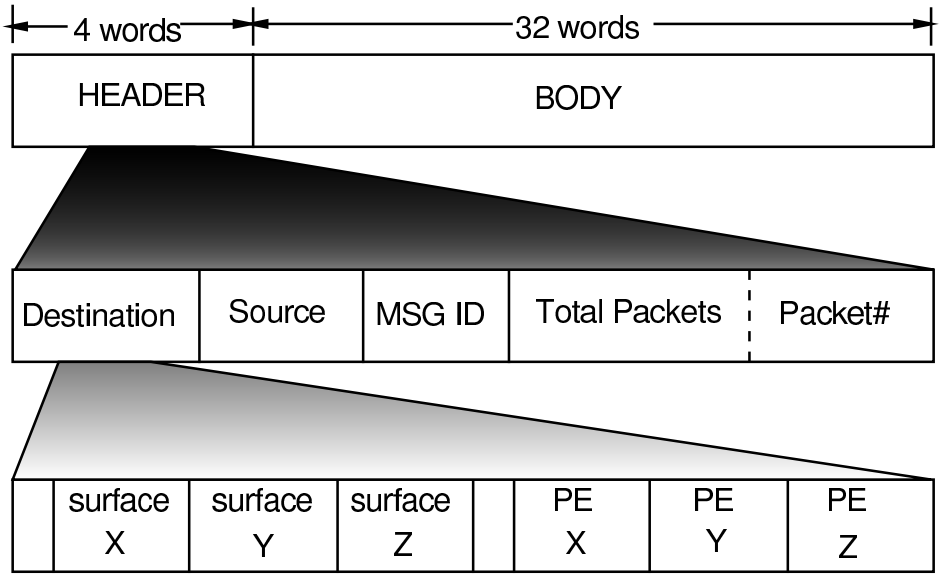

Figure 1: Packet Format

example using the tools, and finally concluding remarks.

2 Post Office Behavioral Description

The Post Office [24, 25] 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 a 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 concurrently active.

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 inherently 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 transfer 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.

The capability to dynamically route packets around congestion and failed components implies that the order of packet arrival may vary from the order in which the packets were sent. The capability to reorder 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 insures 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 live-lock 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 a situation where 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 immediately forwarded to the appropriate destination port. This method is called **virtual cut-through** [12]. 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

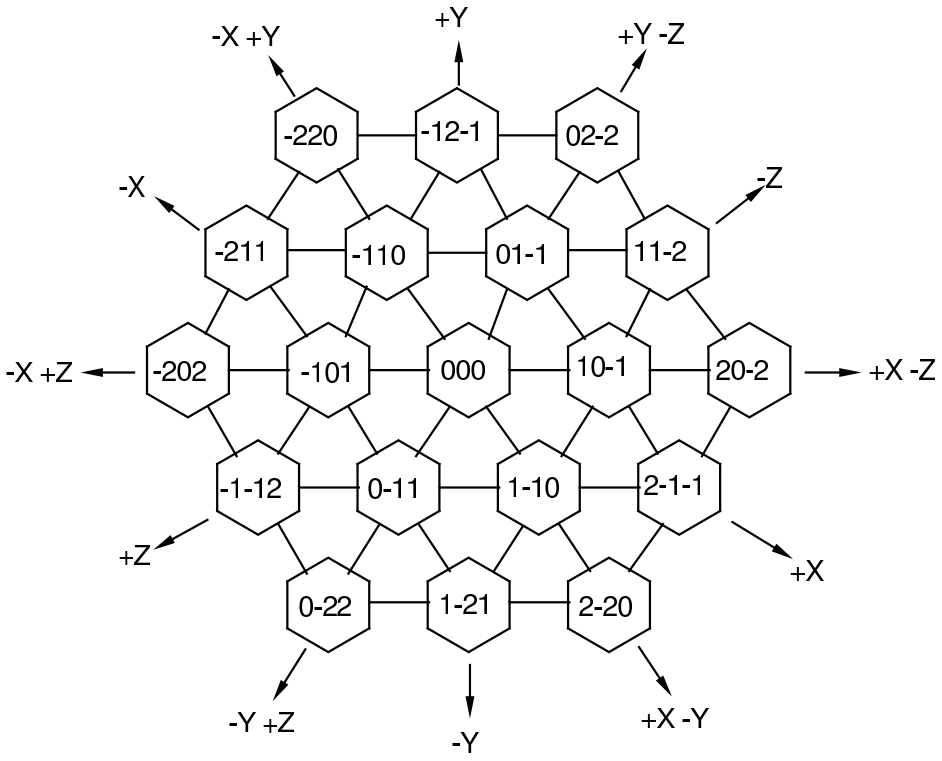

Figure 2: PE Labeling Convention

stagnation count. If a second threshold is exceeded then random routing is also an option. In any case 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 what 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. For example, 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. The table size is quadratically dependent on the number of PEs and would be prohibitive. Therefore the Mayfly Post Office does not use routing tables and simply computes the routing decision using a method that is both simple and fast. 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, if one of the components of the resultant value is greater than n then a wrap line will be needed.

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:

$$2n-1, \left\lfloor \frac{2n-1}{2} \right\rfloor, \left\lceil \frac{2n-1}{2} \right\rceil$$

   Where $\lfloor$ indicates rounding down to the next integer value and $\lceil$ indicates rounding up to the next integer value. For an E–3 surface, this value is <5,2,3>.

3. This normalization vector is then logically end-around right-shifted until the $2n-1$ term is aligned with the highest magnitude term of the unnormalized relative address.

4. The signs of the normalization components are then set to match the signs of the unnormalized relative address components.

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

6. At this point the decision is basically easy. 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, then 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.

3 High-Level Implementation

3.1 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 of these logic blocks are independent agents, 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 any 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 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

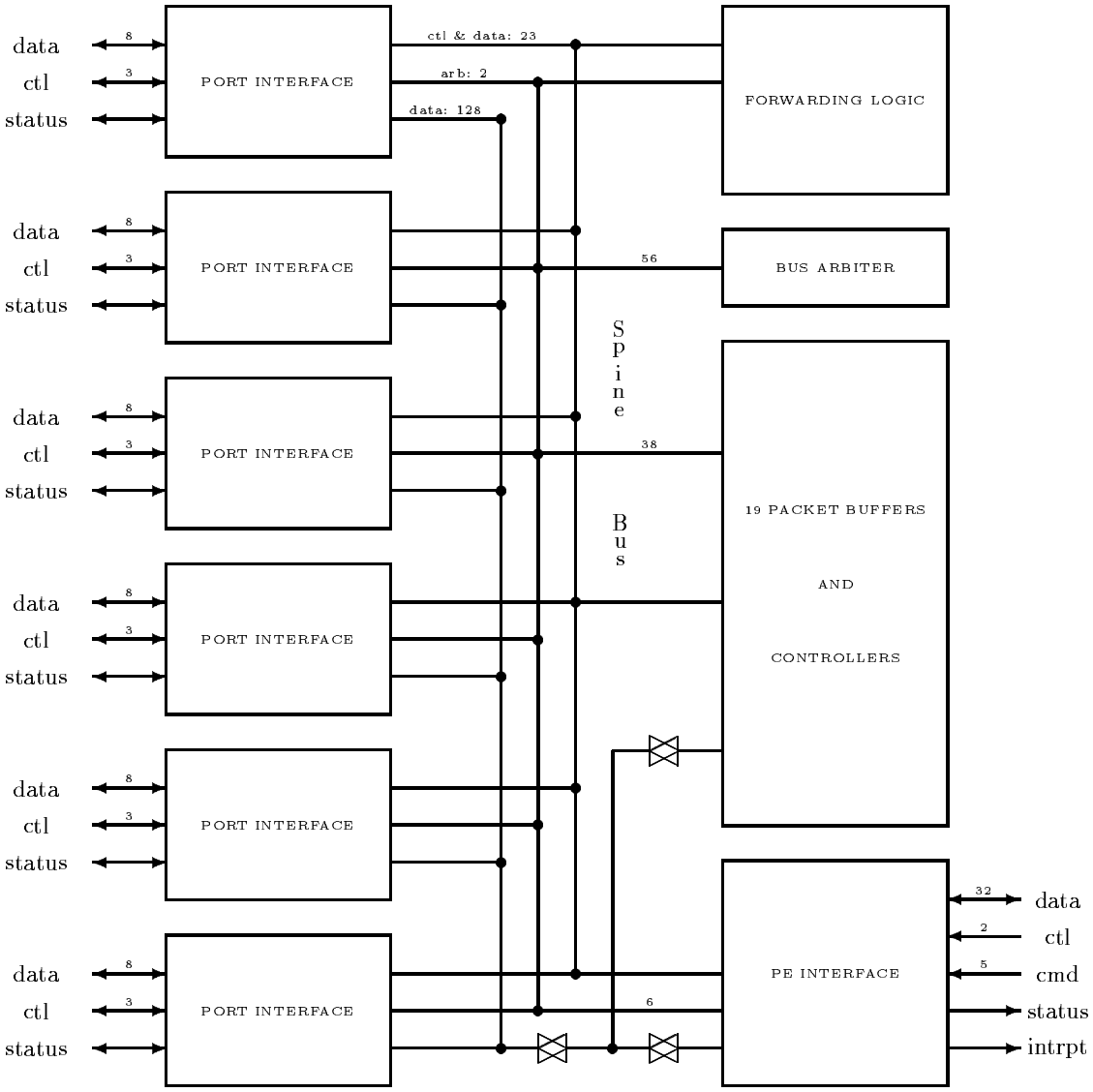

Figure 3: Post Office Block Diagram

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 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 [9], 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. 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.

3.2 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 subtractors, 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 [31], 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 is one library component which is required in our AFSM based design; the mutual exclusion or (ME) element. The Post Office is devoid of all asynchronous library cells except for MEs, and a few C-elements. Arbiters used in the post office consist of one or more ME and a finite state machine.

3.3 Datapath Logic

Data is transferred between elements using a four-cycle self-timed bundled-data protocol [26]. This is a weaker model than that of speed-independence used for control signals inside state machines [18]. There are also some datapath circuits which are controlled in a *clocked* domain, rather than in a self-timed fashion. These datapath cells 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 signalled 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 timing assumption is that the shift register *done* signal will arrive before the completion of the latch/enable cycle, which is easily verifiable.

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 on physical circuit layout, process, and environmental parameters. Hence this is rarely used except for easily verifiable datapath components where completion signal generation is costly in circuit complexity or reduced speed, and where there is a concurrent asynchronous operation and control signal with sufficient delay which can be used to clock the component.

Each RAM block is the size of a packet, 1,152 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. Access 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 interface A and B may both request forwarding a 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 block 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 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 "sequencer" (which is built out of ME elements and a few NAND gates) and a state machine specified by MEAT.

4 MEAT - a Tool for Control Circuit Synthesis

MEAT synthesizes compact, high-performance self-timed circuits from finite state machine specifications. The process is fast enough 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 a burst-mode Mealy state machine. This style of specification is familiar and natural for the typical hardware designer, and provides a powerful way to encapsulate concurrency, communication, and synchronization in an accurate and easily understood form. MEAT compiles the input specification into a set of CMOS functional blocks, which can be thought of as *complex gates*. The result is an implementation which is efficient both in terms of speed and area.

State flow diagrams are used to model the behavior of state machines implemented using MEAT. They provide an intuitive method of defining control functionality, and are similar to commonly used flow charts and state diagrams [11]. 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 the graphical state machine description is then specified textually in the MEAT entry format. 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.

The first automated task performed by MEAT is to generate a primitive flow table [29] from the textual AFSM specification. This is a two-dimensional array structure which captures the behavior represented by the state diagram. 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 AFSM. 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**.

All rows will have a stable entry where an input burst begins. Other entries in the same row may be visited when an input burst occurs. 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. At this point, an unstable state will be entered which will cause a transition to the target row specified and fire the output burst.

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 it is not immediately

evident which values will lead to the simplest circuit, the assignment of specific values to the don't care entries is deferred for as long as possible. The inclusion of don't cares can significantly simplify the state reduction and 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 into one while retaining the specified behavior. The procedure for deciding which rows may be merged is not overly complicated, but for brevity will not be described in this paper. After specifying the flow table, MEAT calculates 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.

The final choice of minimized states must be chosen by the designer. 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). If any of the above constraints are not satisfied, MEAT will inform the user that the covering is invalid.

A new flow table representing the behavior of the minimized AFSM is then generating 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 be assigned to uniquely identify each row of the reduced flow table. 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 [28]. The Tracey algorithm has the advantage that it produces **Single Transition Time (STT)** state assignments. 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*. 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 correct asynchronous operation. Several valid assignments may be produced, and each will be passed to the next stage for evaluation. This will result in unique implementations for each state assignment.

After state codes are assigned, the next synthesis stage computes a canonical sum of products (**SOP**) 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 alternate minimal equations. The large number of don't care entries typically present in the flow table increase the likelihood that more than one minimal expression will be found. Each equation is given a heuristic "weight" that indicates the expected difficulty of implementation and speed of operation using complex CMOS gates. When multiple state assignments have been produced in the previous step, the total weight of each unique SOP equation is used to choose between various instantiations.

The minimized equations produced in the previous step are then used to automatically generate transistor netlists, suitable for simulation, representing complex CMOS gates. A graphical schematic diagram is also produced to help guide the layout process. 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 sum-of-products equation is *folded* into a single gate, the number of logic levels required to generate the output can be reduced. If the function is complex, it can easily be broken up into a tree of complex gates with improved overall performance [27].

5 Design Example

The Post Office design consisted of waves of specification, verification, implementation, and simulation. A set of cooperating AFSMs are specified to control the datapath logic and execute the desired functionality. Logical implementations are then created using MEAT. One or more of the implementations are selected for verification. The verified modules are then physically realized as complex-gate CMOS circuits, which are then checked by simulation. Finally, sets of modules are interconnected and simulated.

Good AFSM design style is dictated by the tool set. Large, monolithic state machines result in large, complex implementations which are difficult to understand, build, and debug. Likewise, their complexity requires exponentially larger compute time for the tool set. Breaking a large specification into a set of small cooperating AFSMs increases the total number of inputs and outputs by adding handshaking signals. However, the overall performance is usually better because the logic is smaller, localized. The number of logic levels and transistors between the output and supply rails are also greatly reduced. Typical state machines in the Post Office have an input to output delay of 2 to 5 inverter delays.

The *sbuf-send-ctl* state machine will be used as a design example. This state machine is from the packet buffer block in Figure 3 and its behavior is specified in Figure 4. This example is taken from the suite of Post Office state machines publicly available for use by other researchers [15, 21]. Sbuf-send-ctl snoops on the spine bus and is activated when it's packet becomes deliverable. The AFSM waits, snooping on the bus, until the forwarding logic has allocated a port for packet delivery. At this point it starts up other state machines which physically forward the packet body with the *send-pkt* signal.

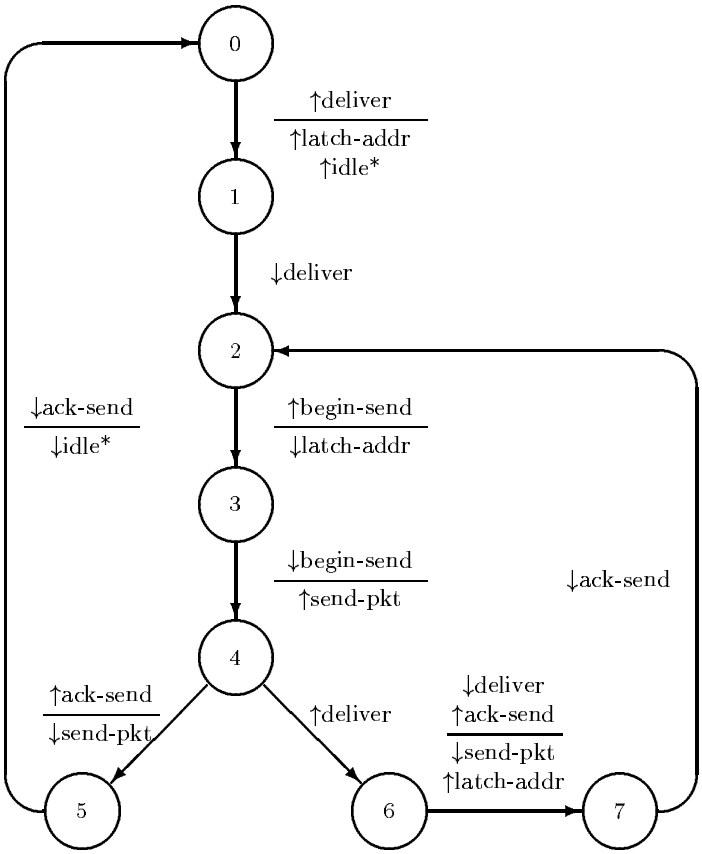

Figure 4: Sbuf-send-ctl State Machine

Sbuf-send-ctl continues to snoop on the bus concurrent with data transfers from this block to determine if the packet was successfully delivered. If the send is aborted and the packet must be delivered again, sbuf-send-ctl waits for the packet to be allocated to another port. Otherwise the function terminates and *ack-send* indicates a successful delivery.

The specification of sbuf-send-ctl from Figure 4 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~)
: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 is then verified 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 $\overline{begin\text{-}send}$ signal is shared by *Y1* and *send-pkt*. The two inverters are merged and the output is forked to both logic blocks. This implementation is then verified. The verifier points out a d-trio hazard [29] which is removed by adding an inverter to change the sequencing of begin-send into the *Y0* logic. The 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 5. The complex gates are then manually implemented using the Electric [23] 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.

To date, we have been unable to specify and verify the behavior of large compositions of elements. Hence all blocks larger than state machines have been simulated but not verified. We have used COSMOS to simulate all logic blocks. Some minor modifications to COSMOS were required in order to simulate the entire chip. Normally new events cannot be injected into COSMOS until the circuit is stable. This model does not work for AFSMs containing both stable and unstable states. We changed the COSMOS interface to allow new events to be injected into an unstable circuit, thereby permitting accurate simulations of the Post Office.

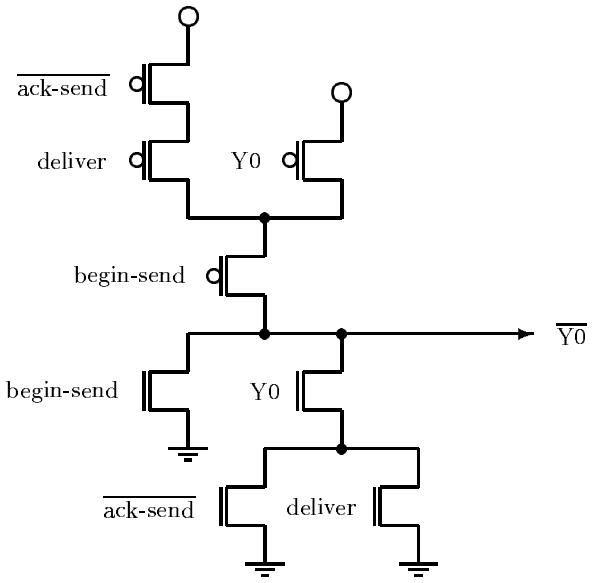

Figure 5: Complex CMOS Gate for sbuf-send-ctl Y0

6 Conclusions

Building a large, fully self-timed circuit has resulted in many insights. The need for synthesis and analysis tools that compare with those available to the synchronous design community is of primary importance. MEAT is a step in the right direction, but many more steps are necessary. The backend only produces schematics at present. Some form of automatic layout is necessary unless we abandon the complex gate approach in order take advantage of standard cell approaches. Automatic layout is a difficult task, but some performance will be lost in the standard cell approach. We are investigating both options. There are a number of performance factors that should be included in the tool set. As a circuit is passed down through the different stages of the tool, some information is lost. The complexity of the algorithms and simplicity of the circuits could be enhanced by preserving some of this information. State graphs lack the formalisms required to analyze compositions of these circuits for safety, liveness, deadlock, and other properties. We are currently investigating a process calculus as a means of specifying and generating MEAT state graphs as well as proving correct operation and construction.

Approximately a fifth of the Post Office control path design was done manually, and the rest was done using MEAT. The automated part of the design took one-fourth the amount of design time and was virtually error free. Those errors were corrected when Steve Nowick pointed out a flaw in our minimization algorithms. Our design style has proven to be a very natural transition for existing hardware designers, primarily since it is based on traditional finite state machine control. Our synthesis techniques have generated compact high-performance circuits that work, and the complexity of the

synthesis algorithms have proven to be viable for large designs.

The present status of the Post Office is that it is operational, although some functional defects do exist, primarily related to inadequate power distribution and pad designs. Packets can 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 $\times$ 8.3 mm. Although this is a research circuit, it is sufficient to demonstrate the validity of the design style.

The inherent modularity of asynchronous designs and their composability into larger modules makes self-timed design of large systems very attractive. The fact that they can then be incrementally improved for performance makes them even more so. A number of open challenges remain. While we have found that testing large asynchronous designs is relatively simple at the board level, it is difficult when the design is a single chip. Since our controllers do not contain latches, it is impossible to use scan paths to improve testability. Electron-beam testers do not help much because it is difficult to image hand-shake signals. We view these problems as opportunities for future research.

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

[8] 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.

[9] 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.

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

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

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

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

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

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

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

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

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

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

[32] 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.