The Design of An Asynchronous Communications Chip

Alan Marshall, Bill Coates, Polly Siegel
External Research Program
HPL-94-38
April, 1994

design,
asynchronous logic,
communications

In this paper we describe a low-power infrared communications receiver chip designed using asynchronous techniques. We focus on aspects that were difficult to implement asynchronously, and contrast our techniques with synchronous ones, where possible. We also detail the methodology used to design the chip, and the asynchronous toolset that was created to support it.

Internal Accession Date Only
To be published in the *IEEE Design and Test Magazine*, June 1994

The Design of An Asynchronous Communications Chip†

Alan Marshall‡ Bill Coates‡ Polly Siegel*
Hewlett-Packard Laboratories, Stanford University

Abstract

In this paper we describe a low-power infra-red communications receiver chip designed using asynchronous techniques. We focus on aspects that were difficult to implement asynchronously, and contrast our techniques with synchronous ones, where possible. We also detail the methodology used to design the chip, and the asynchronous toolset that was created to support it.

1. Introduction

Asynchronous logic design is currently receiving much attention as an alternative to traditional synchronous design [2], [5], [6], [7], [8], [9], [13], [14]. Asynchronous designs do not use a global clock, simplifying global chip routing and eliminating problems due to clock skew. Furthermore, a large portion of a synchronous chip's power budget is dedicated to driving the clock, whereas elimination of the global clock in an asynchronous chip allows it to achieve near-zero standby power when quiescent. For this reason asynchronous design is particularly attractive for battery operated applications.

Unfortunately, asynchronous designs traditionally have been more difficult to design than their synchronous counterparts. The absence of a global clock requires explicit synchronization between communicating blocks. Also, glitches, which are filtered out by the clock in synchronous design, may cause an asynchronous design to malfunction. Additionally, because asynchronous design styles are not yet widely used, supporting design tools and methodologies are still in their infancy, so an asynchronous circuit designer must manually perform many design tasks which are done automatically for synchronous designs.

To demonstrate the suitability of asynchronous techniques for low-power design, we set out to design a chip for a real-world application, and to build up a design methodology and toolset to support our design. We chose an asynchronous communications receiver for an infrared protocol. In parallel with the chip design, we implemented a toolset for asynchronous design, composed of a mixture of commercial and university tools. This toolset supports a design methodology similar to that used by designers of synchronous systems.

The asynchronous nature of the communications receiver impacted many aspects of the design, including the design of the communications protocol and the treatment of noisy inputs and inputs that do not obey asynchronous signalling conventions. The asynchronous design techniques we employed are easily extensible to other power-critical applications. The chip has been fabricated, and was functional on first silicon.

In the remainder of this paper, we describe the overall design of the asynchronous communications receiver, focusing on the portions of the design that were difficult to design and implement asychronously. Where possible, we compare the asynchronous way of doing things with more familiar synchronous design techniques. We also describe the overall design methodology, and the toolset that was created to support it, paying particular attention to synthesis of the control circuitry. Finally, we present power consumption figures measured from the receiver chip, to demonstrate the suitability of these techniques for low-power design.

---

†. Portions of this work were sponsored by the Semiconductor Research Corporation, Contract No. 93-DJ-205.

‡. Hewlett-Packard/Stanford Science Center and Center for Integrated Systems, Stanford University.

*. This author is also a Ph.D. candidate in the Department of Electrical Engineering, Stanford University.

2. Design Description

We chose to design a communications receiver since it is an application that demonstrates the power-saving benefits of asynchronous design. Our goal was to design a receiver which will draw only leakage current while waiting for incoming data, but can start up as soon as a signal arrives, so that it loses no data. A conventional synchronous implementation of such a design would necessarily incur a power penalty while waiting for incoming data, because it can only be sensitive to incoming signals by repeated polling or through the use of active circuitry.† On the other hand, an asynchronous implementation will draw no power in its quiescent state.

The receiver handles an experimental multi-party communication protocol, called ABCS, which operates over an 800Kbit/sec infra-red (IR) link. The protocol can be used for line-of-sight communications between portable computers and peripherals. Figure 1 shows a typical use of this protocol, where peripherals can be shared over an IR link. Other examples include file exchange over an IR link by a group of portable computer users in a conference room, and shared access to the wired LAN via an IR gateway.

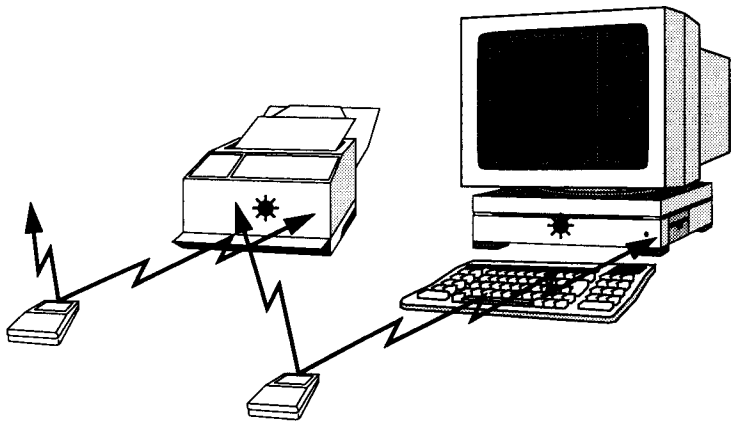


FIGURE 1. Typical ABCS application

The communication receiver chip is a subsystem of the full transceiver circuit that would be required for each station to communicate via the protocol, as shown in Figure 2. Because the environment in which the IR transceivers operate is not well controlled, the receiver must deal with noisy inputs from the IR "link", which could be caused by interference from simultaneous transmissions by several transmitters, reflections from its own transmissions, or stray signals from incompatible IR devices and room lighting.

Because asynchronous circuits are susceptible to glitches on their inputs, they are usually designed to operate in an environment which provides glitch-free inputs, typically conforming to a handshake protocol. We had to develop extensions to traditional asynchronous design to deal with the non-ideal environment in which the transceiver will operate.

Before describing the receiver design, we first describe the ABCS protocol, concentrating on the parts that impact the design. We then describe the overall design of the receiver, and look at areas that were tricky to do asynchronously, such as handling noisy inputs from the environment and handling requests that can be withdrawn before completion.

†. An alternative way to meet our design goal of consuming no standby power is to clock a synchronous implementation of the receiver with a ring oscillator gated as described in Section 2.2.4 and Section 2.2.5. However, this approach would result in large areas of the circuit being clocked more frequently than in the purely asynchronous design, and we expect that the power consumption would be much worse as a result.

2.1 Communication Protocol

2.1.1 Channel access

The ABCS protocol operates over a single half-duplex channel, which is shared using an access contention scheme. This scheme needs no master station to arbitrate access to the channel, and imposes no fixed time schedule on transmissions. Two consequences of this affect the design of an ABCS station, as will be described in Section 2.2.3:

1. An ABCS receiver must wait for incoming data (i.e. it must attempt to receive) for extended periods if it is not to miss data. Since the channel is half-duplex, if a station is given data to transmit while it is waiting for incoming data, then the attempted reception must be aborted.
2. Since incoming data are not synchronized with data transmissions, they must be arbitrated at a single point to determine whether a reception or a transmission should take priority.

A station monitors the IR link for traffic before transmitting a frame, to reduce the probability of frames colliding with each other. Collisions that do occur typically result in corrupted frames, which are ignored by receivers. After a random time-out (to reduce the probability of further collisions), protocols above the ABCS level in each transmitter can retransmit corrupted or lost frames if desired.

2.1.2 Frame structure

The ABCS protocol provides an unacknowledged datagram service where data is transmitted in individually addressed frames (the datagrams) without retries or acknowledgments. Each frame contains Start-of-Frame (SOF) and End-of-Frame (EOF) delimiters, source and destination addresses, payload data and a Cyclic Redundancy Check (CRC).

Frames are divided into bursts called *talkspurts*, which include both calibration information and data. The intervals between talkspurts are used to signal from the receiving station back to the transmitting station, improving the channel utilization and power efficiency of the protocol.

Since there is no global clock, the receiver must be synchronized to an incoming burst. Therefore, each talkspurt begins with a calibration interval (see Figure 3) which provides a timing reference for line decoding. (The calibration process is described in detail in Section 2.2.4.) This is the only feature of the ABCS protocol that was driven by the requirements of an asynchronous implementation.

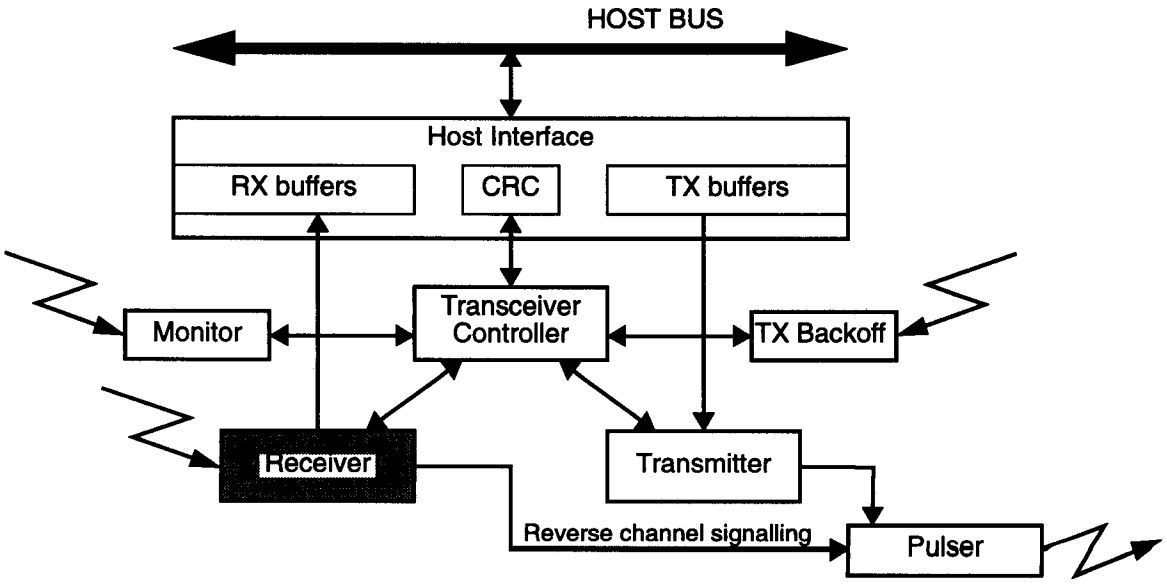


FIGURE 2. Simplified block diagram of an ABCS transceiver

2.1.3 Line code

ABCS uses a Pulse Position Modulation (PPM) line code, both because it is power efficient and because it allows collisions between two stations' transmissions to be detected as line code violations, reinforcing the protection given by the CRC. Each symbol is 2.25μs in duration, and is divided into 9 *ticks* of 0.25μs. The ticks define the nominal IR pulse positions, as shown in Figure 3. There are four data symbols, each of which contains one IR pulse and encodes two bits of data. The address, data and CRC fields of the ABCS frame are all encoded as PPM data symbols. There are also control symbols which encode SOF and EOF in ways that cannot be duplicated by user data.

Since markers are not transmitted at the start of each symbol, the receiver must reconstruct the symbol boundaries from the received pulse stream. The pulse following the reset interval at the start of each talkspurt is used as a timing reference for the first symbol. Each subsequent pulse is used as the reference for measuring the position of the pulse that follows it, allowing the receiver to lock onto the timing of the incoming symbol stream.

2.1.4 Decoding the Incoming Data

In a synchronous design, a crystal-controlled global clock often provides both a timing reference and a mechanism for system synchronization. Since our design does not have a global clock, we need to provide another time reference. To meet our goal of drawing only leakage current while not receiving, the time reference must be inactive when the receiver is waiting for data, and activated only when a signal arrives. Typical crystal oscillators take on the order of 1ms to reach stable oscillation from start-up, which was too long for our needs. Instead, we chose a gated ring oscillator as our time reference, because it can be turned off while the receiver is waiting for data and can be re-started rapidly to receive incoming data. However, the absolute frequency accuracy of a ring oscillator was too poor to decode the PPM line code.

We solved this problem by designing the receiver to dynamically calibrate the ring oscillator against the calibration interval (shown in Figure 3) that is sent over the IR at the start of every talkspurt. At the start of each talkspurt, the ring oscillator is started by the IR pulse that initiates the calibration interval, and is stopped again by the pulse that terminates the calibration interval. The number of ring oscillator cycles that occur during the calibration interval is stored and used to normalize the pulse-to-pulse measurements made during the rest of the talkspurt.

2.2 Receiver Design

The structure of the receiver, shown in Figure 4, reflects the structure of the ABCS protocol, so that the circuitry corresponding to each protocol level is only active and power consuming when an event at that level occurs.

The receiver is composed of:

- Two Interval Timers that measure the IR pulse-to-pulse intervals in terms of the ticks described in Section 2.1.3. They are used alternately, allowing each to "recover" between measurements.
- A Symbol Counter, which sequences the Interval Timers and converts the pulse-to-pulse measurements into a representation of the position of each IR pulse within its symbol period.

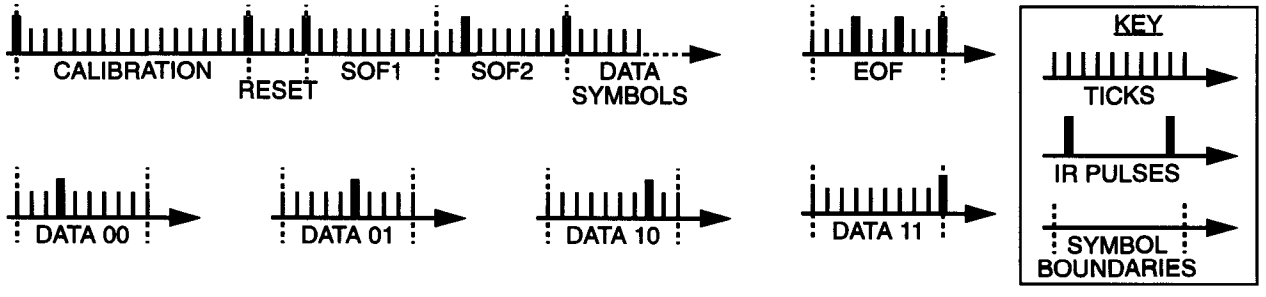


FIGURE 3. The PPM line code, showing a calibration interval and the start of a frame

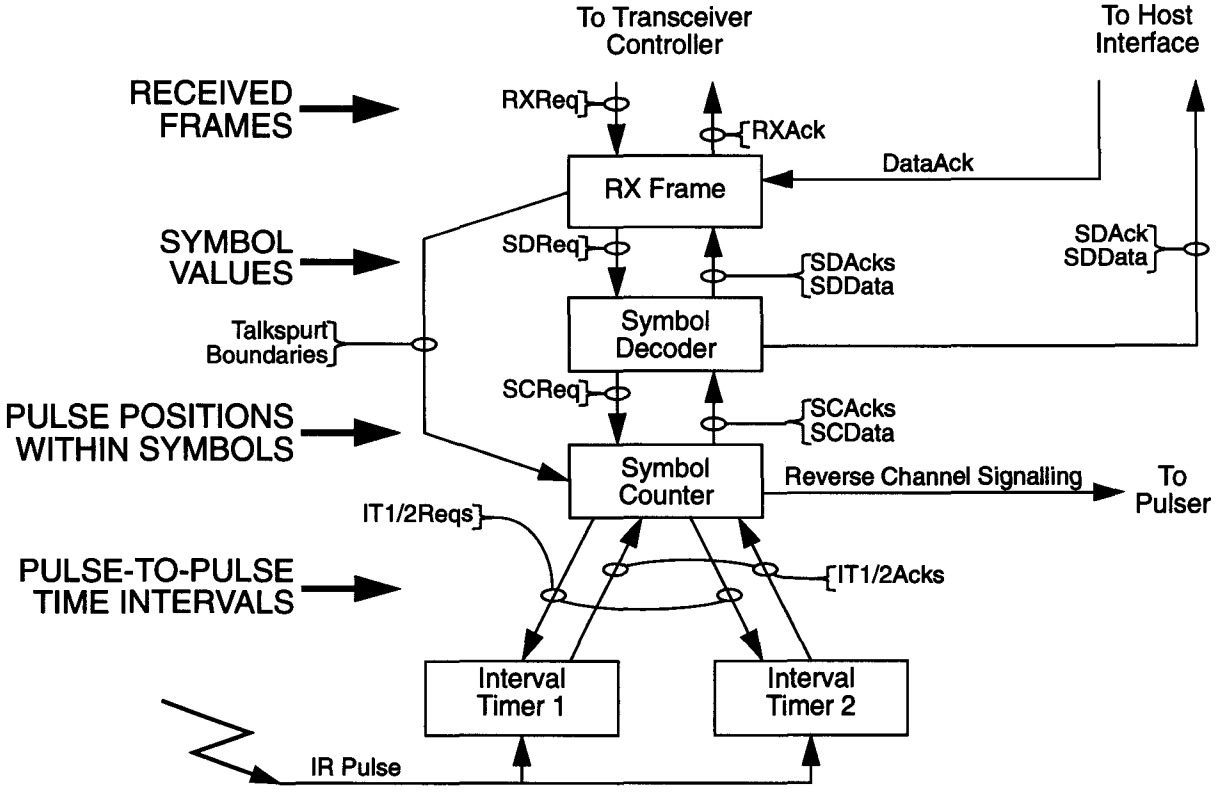


FIGURE 4. Simplified block diagram of an ABCS receiver

- A Symbol Decoder, which decodes the IR pulse positions into logical symbols and checks for line coding errors.
- The RXFrame block, which decodes the incoming frame structure, checks for framing errors, and decides whether the frame is addressed to this station. This block also handles the interface between the receiver and the Transceiver Controller, and identifies talkspurt boundaries to ensure that the reverse channel signalling and Interval Timer calibration are done at the correct times.

The remainder of this subsection will discuss the overall receiver implementation, emphasizing those parts that were impacted by the asynchronous nature of the design.

2.2.1 Receiver Implementation

The receiver chip was manually decomposed into datapath and control blocks, paralleling synchronous chip design methodologies. The control blocks provide localized control of the supporting datapath blocks and handle exception conditions. However, the receiver is designed so that the control blocks do not handle individual data samples directly, reducing power consumption both by reducing the complexity of the control circuitry and by allowing the controllers to be inactive while "routine" data samples are handled by the datapath. The datapath circuitry (mainly registers, counters, comparators and adders) is specifically designed to handle the individual data samples, and does so with less power than a state machine implementation of the same function. Overall, separating the datapath from the control saved power in this design.

2.2.2 Inter-block Communication

Since our design style does not use a global clock, the system must be coordinated through some handshaking protocol between communicating blocks. The handshake protocol forces each participating block to wait until the other is ready to proceed.

The two most common handshaking protocols in use for asynchronous design are *2-phase* and *4-phase* [1]. Both handshaking protocols connect two blocks through two wires, Request and Acknowledge. In a 2-phase handshake a request is made by a transition of the Request wire from its current state, which can be either high or low, whereas a 4-phase handshake obeys a more complicated protocol that, when complete, leaves the Request and Acknowledge wires in the same state in which they started.

Although a 2-phase handshake is seemingly simpler, a state machine using a 2-phase interface needs extra states to cope with each of the two possible initial states of the interface, potentially doubling the number of states compared to those needed for an equivalent state machine using a 4-phase interface (and re-doubling the number of states for each additional independent 2-phase interface connected to the state machine). We felt that the larger size of the 2-phase state machines outweighed the power savings from elimination of the reset transitions over the interface wires, so we chose to use a 4-phase protocol.

The basic 4-phase handshake works as follows:

1. The controlling block drives Request high, which requests some action from the controlled block. The controlling block then waits for a response on Acknowledge before proceeding.
2. The controlled block drives Acknowledge high when it has completed the action, to allow the controlling block to proceed.
3. The controlling block drives Request low, to acknowledge receipt of the result of the action and allow the controlled block to proceed. The controlling block again waits for a response on Acknowledge.
4. The controlled block drives Acknowledge low when it is prepared to accept another request. This returns the interface to its initial state, and allows the controlling block to drive Request high again.

The basic 4-phase handshake described above can be expanded to include multiple Request and/or Acknowledge wires when one of several actions or results needs to be communicated. Also, one or more data wires can be controlled by a single 4-phase handshake interface to control the data transfer. This approach is described in [1], and has since become known as *bundled data.* The major inter-block connections shown in Figure 4 use all of these techniques.

Design Issues for Communicating AFSMs

Handshaking alone cannot always ensure that control blocks can communicate without violating the constraints imposed by the state machine design style we use. Problems can arise when the control block is connected to an environmental (off-chip) signal and is exposed to unexpected inputs (like noise), and when the control block is controlling multiple blocks. Furthermore, timing requirements on feedback paths must be met for the control blocks to work properly. To understand these problems, we must first preview some aspects of the asynchronous finite state machine (AFSM) design style, which will be explained more fully in Section 3.4.3.

Each control block is implemented as one or more *burst-mode* AFSMs [2] which must conform to specific environmental constraints. These state machines can accept specified bursts consisting of multiple input transitions before changing state and possibly outputs. The input transitions within a burst can arrive in any order, but they must be glitch-free, and only specified input transitions are allowed to occur. Furthermore, after an input burst has been completed, the state machine must be allowed sufficient time to change its outputs and settle its internal state before further input transitions can be applied. Because the handshake protocol cannot always ensure that these constraints will be met, we must sometimes add preprocessing circuitry at the boundary of a control block.

There are three cases in which we need to preprocess state-machine inputs to meet burst-mode constraints:

1. When there is a fast feedback path from a state machine output to its inputs, resulting in a change of the state machine's inputs before the state machine's internal state has settled. This can occur when a slow state machine interacts with a much faster environment.
2. When a state machine must process a sequence of external signals, where only a few of the possible sequences occur in normal operation. This can occur when a signal comes from the environment and numerous erroneous sequences are possible.

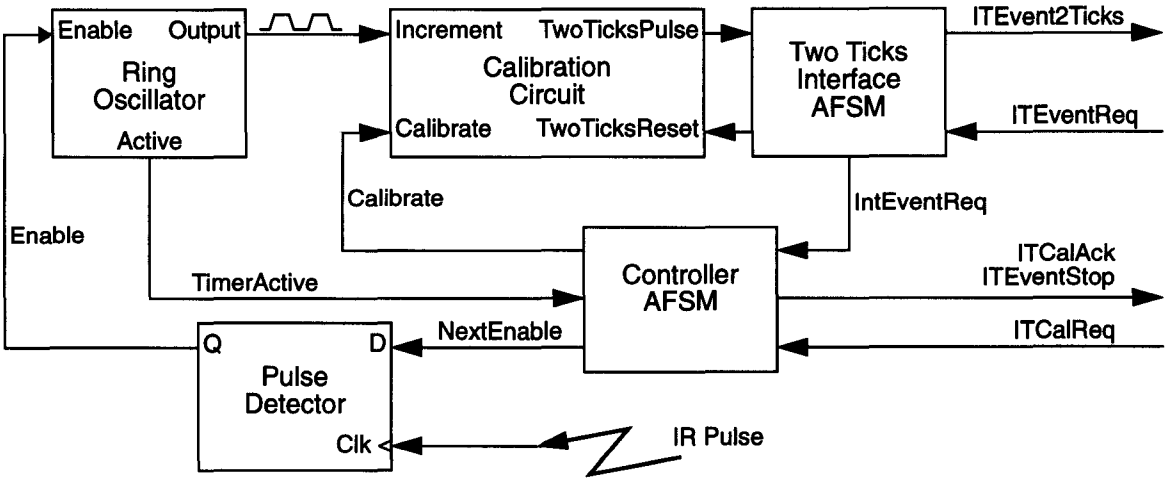


FIGURE 5. Simplified block diagram of an Interval Timer

3. When the relative timing of some input signals determines the next state. This can occur when an AFSM is controlling multiple blocks and wants to act on the first acknowledgment from any of the controlled blocks, or when a signal comes from the environment and is thus not synchronized with any internal signal.

Case 1 can be corrected by adding a delay in the feedback path, although this might reduce the operating speed of the system. We added delays built out of inverter chains at a few places in this design to handle problems of this type.

Case 2 is a problem because a burst-mode AFSM is only guaranteed to work correctly if an input burst falls within the set of allowable bursts specified at design time. Thus, it can only handle erroneous inputs if all possible input sequences are included in the specification, which can lead to a complex state machine. In our design, the Symbol Decoder had this problem. Rather than adding all possible sequences of input values to the specification of the Symbol Decoder controller, we explicitly preprocess the inputs of the AFSM so that only valid inputs are passed to the AFSM, and invalid inputs are flagged as errors.

Case 3 can result in the AFSM entering a metastable state, or becoming locked in some unspecified state until it is reset. To handle these situations, the receiver design uses several mutual-exclusion elements (MEs), derived from the design described in [1]. Arbiters [1] provide a more general mechanism to handle such situations, but MEs worked fine for the cases we encountered.

2.2.3 Withdrawable Requests

The 4-phase handshaking protocol demands that a request is maintained until acknowledged. However, referring to Figure 2, if the Transceiver Controller requests the Receiver to become sensitive to incoming data, and the host subsequently requires the Transceiver Controller to initiate a transmission, then the Transceiver Controller will withdraw its original request unless it has already been notified by the Receiver that reception has started.

Withdrawing a request before it is acknowledged can cause a violation of the timing requirements for burst-mode state machines, and so this aspect of the design is hard to handle in our design style. In addition, because the incoming data is not synchronized to the request to transmit data, these two events must be arbitrated at a single point to make an unambiguous decision which of the transmission or reception should proceed. We placed this decision point in the Transceiver Controller to simplify the overall structure of the ABCS station. This means that the Transceiver Controller can initiate a transmission, withdrawing its request for the Receiver to receive incoming data, even if the Receiver is just starting to receive a frame.

We could not see how to use a straightforward combination of burst-mode AFSMs and arbitration circuits to safely handle the withdrawal of a reception request. Instead, we transformed the withdrawal of the reception request into a

hard-reset request, exploiting the existing power-up hard-reset circuitry. This forces the affected AFSMs into their initial states regardless of the arrival times of any other incoming signals. A reset acknowledgment signal is used to ensure that the reset request cannot be removed before the reset completes. Although aesthetically unappealing, this approach provides an economic implementation of a withdrawable request mechanism.

2.2.4 Recovering Pulse-to-pulse Timing from the IR Input

Figure 5 shows a block diagram of an interval timer, which is responsible for recovering pulse-to-pulse timing from the IR input. Each time the Interval Timer is activated, it measures the interval between a pair of successive IR pulses by starting its Ring Oscillator on the first pulse, stopping it on the second, and counting the number of Ring Oscillator cycles during the interval.

The Controller drives the NextEnable signal to determine whether or not the Ring Oscillator should run after the next IR pulse. When an IR pulse arrives, the Pulse Detector transfers the value of NextEnable to Enable, starting or stopping the Ring Oscillator at the correct time. The Pulse Detector does this while isolating any noise on the IR pulse input from the rest of the system, as will be described in the next section.

At the start of each talkspurt, the Interval Timer is requested to treat the next pair of IR pulses as delimiters for the calibration interval. In this case, the Calibration Circuit counts the Ring Oscillator cycles generated during the interval, and stores the result for the duration of the talkspurt.

Throughout the rest of the talkspurt the Interval Timer is requested to treat pairs of IR pulses as delimiting an interval which represents a data symbol. In this case the cycle count from the Ring Oscillator is divided by the count derived from the calibration interval. This gives a pulse-to-pulse measurement normalized to the length of the calibration interval, expressed in units of the ticks which were described in Section 2.1.3. In this way adequate measurement accuracy can be achieved with a Ring Oscillator which has a frequency stable to about 1% over the 2.8ms duration of a talkspurt, almost independent of its absolute frequency accuracy. We think that the 1% stability requirement can be met if a suitable power supply is used.

2.2.5 Handling Noisy Inputs

Since the IR "link" is susceptible to noise from the environment, the receiver must be able to filter out any noise in an incoming IR signal. Unfortunately, most asynchronous systems are potentially responsive to transitions on any of their inputs at any time, since they have no clock with which to sample their inputs. If the IR signal was fed directly to conventional gate-level circuitry, then invalid logic levels could be propagated throughout the system causing the circuit to malfunction. In a conventional synchronous implementation, we would simply sample the IR signal and allow a long enough resolution time for the probability of metastability to be acceptably low. However, since in our asynchronous implementation the IR signal is used to *trigger* operation of the whole receiver, this solution is not available.

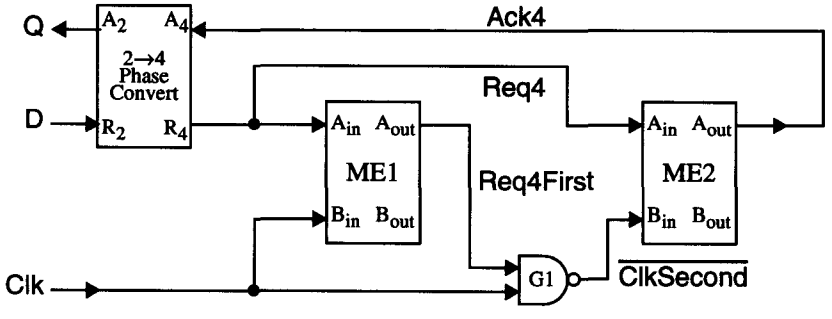
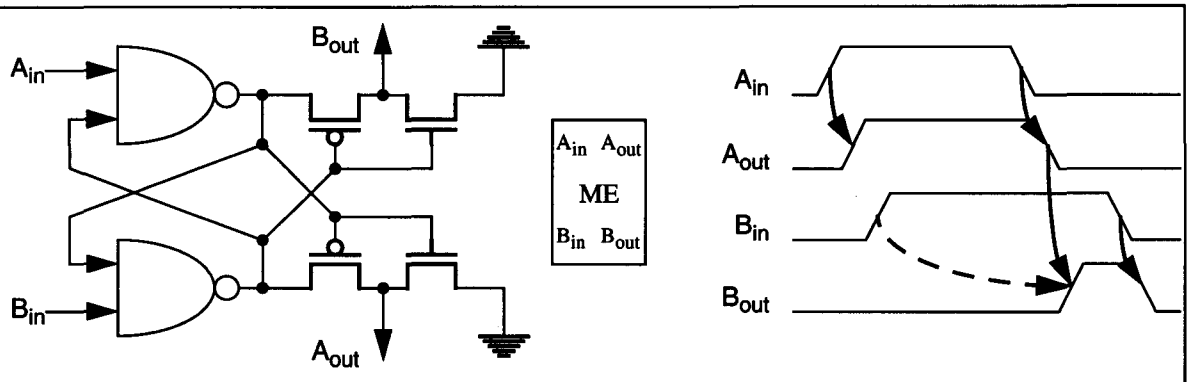


FIGURE 6. Schematic of a pulse detector



A mutual-exclusion element (ME) is a basic building block of asynchronous logic, which is designed to resolve race conditions on its inputs. The ME accepts two inputs with no constraints on timing, and passes each signal to its corresponding output unless both inputs are high, in which case only the first arriving signal is passed to its output. Figure A shows a schematic of the ME we used in our design, and its corresponding timing diagram. This ME was derived from one presented in [1].

If two inputs arrive simultaneously on the ME's inputs, then the cross-coupled NAND gates can become metastable. However, the analog behavior of the output transistors ensures that the circuit outputs switch cleanly between valid logic levels and that only one output can be high at any time. Although the output is resolved to a valid logic level, metastability may cause an arbitrarily long delay between an input transition and the resultant output transition.

INSET A. How a mutual exclusion element works.

To solve this problem, we developed a circuit which we call a *pulse detector*, shown schematically in Figure 6. This circuit has an external interface similar to a D-type flip-flop. In the terminology of asynchronous systems, a transition on the D input is interpreted as a request to output a transition on Q when the next high-going clock edge is detected.

The pulse detector differs from both a D-type flip-flop and a Qflop [3] in that an invalid logic level on the Clk input does not result in an invalid logic level on the Q output. Instead, any transition on Q is delayed until the input change is large enough to cause a full output swing, and until any metastability is resolved. The pulse detector also shields its output from internal metastability caused by simultaneous transitions on the Clk and D inputs, as does a Qflop. As a result the pulse detector will clean up the receiver's noisy IR input signal, which is applied to the Clk input of the pulse detector.

The pulse detector uses mutual-exclusion elements (MEs) to resolve any races and to filter out noise on the signal. ME1 masks any metastability caused by simultaneous transitions on the D and Clk inputs. ME2 acts as a "threshold detector," preventing any invalid levels, runt pulses or noise on the Clk input from propagating to the rest of the circuit. G1 blocks input pulses until the D input has been asserted and the Clk input has gone low, ensuring that only complete pulses are detected. (The operation of an ME is explained in detail in Inset A.)

The operation of the circuit depends on the fact that, even if the two MEs exhibit internal metastability, only valid logic levels will appear on their A_{out} outputs. If Clk is driven to valid logic levels, the MEs produce valid A_{out} output levels as described in Inset A. If Clk is driven to invalid logic levels, these levels are fed to the inputs of the MEs, putting the MEs outside their original design operating conditions. In this case the MEs' A_{out} output levels are still logically valid, but only because of the detailed analog behavior of our ME implementation, and because we know that the Clk input will be driven by a band-limited signal in which transitions will be separated by much more than a gate delay.

Referring again to the schematic, the two-phase to four-phase converter implements a complete four-phase handshake on its R_4 and A_4 terminals for each transition on its R_2 input (the D input of the pulse detector). Only when the four-phase sequence has been completed does the converter output a transition on its A_2 output to acknowledge the original request. The remaining circuitry delays the completion of this handshake until a pulse arrives, as explained below.

In the simple case when the Clk input (connected to the IR link) is driven to valid logic levels, the circuit operates as follows. Starting from a state where Clk is low and Q = D, changing the level of D will cause Req4 to go high. In the most common chain of events, if Req4 goes high before Clk, then ME1 drives Req4First high. The circuit is now stable until Clk goes high, when G1 will drive $\overline{\text{ClkSecond}}$ low. This allows ME2 to pass the high level on Req4 through to Ack4, making the converter lower Req4. Lowering Req4 causes Ack4 to go low, which causes the converter to change Q to the new value of D, leaving the circuit stable once more. Another possible chain of events occurs if D is asserted when Clk is already high, when ME1 will block the high level on Req4 from appearing on Req4First, G1 will prevent the high level on Clk from lowering $\overline{\text{ClkSecond}}$, the pulse on Clk will have no effect on either Ack4 or Q, and the pulse will not be detected.

3. Design Methodology and Tools

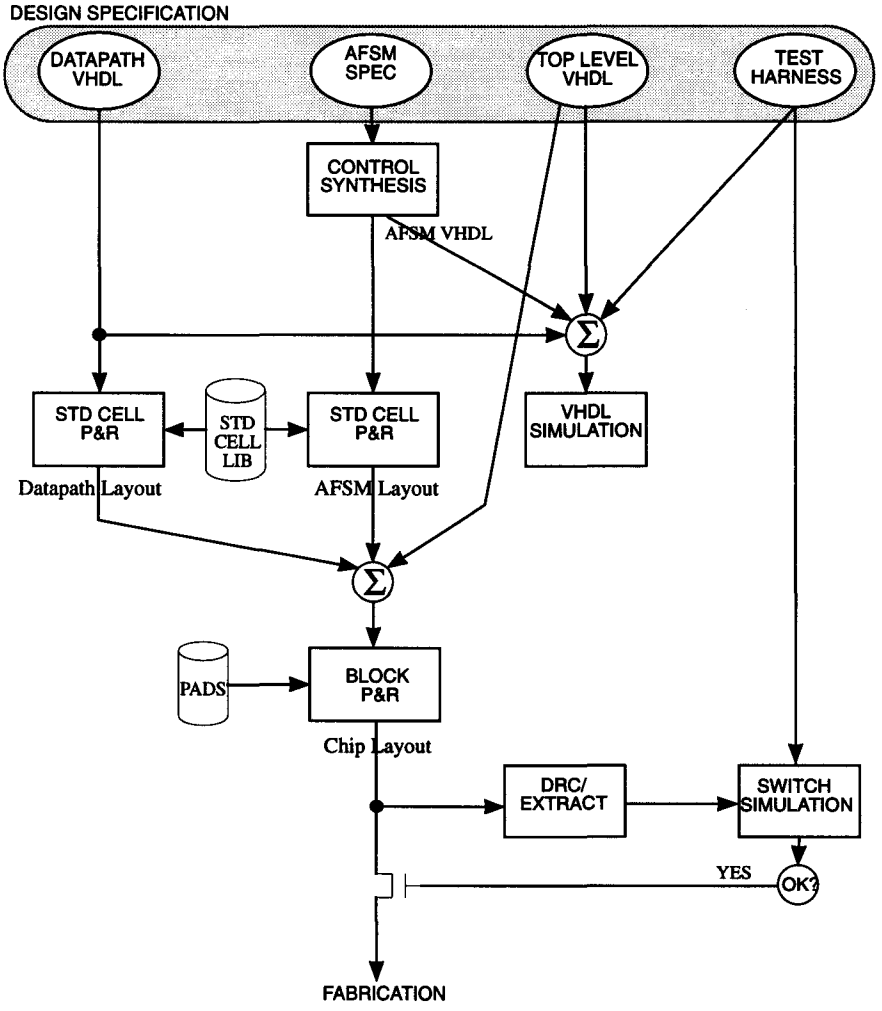


FIGURE 7. Overall design tool flow

In this section we describe the design methodology and toolset that we developed and used to implement the receiver IC. First, we give an outline of the overall flow of the tools. Next, we briefly discuss some modeling issues we encountered while implementing the design. We then describe how the datapath and control blocks are generated,

going into some detail on the control generation since the datapath generation is straightforward. Finally, we briefly describe the physical design tools we use and how they fit into the overall flow.

3.1 Overall Tool Flow

Our design methodology starts with a behavioral description of the design in VHDL. The design is partitioned manually into control and datapath components, much as in synchronous design. Structural VHDL is used to connect the high-level components together. The high-level tool flow is illustrated in Figure 7.

Once the partitioning into control and datapath blocks has been done, the designer creates AFSM descriptions describing the behavior of the control portions of the design. These descriptions are written in a simple language, and from them both standard-cell layouts and behavioral VHDL models can be automatically synthesized by the tools.

The datapath components are described initially as behavioral VHDL models, and then are manually refined into gate-level implementations composed of standard-cell components. The gate-level VHDL for each datapath component is then converted automatically to a standard-cell layout by the tools.

The design is simulated in VHDL at each step of refinement, to check both the control synthesis and the incremental refinement of the datapath. Once all components of the design have been synthesized and simulated, a block-level router is used to connect the standard-cell blocks together and to route external signals to the pads. A transistor-level netlist of the design is then extracted from the layout data and simulated, the results being compared against the results from the high-level VHDL simulation.

3.2 Behavioral Modelling Issues, Simulation and Timing Analysis

As mentioned earlier, the burst-mode asynchronous finite state machines which we have used in this design [2] are specified to accept only certain inputs when in a given state. If a burst-mode state machine is presented with an input event which is not specified for its current state, the behavior is undefined. To aid the designer in detecting such errors, the behavioral model must detect the occurrence of any unspecified input event, and either report the event or set the state machine's internal state and outputs to an unknown state. We found that behavioral VHDL models of burst-mode state machines which met these requirements could be very lengthy—over 1600 lines of VHDL for one of our state machines. Because of this, we wrote a tool to automatically generate these VHDL models from the AFSM descriptions.

Because timing information is not available until the design has been implemented, "guesstimated" timing is used in the behavioral VHDL descriptions. Once the design is fully implemented, accurate timing information is extracted from the layout and used in a switch- or circuit-level simulation of the complete design. The lack of accurate timing information in the early design stages meant that some timing problems were revealed by switch-level simulation that weren't evident in the behavioral simulations. Because our design path is highly automated, these problems were easily corrected by changing the VHDL and state machine descriptions, and then re-synthesizing the design.

Since our toolset lacks a timing analysis tool and our state machine synthesis tools do not give us any control over the timing specifications of the state machines, we had to check whether the burst-mode timing constraints were satisfied through manual analysis of data from the switch-level simulation. We found a few violations where relatively slow state machines interacted with faster datapath components or state machines, and we added delays (inverter chains) manually in a few places to ensure safe timing margins.

3.3 Datapath Generation

For our design, we generated the datapath components through successive simulation and manual refinement of VHDL from the behavioral down to the gate-level. We could have used existing commercial datapath generation tools to generate a gate-level datapath description from the behavior, but the cost of library customization and software would have been more than the cost of the labor for manual refinement of this one design. However, automatic datapath generation software can be incorporated into the toolset easily, and should be for future designs.

Once a datapath component has been described down to the gate level, its structural VHDL is converted automatically to the format used by the Mentor GDT AutoCells standard-cell place-and-route tool, and the datapath compo-

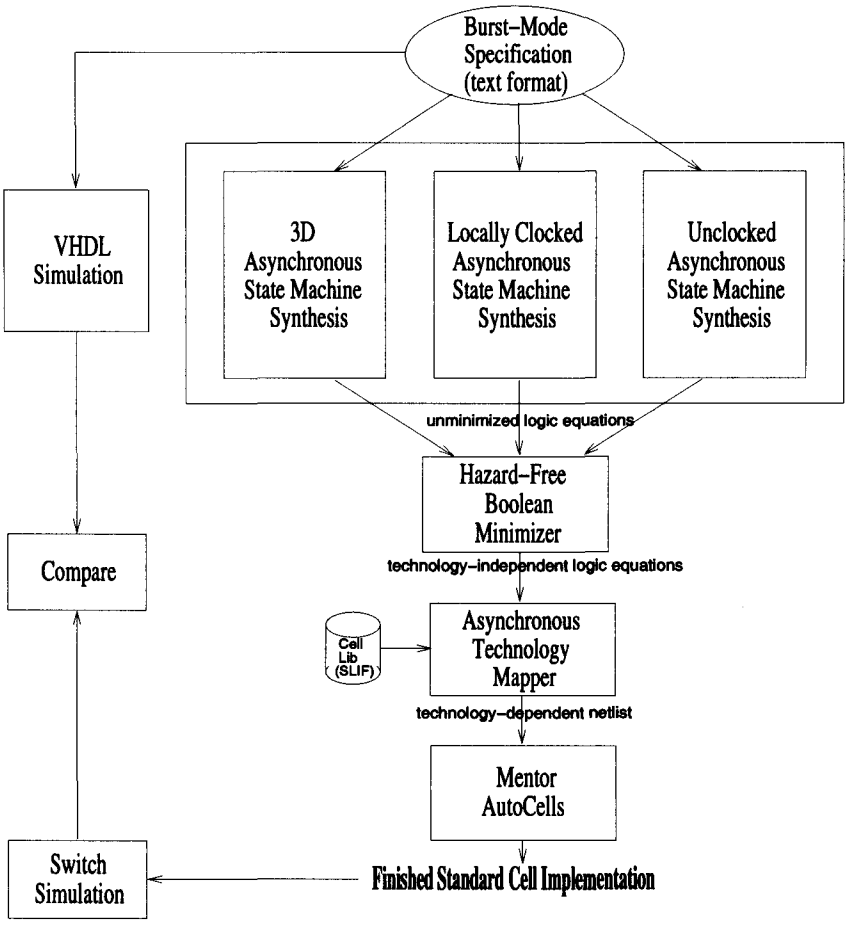


FIGURE 8. Control Synthesis Tool Flow

nent is implemented as a standard-cell block. This block is then connected into the circuit by an HP proprietary block place-and-route tool (although any BPR tool can be used).

3.4 Control Synthesis

The control synthesis process is by far the most interesting part of the tool flow, and will be described in some detail. The control synthesis path uses a combination of university tools, custom scripts, and commercial tools to take a high-level asynchronous finite-state machine specification and generate a standard-cell block implementing the state machine. The resulting implementation is then connected to the rest of the circuit with the block place-and-route tool.

3.4.1 Overall Flow of Control Synthesis

Figure 8 outlines the overall flow of the control synthesis process. The control portion of the design is entered as a simple textual description of the state machine. This description is then translated automatically into VHDL for simulation. The description can also be fed into one of three state machine synthesis programs, each of which synthesizes an AFSM in a specific implementation style. The output of the state machine synthesis step is a set of hazard-free equations which implement the specified state machine behavior. The equations are then minimized using a hazard-free asynchronous logic minimizer and the resulting equations can be mapped automatically to a standard-cell or gate-array implementation using an asynchronous technology mapper which is guaranteed not to introduce new hazards. Each state machine is implemented as a separate standard-cell block, to maintain close control over the wire lengths and hence the delays of the internal signals.

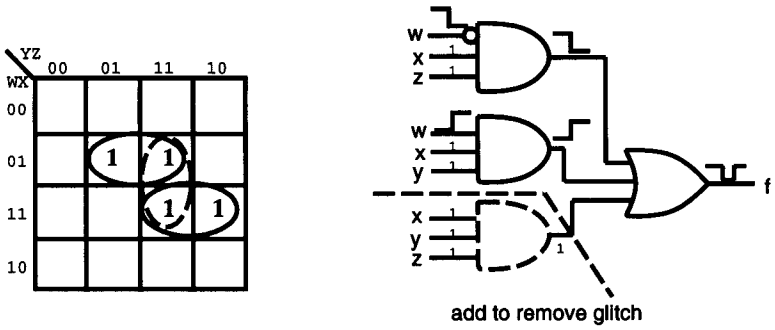


FIGURE 9. Example illustrating hazards

3.4.2 Hazards

Before we discuss the details of the state machine specification and synthesis, we explain the significance of *hazards*, which are a key to the difficulties faced in asynchronous design.

A hazard may be defined informally as the potential for an unwanted signal transition, or *glitch*, to occur in response to a given transition on the inputs. Hazards, which usually can be ignored in a synchronous system, may cause improper operation in an asynchronous system. Since signal transitions in an asynchronous system are meaningful at any time, *no* hazards can be tolerated in the control portion of the final circuit. This is arguably the single most important factor which contributes to the difficulty of asynchronous design.

Figure 9 illustrates how a type of hazard known as a *logic hazard* can be introduced into an implementation. Figure 9(a) shows a Karnaugh map which defines a simple function of four variables, and Figure 9(b) is a two-level AND/OR implementation of the function. Each circled group of minterms in the Karnaugh map corresponds to an AND gate in the implementation. Consider the behavior of the circuit in the case where input w changes from 0→1 while all other inputs remain high. Unless the dashed gate is included in the implementation, variations in logic gate speed could result in an unwanted 1→0→1 glitch appearing on the output f.

Hazards can be a problem at each step of the synthesis. At the state machine synthesis stage, the synthesis methods must ensure that the behavior of the state machine does not depend on the outcome of any races between internal signals, as the real outcome of such races might not be as the designer predicted. Modifications to logic minimization and technology mapping algorithms are also required to ensure that the algorithms do not introduce logic hazards into the final implementation.

3.4.3 State Machine Specification

To produce hazard-free AFSMs, constraints must be placed on how inputs are allowed to change. Traditional ASFMs typically obey a fundamental-mode *single-input change* (SIC) constraint where only a single input is allowed to change at a time, with a sufficient time left between transitions to allow internal logic to stabilize [4]. This requirement is too restrictive for real-world designs. The specification style we use is a relaxed form of *multiple-input change* (MIC) operation, called *burst-mode*, in which the outputs and state variables change in response to the last transition of a specified multiple-input burst [4]. Note that the set of transitions in an input burst can occur in any order, and at any time. This specification style has been shown to be practical for large designs and is the subject of active research ([2], [5], [6], [7], [8], [9]).

A burst-mode AFSM is specified by a state diagram consisting of a finite number of states, joined by directed arcs representing state transitions. Each arc is labeled with a set of input signal transitions, called an *input burst*, and a (possibly empty) accompanying set of output transitions, called an *output burst*. In a given state, once all input transitions in an input burst have occurred, the state machine moves to the new state and generates the corresponding output burst. Figure 10 shows an example of a burst-mode state machine specification.

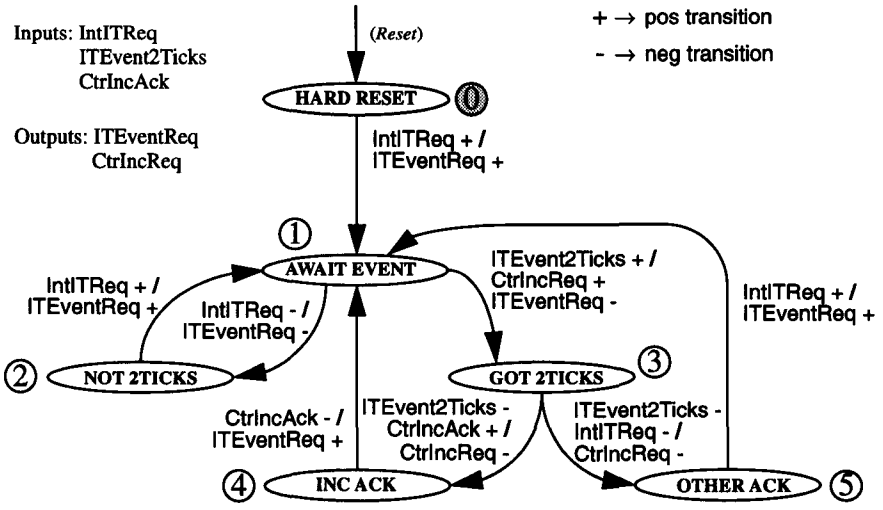


FIGURE 10. Burst-Mode AFSM Specification

There are several restrictions on the AFSM specification which must be taken into account during the design process:

1. To ensure deterministic behavior, any two arcs leaving a single state must be mutually exclusive.
2. Any input or output transitions not explicitly specified are forbidden.
3. A given state must always be entered with the same combination of input and output values.

Some constraints, such as restriction 3, are checked automatically, while others impose requirements on the external environment. In some cases, restriction 1 requires that special asynchronous blocks such as mutual exclusion elements or arbiters be used at the inputs to the AFSM, as we described in Section 2.2.2.

Restriction 2 may be the most difficult one for a synchronous designer to become accustomed to. Since all transitions on input signals can potentially affect the circuit, care must be taken to ensure that no "unexpected" transitions occur. This requires a very careful analysis of the environment in which the AFSM is to operate, and is one of the reasons why asynchronous signaling events must obey handshake protocols.

3.4.4 State Machine Implementation and Synthesis

In its simplest form, an asynchronous FSM is implemented as shown in Figure 11. A change in the value of the inputs to the combinational logic block may cause the outputs to change. In addition, some subset of the outputs is fed back to perform a latching function. To operate correctly, the combinational logic must be allowed time to settle before the next input change is applied (the fundamental-mode requirement). Ensuring correct fundamental-mode operation requires making some easily verifiable local timing assumptions. [4]

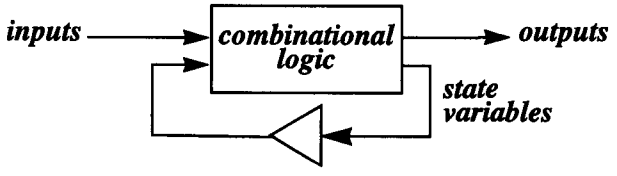


FIGURE 11. Basic asynchronous finite state machine (AFSM) architecture

There are several burst-mode design styles available which map into this architecture ([10], [2], [5]). In some cases, the state variables are fed back directly to the combinational logic as shown. In other cases, a set of latches isolates the feedback path. Since there is no global clock, the AFSM is sensitive at all times to the level values on the inputs. This implies that outputs must make clean, monotonic transitions, and must remain stable until the next desired transition; i.e., they must be *hazard-free*.

Three synthesis methods, all developed at Stanford and based on burst-mode synthesis tools originally developed at HP Labs [5], are available to synthesize the control logic from a high-level textual description. At present, however, only the 3D synthesis method [2] is fully integrated into our toolset. All methods accept a common input format, which is a simple textual description of the desired AFSM behavior, and produce a hazard-free implementation of a burst-mode specification, differing in the details of the circuitry produced. The input file corresponding to the state diagram in Figure 10 is shown in Figure 12.

The operation of a 3D state machine is similar to the operation of a Huffman machine [4]. In response to an input burst, the outputs and state variables change as specified. The fundamental-mode assumption must hold, meaning that the next input burst must not occur until the state variables and outputs have settled at their final values. The feedback path must meet some one-sided timing constraints to ensure proper state machine operation.

3.4.5 Hazard-Free Logic Minimization

The output of the state machine synthesis step is a set of hazard-free logic equations which implement the state machine behavior. These equations are not necessarily minimal, however, so they are processed by an asynchronous logic minimizer before being passed to the technology mapper. Each input burst allowed by the specification must be examined to ensure that no hazards will exist in the implementation.

```
;start specification
;input is free-format, comments start with a semicolon

;signal names section:
;type   signame    initial value

input   IntITReq        0
input   ITEvent2Ticks   0
input   CtrIncAck       0
output  ITEventReq      0
output  CtrIncReq       0

;behavioral description section:
;from   to      input burst          output burst

0       1       IntITReq+           | ITEventReq+
1       2       IntITReq-           | ITEventReq-
2       1       IntITReq+           | ITEventReq+
1       3       ITEvent2Ticks+      | CtrIncReq+
                                      ITEventReq-
3       4       ITEvent2Ticks-
                CtrIncAck+          | CtrIncReq-
3       5       ITEvent2Ticks-
                IntITReq-           | CtrIncReq-
4       1       CtrIncAck-          | ITEventReq+
5       1       IntITReq+           | ITEventReq+

;end specification
```

FIGURE 12. Textual State Machine Specification

The asynchronous logic minimizer [6] uses a constrained version of the familiar Quine-McCluskey algorithm [11], and takes hazards into account during the minimization process. The logic produced by the minimizer is typically larger than that produced by an equivalent synchronous minimizer, but in the cases we have examined the overhead (with respect to the number of product terms) has been no more than 6%.

3.4.6 Asynchronous Technology Mapping

The technology mapping step takes as input the hazard-free technology-independent equations generated by the logic minimization step, along with a cell library description. The output of this step is a netlist of elements from the library which represents an implementation of the state machine in the given technology. In contrast to synchronous approaches, the outputs and state variables must be hazard-free for all transitions of interest.

Ceres [12], a synchronous algorithmic technology mapper developed at Stanford University, was modified for use with asynchronous circuits [9]. The technology mapper first examines the cell library elements to characterize their hazard behavior. A matching step follows which replaces portions of the input network with library elements that have equivalent functionality. During this step, any subnetwork which matches a hazardous library element is also analyzed for hazards, and only library elements which contain a subset of those hazards can be accepted. Thus the technology-mapping step is guaranteed not to introduce any new hazards into the design. Any existing hazards in the initial network will correspond to input transitions which are disallowed by the specification.

Netlists generated by the technology mapper are automatically translated into a format used by the Mentor GDT suite of tools. Place-and-route of the basic state machine is then done automatically using the GDT AutoCells place and route tool. The resulting state machine is thus implemented as a single standard-cell block.

3.5 Physical Layout and Verification

Once all control and datapath blocks have been generated, a block place-and-route tool is used to wire the chip together from the top-level VHDL netlist description. After routing, the design is extracted with Mentor's CheckMate and simulated with Lsim, using a test harness equivalent to the one used for VHDL simulation. The VHDL and Lsim simulation results are compared to check for any errors introduced by the synthesis process, and for those timing errors which can only be detected after synthesis.

4. Results

The receiver IC was implemented using the MOSIS 1.2μ CMOS process, and contains approximately 14,000 transistors, of which 1,200 are used to improve the observability of internal nodes for testing. We have measured the current consumption of the receiver core (i.e. without pads) to be less than 1mA @ 5V when the receiver is actually receiving data. The receiver draws less than 1μA @ 5V while waiting for data, when it is electrically static. The final VHDL description of the receiver is about 13,000 lines long, of which 6,000 lines are structural VHDL representing the netlist of the circuit, 4,000 lines are (generated) behavioral VHDL representing the asynchronous state machines, and 3,000 lines are behavioral VHDL representing the standard-cell gates in our library.

Our measurements confirm that asynchronous design techniques allow designs to operate in a data-driven fashion, and thus allow individual modules to be electrically active only when they have processing to do. In our design this means that most of the circuitry operates well below the maximum frequency of operation. However, this less-active circuitry is not constrained to operate with large latencies and low potential throughput as would be the case if it was

simply clocked more slowly. The effects of this are illustrated in Table 1, which gives the average current consump-

BLOCK		# FETs	I_{supply} (μA) @ 5V	I_{supply} /FET (nA)
Interval Timer	- Control	242	6	25
	- Oscillator	534	151	283
	- Remainder	2,296	180	78
Symbol Counter	- Control	794	45	57
	- Datapath	672	67	100
Symbol Decoder	- Control	816	52	64
	- Datapath	306	17	56
RX Frame	- Control	218	1	5
	- Datapath	2,452	14	6
TOTAL (with 2 Interval Timers)		11,402	870	76

TABLE 1. Measured current consumption of major receiver blocks during data reception

tion during data reception for each of the receiver's major blocks. The table also shows the current consumption on a current/FET basis as a measure of the level of circuit activity. All of the measured power figures were within 20% of figures simulated with the Mentor Lsim simulator, using the ADEPT circuit simulation mode.

The ring oscillators in the Interval Timers are the most active blocks, having the highest current consumption/FET in spite of having very short internal interconnections, and accounting for about a third of the total current consumption. The controller in the Interval Timer is small and makes only a few transitions per received symbol, resulting in a surprisingly low current consumption/FET for a block at the lower levels of the receiver. The remainder of the Interval Timer circuitry operates at a wide range of frequencies, ranging from once per ring oscillator cycle to once per talkspurt, and has an intermediate average current consumption/FET. The Symbol Counter and Symbol Decoder blocks are both active only once per received symbol (every 2.25 μs), and have lower values of current consumption/FET than the Interval Timer. The RX Frame block is active mainly at talkspurt and frame boundaries, resulting in very low current consumption/FET figures.

The toolset was implemented as a mixture of commercial tools from the Mentor GDT toolset, university tools, and some custom code for netlist translation, constraint checking and tool sequencing. The custom code consists of approximately 9000 lines of C code, 5000 lines of Lisp code, and 2000 lines of Perl, shell, and nawk code. All tools were run on an HP 9000/7xx platform, but should be portable to other platforms. The tools can be obtained from the authors.

Control synthesis, from specification to netlist generation, takes a few minutes on an HP 9000/730 for a typical sized state machine.

5. Conclusions and Future Work

Our experiences designing the ABCS receiver chip confirm that asynchronous design has the potential to realize low power implementations for some applications, both because of the ability to build circuits that can be electrically static while being ready for immediate operation, and because of the ease with which different parts of a system can be made to operate at different speeds to match local processing requirements. We also demonstrated that design automation can be used to hide the complexities of asynchronous state machine implementation from the system designer, making asynchronous design feasible for commercial complexities and time-scales.

However, we found that an asynchronous designer needs skills that differ from traditional synchronous design skills. In a burst-mode design, no hazards can be tolerated on the inputs to an AFSM, since they are sensitive to transitions

at all times, even when the AFSM is not participating in the operation of the circuit. This means that when a datapath component, for example, generates a signal to be used by a state machine, the signal must be hazard-free.

Defining the overall communication scheme for an asynchronous system can also be more difficult than for synchronous design. Communication schemes in which some modules have multiple handshake interfaces need careful consideration of their relative sequencing, because the sequencing has a large impact on the overall performance of the system.

Several factors suggest the control portions of an asynchronous design should be partitioned into smaller state machines than an equivalent synchronous design. Because an AFSM does not impose a fixed input-output latency, a collection of interconnected AFSMs may have better performance than a single monolithic controller. Secondly, a single burst-mode AFSM does not allow independent parallel operations—these must be implemented using an individual AFSM for each operation. Finally, some algorithms within the current control synthesis path exhibit exponential growth with AFSM complexity, further motivating a decomposition into interconnected AFSMs. There are currently no tools to aid in the partitioning of AFSMs, and this is an area that warrants future research.

Several areas deserve further attention, in addition to control block partitioning. We did not attempt to make our design testable to production standards or to quantify the testability of our design. We expect that scan-paths could be used to improve the controllability and observability of internal nodes, although other methods would be needed to test any redundant gates added to AFSMs for hazard removal.

Our design needed some delays to be inserted before it operated with safe timing margins, highlighting the need for timing analysis tool(s) and/or a synthesis tool which accepts timing specifications to reflect the environment in which the circuit is to operate. Timing analysis is complicated by the fact that it is not possible to tell when a state machine has stabilized and is ready to accept further inputs by examining just the external ports of the state machine—the internal nodes must be examined as well. Timing analysis tools which produce the following outputs would be useful to future designs:

- The minimum delay that the environment of a given state machine should impose between each causally related pair of {AFSM output event, AFSM input event} in the state diagram.
- An HDL description of a timing checker that could be simulated in parallel with a gate level implementation of the state machine to verify that the state machine was used within its safe region during a gate level simulation.

6. Acknowledgments

Michael Spratt, of Hewlett Packard Laboratories Bristol, devised the key features of the ABCS protocol. Ken Yun, of Stanford University, and Steve Nowick, now of Columbia University, provided parts of the control synthesis path. Ed Lock, of Hewlett Packard's California Design Center, went out of his way to do the block place-and-route of the IC for us. Professors David Dill and Giovanni De Micheli of Stanford, and Drs. Al Davis and Steven Rosenberg of Hewlett-Packard have been instrumental in encouraging and assisting the collaboration between Stanford and HP, and have provided us with helpful comments on this paper.

References

[1] C. L. Seitz. "System timing." In *Mead and Conway, Introduction to VLSI Systems–Ch 7.*, pages 218–262. Addison-Wesley, 1980.

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

[3] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T. P. Fang. "Q-modules: Internally clocked delay-insensitive modules." *IEEE Transactions on Computers*, 37:1005–1018, 1988.

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

[5] W. S. Coates, A. L. Davis, and K. S. Stevens. "Automatic synthesis of fast compact self-timed control circuits." In *IFIP Workshop on Asynchronous Circuits*, Manchester, UK, 1993.

[6] S. M. Nowick and D. L. Dill. "Exact two-level minimization of hazard-free logic with multiple-input changes." In *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 626–630, 1992.

[7] S. M. Nowick and D. L. Dill. "Automatic synthesis of locally-clocked asynchronous state machines." In *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 318–321, 1991.

[8] S. M. Nowick, K. Yun, and D. L. Dill. "Practical asynchronous controller design." In *ICCD, Proceedings of the International Conference on Computer Design*, pages 341–345. IEEE Computer Society Press, 1992.

[9] P. Siegel, G. De Micheli, and D. Dill. "Automatic technology mapping for generalized fundamental-mode asynchronous designs." In *DAC, Proceedings of the Design Automation Conference*, pages 61–67, June 1993.

[10] S. M. Nowick and D. L. Dill. "Synthesis of asynchronous state machines using a local clock." In *ICCD, Proceedings of the International Conference on Computer Design*, pages 192–197. IEEE Computer Society Press, 1991.

[11] E. J. McCluskey. "Minimization of Boolean functions." *Bell Syst. tech J.*, 35(5):1417–1444, November 1956.

[12] F. Mailhot and G. De Micheli. "Algorithms for technology mapping based on binary decision diagrams and on boolean operations." *IEEE Transactions on CAD/ICAS*, pages 599–620, May 1993.

[13] L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits.* Kluwer Academic Publishers, 1993.

[14] I. Sutherland, "Micropipelines," *Communications of the ACM*, pages 720–738, June 1989.

Keywords

Asynchronous design, low-power, CAD, IR communications

Biographies

Alan Marshall has an MA in Engineering from Cambridge University, England. Since joining Hewlett Packard Laboratories in Bristol, England, he has worked in the areas of computer algorithms, networking and hardware design. Most recently he has spent two years at the Center for Integrated Systems, Stanford University, working on asynchronous design.

Bill Coates received his B.Sc. in Computer Science in 1980, and his M.Sc. in Computer Science in 1985, both from the University of Calgary, Canada. In 1984 he joined Hewlett-Packard Laboratories in Palo Alto, California, where he was engaged in research relating to multiprocessor architectures, CMOS VLSI design and asynchronous hardware synthesis. He is now with Sun Labs where he is working on the design and synthesis of asynchronous processors.

Polly Siegel is a Ph.D. candidate in Electrical Engineering at Stanford University, where her dissertation is on technology mapping for asynchronous designs. She received a Best Paper Award at the 1993 Design Automation Conference. She has also been with Hewlett-Packard since 1982, where she has been involved in research and development of a wide variety of CAD tools including schematic capture, behavioral simulation and IC design systems frameworks, and is currently working at HP Labs on asynchronous synthesis. She received her M.S. in Engineering Management from Stanford in 1994, and her M.S. and B.S. in EECS from U.C. Berkeley in 1983 and 1981, respectively. She is a member of the IEEE and the ACM.