Asynchronous Circuit Design. Chris J. Myers

ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

9

Applications

The memories of a man in his old age are the deeds of a man in his prime.
—Pink Floyd

In theory, there's no difference between theory and practice; in practice, there is.
—Chuck Reid

It's all very well in practice, but it will never work in theory.
—French management saying

An ounce of action is worth a ton of theory.
—Friedrich Engels

A theory must be tempered with reality.
—Jawaharlal Nehru

If the facts don't fit the theory, change the facts.
—Albert Einstein

In this chapter we give a brief history of the application of asynchronous design. An in-depth study is given for one recent successful asynchronous design, Intel's RAPPID instruction-length decoder. Issues in performance analysis and testing of asynchronous designs are also discussed. The chapter concludes with a discussion of the *synchronization problem.* Although this problem may have prevented the commercial application of RAPPID, it may ultimately be the problem that makes asynchronous design a necessity.

9.1 BRIEF HISTORY OF ASYNCHRONOUS CIRCUIT DESIGN

Since the early days, asynchronous circuits have been used in many interesting applications. In the 1950s and 1960s, asynchronous design was used in many early mainframe computers, including the ILLIAC and ILLIAC II designed at the University of Illinois and the Atlas and MU-5 designed at the University of Manchester. The ILLIAC and ILLIAC II were designed using the speed-independent design techniques developed by Muller and his colleagues. The ILLIAC, completed in 1952, was 10 feet long, 2 feet wide, 8 $\frac{1}{2}$ feet high, contained 2800 vacuum tubes, and weighed 5 tons. ILLIAC II was completed in 1962, and it was 100 times faster than its predecessor. This computer contained 55,000 transistors, and it could perform a floating-point multiply in 6.3 μs. The ILLIAC II used three concurrently operating controls: an arithmetic control, interplay control for data transfers, and a supervisory control called Advanced Control. The Advanced Control is responsible for fetching and storing operands, address construction and indexing, partial decoding of orders for the other controls, etc. The three controls are largely asynchronous and speed-independent. The reasons cited for using speed-independent design were increased reliability and ease of maintenance. These controls collected *reply* signals to indicate that all operations for the current step are complete before going on to the next step. The arithmetic unit was not speed-independent, as they believed that would increase its complexity and cost while decreasing its speed. The electromechanical peripheral devices were also not speed-independent, since they were inherently synchronous. This computer was used until 1967, and among its accomplishments was the discovery of three Mersenne prime numbers, including the largest then known prime number, $2^{11213} - 1$, which is over 3000 digits.

In the 1960s and 1970s at Washington University in St. Louis, asynchronous *macromodules* were developed. Macromodules are "building blocks such as registers, adders, memories, control devices, etc., from which it is possible for the electronically-naive to construct arbitrarily large and complex computers that work" [302]. The set of macromodules was designed such that they were functionally large enough to be useful, easy to interconnect to form larger computing engines, and robust enough to allow designers to worry only about logical and not electrical problems. Asynchronous design was chosen for this reason. Macromodules were used to build macromodular computer systems by placing them in a rack and interconnecting them by wires. These wires carry data signals as well as a bundled data control signal that indicates the arrival of valid data. There were also wires for control to sequence operations. Macromodular systems were developed in such a way as to be almost directly realizably from a flowchart description of an algorithm. Through this procedure many special-purpose computing engines were designed using macromodules to solve numerous problems, many from the biomedical field. Also in the 1970s, asynchronous techniques were used at the University of

Utah in the design of the first operational dataflow computer, DDM-1, and at Evans and Sutherland in the design of the first commercial graphics system.

In the late 1980s, Matsushita, Sanyo, Sharp, and Mitsubishi developed *data-driven processors*. Rather than the traditional view of a single program counter controlling the timing of instruction execution, the flow of data controls the operation speed. In other words, when all needed data arrives, it is operated on. Several signal processors were designed using this idea. Most recently, a data-driven media processor (DDMP) has been designed at Sharp capable of 2400 million signal processing operations per second while consuming only 1.32 W. Videonics has utilized the DDMP design in a high-speed video DSP. The asynchronous design of the DDMP is cited to have simplified the board layout and reduced RF interference.

In 1989, researchers at Caltech designed the first fully asynchronous microprocessor. The processor has a 16-bit datapath with 16- and 32-bit instructions. It has twelve 16-bit registers, four buses, an ALU, and two adders. The chip design consisted of about 20,000 transistors, and it was fabricated in both 2 μm and 1.6 μm MOSIS SCMOS. The 2 μm version could perform 12 million ALU instructions per second, while the 1.6 μm version could perform 18 million. The chips were shown to operate with VDD ranges from 0.35 to 7 V, and they were shown to achieve almost double the performance when cooled in liquid nitrogen. While the architecture of this design is quite simple and the results are modest, this design is significant for several reasons. First, the design is entirely *quasi-delay insensitive* (QDI), which means that it will operate correctly regardless of delays, except on specific isochronic forks. Second, the design was derived from a high-level channel description much like that described in Chapter 2. Using program transformations, ideas like pipelining were introduced. Third, the entire design took five people only five months. The group at Caltech went on to design the first asynchronous microprocessor in gallium arsenide technology. This processor ran at 100 MIPS while consuming 2 W. The most recent design from this group is an asynchronous MIPS R3000 microprocessor. This design introduced new ways to reduce the overhead of completion detection through bit-level pipelining of functional units and pipelining global completion detection. The design was fabricated in 0.6 μm CMOS, and it uses 2 million transistors, of which 1.25 million are in its caches. The measured performance ranged from 60 MIPS and 220 mW at 1.5 V and 25°C to 180 MIPS and 4 W at 3.3 V and 25°C. Running Dhrystone, the chip achieved about 185 MHz at 3.3 V.

In 1994, the AMULET group at the University of Manchester completed the AMULET1, the first asynchronous processor to be code-compatible with an existing synchronous processor, the ARM microprocessor. Their design style followed Sutherland's two-phase micropipeline idea. Their chip was fabricated in a CMOS 1 μm process and a 0.7 μm process. The performance was measured for the 1 μm part from 3.5 to 6 V using the Dhrystone benchmark code. At these voltages, the processor could complete between 15 and 25 thousand Dhrystones per second. The MIPS/watt value was also measured

to be from 175 down to 50. The chip was also shown to operate correctly between −50°C and 120°C. This project was followed up with the design of the AMULET2e, which targeted embedded system applications. In this design, an AMULET2 core is coupled with a cache/RAM, a memory interface, and other control functions on chip. One key difference is that the design used a four-phase bundled-data style rather than two-phase, as it was found to be simpler and more efficient. The design was fabricated in 0.5 μm CMOS, and its measured performance at 3.3 V was 74 kDhrystones, which is roughly equivalent to 42 MIPS. At this peak rate, it consumes 150 mW. One other interesting measurement they performed was the radio-frequency emission spectrum or EMC. Results show significant spread of the peaks as compared with a clocked system. One last interesting result was that the power consumed when the processor enters "halt" mode drops to under 0.1 mW. The most recent design by this group is the AMULET3, which has incorporated ARM thumb code and new architectural features to improve performance.

In 1994, a group at SUN suggested that replicating synchronous architecture may not be the best way to demonstrate the advantages of asynchronous design. Instead, they suggested a radically different architecture, the *counterflow pipeline* in which instructions are injected up the pipeline while contents of registers are injected down. When an instruction meets the register values it needs, it computes a result. Key to the performance of such a design are circuits to move data very quickly. This group has subsequently designed several very fast FIFO circuits. Their test chips fabricated in 0.6 μm CMOS have been shown to have a maximum throughput of between 1.1 and 1.7 Giga data items per second.

Philips Research Laboratories has designed numerous asynchronous designs targeting low power. This group developed a fully automated design procedure from a hardware specification in their language TANGRAM to a chip, and they have applied this procedure to rapidly design several commercially interesting designs. In 1994, this group produced an error corrector chip for the digital compact cassette (DCC) player that consumed only 10 mW at 5 V which is one-fifth of the power consumed by its synchronous counterpart. This design also required only 20 percent more area. In 1997, they designed asynchronous standby circuits for a pager decoder which dissipated four times less power and are only 40 percent larger than their comparable synchronous design. Since the standby circuit in modern pagers is responsible for a substantial portion of the power consumption while using only a small amount of silicon area, this result maps to a 37 percent decrease in power for the entire pager while showing only a negligible overall area increase. In 1998, this group designed an 80C51 microcontroller. A chip fabricated in 0.5 μm CMOS was shown to be three to four times more power efficient than its synchronous counterpart, consuming only 9 mW when operating at 4 MIPS. Perhaps the most notable accomplishment to come out of this group is a fully asynchronous pager being sold by Philips that uses the standby circuits and the 80C51 microcontroller just described. While the power savings reported

was important, the major reason for Philips to use the asynchronous pager design is the fact that the asynchronous design had an emission spectrum more evenly spread over the frequency range. Due to interference produced at the clock frequency and its harmonics, the synchronous version shuts off its digital circuitry as a message is received so as not to interfere with the RF communication. The spread-spectrum emission pattern for the asynchronous design allowed the digital circuitry to remain active as the message is received. This permitted their pager to be capable of being universal in that it can accept all three of the international pager standards.

Another recent asynchronous application came out of the RAPPID project at Intel conducted between 1995 and 1999. RAPPID is a fully asynchronous instruction-length decoder for the PentiumII 32-bit MMX instruction set. The chip design in this project achieved a three fold improvement in speed and a two fold improvement in power compared with the existing synchronous design. This design is described in some detail in the following section.

9.2 AN ASYNCHRONOUS INSTRUCTION-LENGTH DECODER

RAPPID (Revolving Asynchronous Pentium Processor Instruction Decoder) is a fully asynchronous instruction-length decoder for the complete PentiumII 32-bit MMX instruction set. In this instruction set, each instruction can be from 1 to 15 bytes long, depending on a large number of factors. In order to allow concurrent execution of instructions, it is necessary to rapidly determine the positions of each instruction in a cache line. This was at the time a critical bottleneck in this architecture. A partial list of the rules that determine the length of an instruction is given below.

- Opcode can be 1 or 2 bytes.
- Opcode determines presence of the ModR/M byte.
- ModR/M determines presence of the SIB byte.
- ModR/M and SIB set length of displacement field.
- Opcode determines length of immediate field.
- Instructions may be preceded by as many as 15 prefix bytes.
- A prefix may change the length of an instruction.
- The maximum instruction length is 15 bytes.

For real applications, it turns out that there are only a few common instruction lengths. As shown in the top graph from Figure 9.1, 75 percent of instructions are 3 bytes or less in length. Nearly all instructions are 7 bytes or less. Furthermore, prefix bytes that modify instruction lengths are extremely rare. This presents an opportunity for an asynchronous design to optimize for the common case by optimizing for instructions of length 7 or less with no

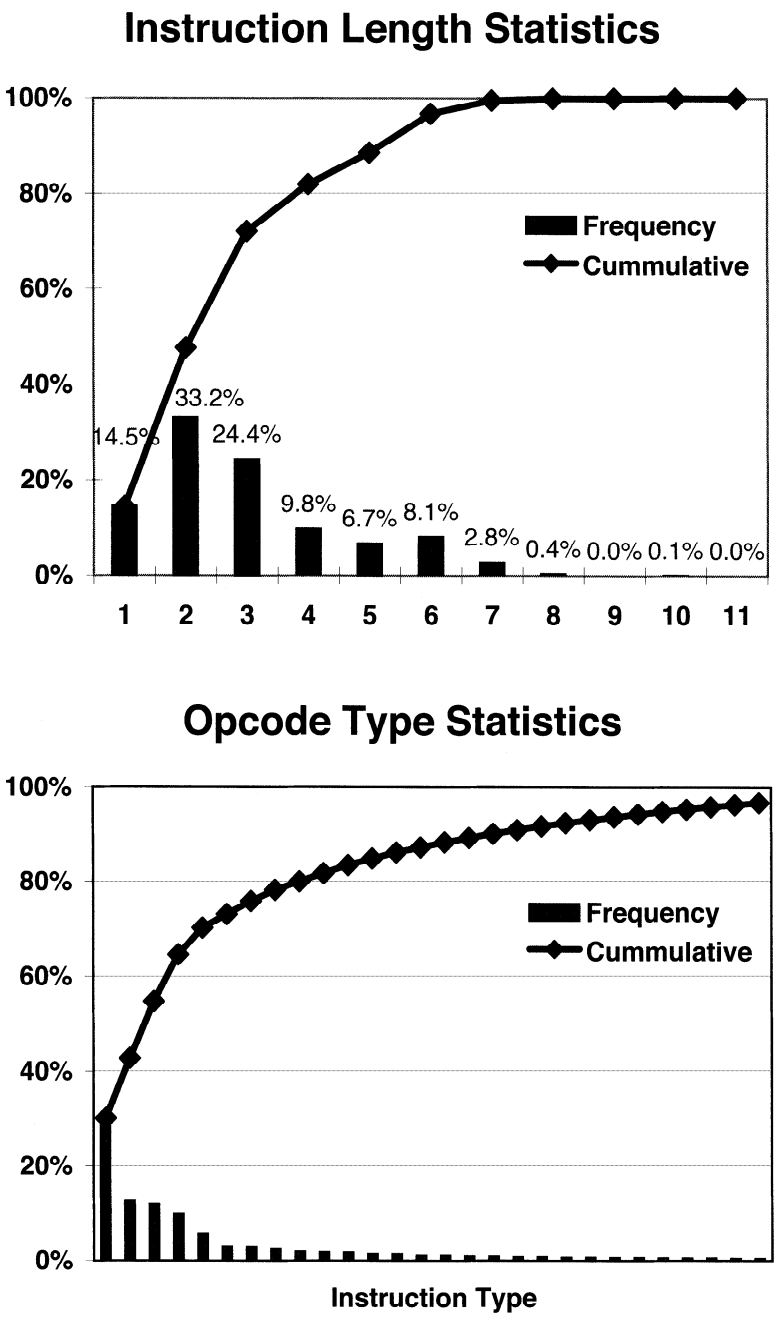

Fig. 9.1 Histogram for proportion of instruction lengths and cumulative length statistics, and histogram for proportion of instruction types and cumulative statistics.

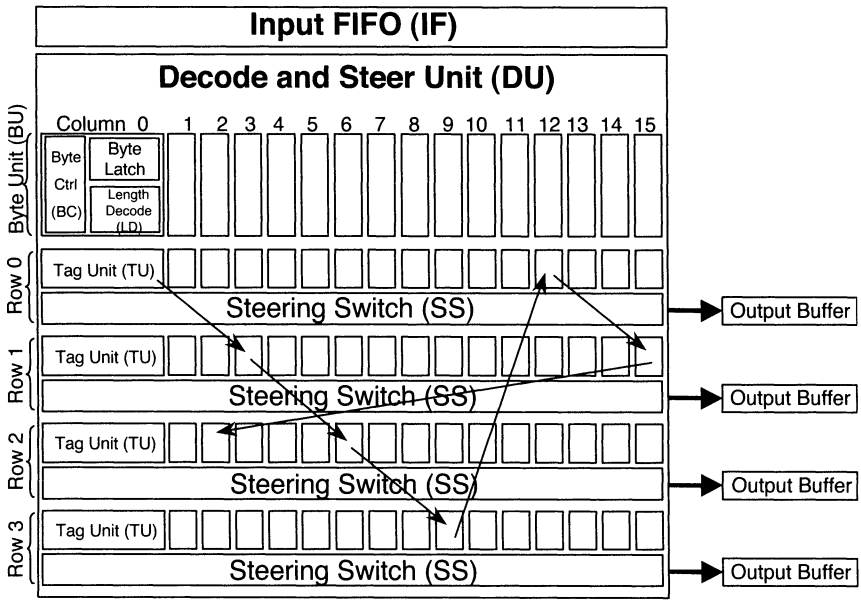

Fig. 9.2 RAPPID microarchitecture.

prefix bytes. Other less efficient methods are then used for longer instructions and instructions with prefix bytes.

The RAPPID microarchitecture is shown in Figure 9.2. The RAPPID decoder reads in a 16-byte cache line, and it decodes each byte as if it is the first byte of a new instruction. Each byte speculatively determines the length of an instruction beginning with this byte. It does this by looking at three additional downstream bytes. The actual first byte of the current instruction is marked with a tag. This byte uses the length that it determined to decide which byte is the first byte of the next instruction. It then signals that byte while notifying all bytes in between to cancel their length calculations and forwards the bytes of the current instruction to an output buffer. To improve performance, four rows of tag units and output buffers are used in a round-robin four-issue fashion.

As shown in benchmark analysis at the bottom of Figure 9.1, only a small number of instruction types are common. In fact, 15 percent of the opcode types occur 90 percent of the time. This provides another opportunity for an asynchronous design to be optimized for the common case. Each length decode unit is essentially a large, complex PLA structure. Using the statistics shown in Figure 9.1, it is possible to restructure the combinational logic to be faster for the common case. Consider the logic shown in Figure 9.3(a). Assume that gate A causes the OR gate to turn on 90 percent of the time, gate B 6 percent, and gate C 4 percent. If the delay of a two-input gate is

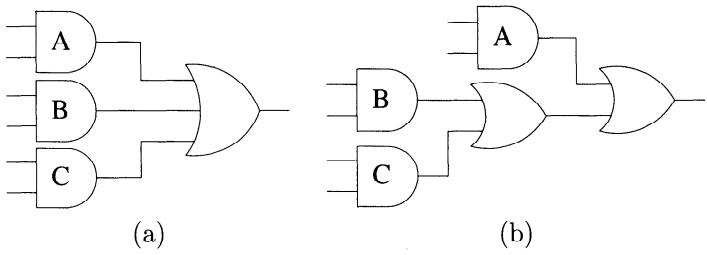

Fig. 9.3 (a) Balanced tree logic function. (b) Unbalanced tree logic function.

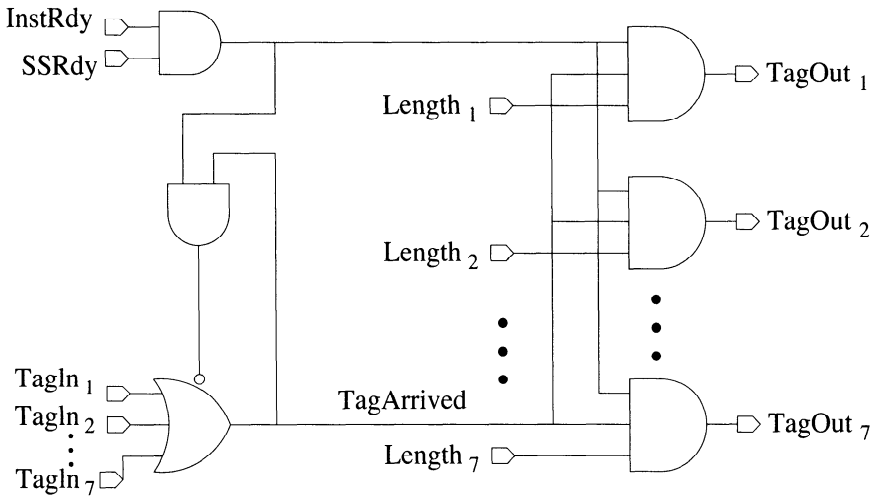

Fig. 9.4 Tag unit circuit.

1 and a three-input gate 1.5, the delay of the logic shown in Figure 9.3(a) is 2.5. However, the logic shown in Figure 9.3(b) has an average-case delay of only 2.1. The length decode logic takes advantage of this idea, and it is implemented using large unbalanced trees of domino logic that have been optimized for common instructions.

The key to achieving high performance is the tag unit, which must be able to rapidly tag instructions. The timed circuit for one tag unit is shown in Figure 9.4. Assuming that the instruction is ready (i.e., *InstRdy* is high, indicating that one $Length_i$ is high and all bytes of the instruction are available) and the steering switch is ready (i.e., *SSRdy* is high), then when a tag arrives (i.e., one of $TagIn_j$ is high), the first byte of the next instruction is tagged (i.e., $TagOut_i$ is set to high). In the case of a branch, the tag is forwarded to a branch control circuit which determines where to inject the tag back into the new cache line.

If the instruction and steering switch are ready when a column gets tagged, it takes only two gate delays from TagIn to TagOut. In other words, a synchronization signal can be created every two gate delays. It is difficult to imagine distributing a clock which has a period of only two gate delays. The tag unit in the chip is capable of tagging up to 4.5 instructions/ns.

This circuit, however, requires timing assumptions for correct operation. In typical asynchronous communication, a request is transmitted followed by an acknowledge being received to indicate that the circuit can reset. In this case, there is no explicit acknowledgment, but rather, acknowledgment comes by way of a timing assumption. Once a tag arrives (i.e., *TagArrived* is high), if the instruction and steering switch are ready, the course is set to begin to reset *TagArrived.* The result is that the signal produced on $TagOut_i$ is a pulse. Let us consider now the effect of receiving a pulse on a *TagIn* signal. If either the instruction or steering switch are not ready, then *TagArrived* gets set by the pulse, in effect latching the pulse. *TagArrived* will not get reset by the disappearance of the pulse but rather the arrival of a state in which both the instruction and steering switch are ready.

For this circuit to operate correctly, there are two critical timing assumptions. First, the pulse created must be long enough to be latched by the next tag unit. This can be satisfied by adding delay to the AND gate used to reset *TagArrived.* An arbitrary amount of delay, however, cannot be added since the pulse must not also be so long that another pulse could come before the circuit has reset. Therefore, we have a *two-sided timing constraint.* Analysis methods such as those described in Chapter 7 are needed to verify that this design operates correctly.

The RAPPID test chip was fabricated in May 1998 using a 0.25 μm CMOS process. The test chip was capable of decoding and steering instructions at a rate of 2.5 to 4.5 instructions per nanosecond. This is about three times faster than the peak performance of the fastest synchronous three-issue product in the same fabrication process clocked at 400 MHz which is capable of only 1.2 instructions per nanosecond. The chip operated correctly between 1.0 and 2.5 V while the synchronous design could only tolerate about 1.9 to 2.1 V. The RAPPID design also consumes only one-half of the energy of the clocked design. The RAPPID design was found to achieve these gains with only a 22 percent area penalty over the clocked design.

9.3 PERFORMANCE ANALYSIS

One major difficulty in designing an asynchronous circuit such as RAPPID is determining its performance. It is not simply a matter of finding the critical path delay or counting the number of clock cycles per operation. One of the major driving forces behind the design of RAPPID is optimizing for the common case. This means that worst-case analysis as done for synchronous

design may actually be quite pessimistic, as our goal is to achieve high rates of performance on average.

To address this problem, one must take a probabilistic approach to performance analysis. Consider, for example, the TEL structure model described in Chapter 4. In a TEL structure, the delay between two events is modeled as a range $[l, u]$, where l is a lower bound and u is an upper bound of delay. For purposes of the performance analysis, it is necessary to extend this model to include a distribution function for this delay. One simple approach is to simply assume that the delay falls uniformly in this range. If this design is a timed circuit, though, where the correctness may depend on the delay never stepping out of this range, a uniform assumption is hopefully not realistic. Another possibility would be to use a more interesting distribution such as a truncated Gaussian.

Once a probabilistic delay model has been chosen, the next step is to use it to determine performance. The most direct approach is a Monte Carlo simulation.

Example 9.3.1 Consider the timed circuit shown in Figure 7.38 and the delay numbers given in Table 7.1. Let us also assume that although theoretically the patron may have infinite response time, in practice it rarely takes him more than 10 minutes to come fetch the wine. If we take the standard synchronous performance analysis approach of using just the worst-case delay, we find that the cycle time of this circuit is 18.3 minutes. On the other hand, if we consider each delay to be distributed within its range using a truncated Gaussian with a mean in the middle and a standard deviation of one-fourth of the range, we find the cycle time to be 14.2 minutes. Since an asynchronous circuit operates at its average rate, this is a more true reflection of the actual performance.

9.4 TESTING ASYNCHRONOUS CIRCUITS

Another major obstacle to the commercial acceptance of asynchronous circuits, such as RAPPID, is the perception that they are more difficult to test. Once a chip has been fabricated, it is necessary to test it to determine the presence of manufacturing defects, or *faults*, before delivering the chip to the consumer. For asynchronous circuits, this is complicated by the fact that there is no global clock which can be used to single step the design through a sequence of steps. Asynchronous circuits also tend to have more state holding elements, which increases the overhead needed to apply and examine test vectors. Huffman circuits employ redundant circuitry to remove hazards, and redundant circuitry tends to hide some faults, making them untestable. Finally, asynchronous circuits may fail due to glitches caused by *delay faults*, which are particularly difficult to detect.

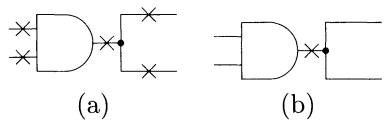

Fig. 9.5 (a) Potential fault locations. (b) Output stuck-at fault model.

However, it's not all bad news for asynchronous circuits. Since many asynchronous styles use handshakes to communicate data, for many possible faults, a defective circuit simply halts. In particular, in the *stuck-at fault model*, a defect is assumed to cause a wire to become permanently *stuck-at-0* or *stuck-at-1*. If an acknowledge wire is stuck-at-0, the corresponding request is never acknowledged, causing the circuit to stop and wait forever. This is obviously easy to detect with a timeout. In this case, the circuit is said to be *self-checking*. Since in a delay-insensitive circuit every transition must be acknowledged by the receiver of the transition, delay-insensitive circuits will halt in the presence of any stuck-at fault on any wire in the design.

As shown in Chapter 1, the class of circuits that can be designed using the delay-insensitive model is very limited. Unfortunately, for a more general class of designs such as Muller circuits designed under the speed-independent model, the circuit may not halt for all stuck-at faults. For the gate shown in Figure 9.5(a), a stuck-at fault can occur at any of the five locations marked with an $\times$. Under this model, not all faults cause a Muller circuit to halt. In the *output stuck-at fault model*, there is only one possible fault location shown in Figure 9.5(b). Since in a Muller circuit, a transition on an output must be acknowledged by one of the two branches of the isochronic fork, a fault at this location results in the circuit halting. Therefore, Muller circuits do halt for any output stuck-at fault.

Some Muller circuits such as the one shown in Figure 9.6(a) do halt for all stuck-at faults. Others, however, such as the one in Figure 9.6(b) do not. In these cases, the fault can cause a *premature firing*.

> **Example 9.4.1** Consider the fault where *r1* is stuck-at-0 in the circuit shown in Figure 9.6(b). Assume that all signals start out initially low. After *req_wine* goes high, x goes high, allowing *req_patron* to go high since *r1* is stuck-at-0. This is a premature firing since *req_patron* is not supposed to go high until after *ack_wine* goes high, followed by *req_wine* going low. Similarly, if *a2* is stuck-at-0, it can cause *ack_wine* to go low prematurely (i.e., before *ack_patron* goes low).

It is clear from these examples that the problem is due to isochronic forks. If we consider all wire forks and determine those which must be isochronic for correct circuit operation, we can determine a fault model which lies between the output stuck-at fault model and the more general stuck-at fault model. Namely, in the *isochronic fork fault model*, faults can be detected on

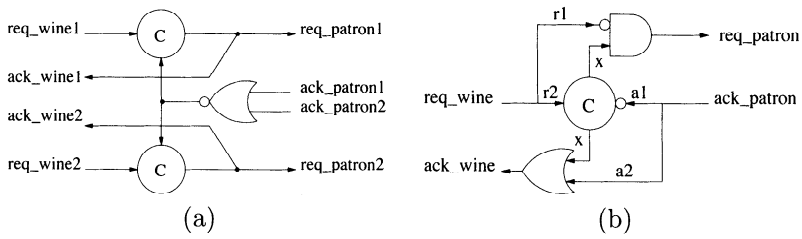

Fig. 9.6 (a) Circuit in which all stuck-at faults cause it to halt. (b) Circuit in which some stuck-at faults cause premature firings.

all branches of a nonisochronic fault by the circuit halting, but only on the input to forks that are isochronic.

Granted even the general stuck-at fault model is too simplistic to truly capture the symptoms of manufacturing defects. However, more complex *delay fault* and *bridging fault* models have been successfully adapted to asynchronous circuits. These topics, though, are beyond the scope of this short discussion.

Once we have decided on a fault model, the next step is to add any necessary circuitry to apply and analyze test vectors, such as *scan paths*. We must also generate a sufficient set of test vectors to guarantee a high degree of coverage of all possible faults in our model. As in models, the main hurdle to testing asynchronous circuits appears to be that the traditional synchronous testing methods do not work right off the shelf. Again, however, many of the popular methods of test have been adapted to the asynchronous design problem.

9.5 THE SYNCHRONIZATION PROBLEM

Despite the excellent results demonstrated by the RAPPID design, it has not been used in a commercial product. One important reason is that the asynchronous design must communicate with the rest of the microprocessor that operates synchronously. Unfortunately, this is difficult to do reliably without substantial latency penalties. When this latency penalty is taken into account, most, if not all, of the performance advantage gained by the RAPPID design is lost.

Consider again our wine patron shopping from two shops, one that sells chardonnay and another that sells merlot. When one shop calls him, he immediately heads off to that shop to buy the wine. What is he to do if they both notify him at nearly the same instant that they have a fresh bottle of wine to sell? He must make a choice. However, if both types of wine are equally appealing to him at that instant, he may sit there pondering it for some time. However, if the shopkeepers get impatient, they have a tendency to drink the wine themselves. Therefore, if he is really indecisive and cannot make up

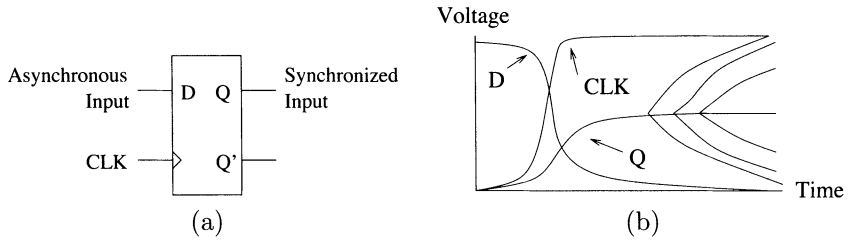

Fig. 9.7 (a) Simple, dangerous synchronizer. (b) Oscilloscope view of metastable behavior.

his mind for too long, he will not get either bottle of wine. This state in which the patron is stuck considering two equally appealing choices is called a *metastable state.* If this state persists so long that something bad happens (like the shopkeepers drinking his wine), this is called a *synchronization failure.*

In a circuit, this can happen when a synchronous circuit must synchronize an asynchronous input. This can be done using a single D-type flip-flop as shown in Figure 9.7(a). However, if the clock edge arrives too close in time to data arriving from an asynchronous circuit, the circuit may enter a metastable state in which its output is at neither a logic 0 or logic 1 level, but rather, lies somewhere in between. This behavior is depicted in Figure 9.7(b). Assume that Q is initially low and that D has recently gone high. If D goes low again at about the same time that CLK rises, the output Q may start to rise and then get stuck between the logic levels as it observes D falling. Should Q rise or fall? Actually, either answer would be okay, but the flip-flop becomes indecisive. At some point, Q may continue to a logic 1 level, or it may drop to the logic 0 level. When this happens, however, is theoretically unbounded. If during this period of indecision, a circuit downstream from this flip-flop looks at the synchronized input, it will see an indeterminate value. This value may be interpreted by different subsequent logic stages as either a logic 0 or a logic 1. This can lead the system into an illegal or incorrect state, causing the system to fail. Such a failure is traditionally called a synchronization failure. If care is not taken, the integration of asynchronous modules with synchronous modules can lead to an unacceptable probability of failure. Even if no asynchronous modules are used, synchronous modules operating at different clock rates or out of phase can have the same problem. The latter problem is becoming more significant as it becomes increasingly difficult to distribute a single global clock to all parts of the chip. Many designers today are considering the necessity of having multiple clock domains on a single chip, and they will need to face this problem.

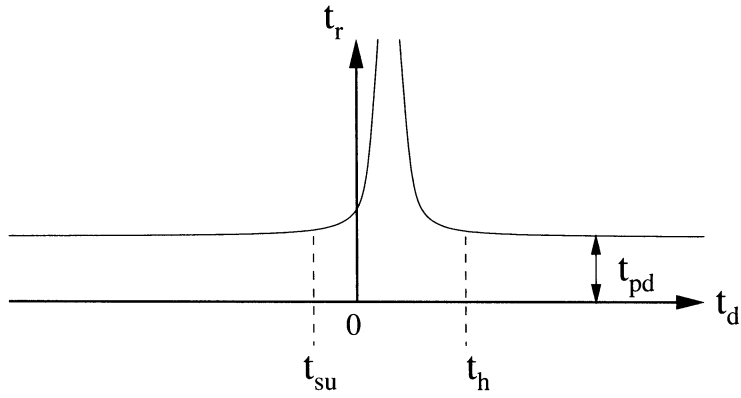

Fig. 9.8 Flip-flop response time as a function of input arrival time in relation to clock arrival time (clock arrives at time 0).

9.5.1 Probability of Synchronization Failure

Figure 9.8 shows a representative plot based on measured data for the response time of a flip-flop as a function of the arrival time, t_d, of data with respect to the clock. If data only changes before the *setup time*, t_{su}, and after the *hold time*, t_h, of a flip-flop, the *response time*, t_r, is roughly constant and equal to the *propagation delay* through the flip-flop, t_{pd}. If, on the other hand, the data arrives between the setup and hold times, the delay increases. In fact, if the data arrives at just the absolutely wrong time, the response time is unbounded.

If data can arrive asynchronously with respect to the clock, we can consider that it arrives at a time which is uniformly distributed within the clock cycle. Therefore, the probability that the data arrives at a time t_d which falls between t_{su} and t_h is given below.

$$P(t_d \in [t_{su}, t_h]) = \frac{t_h - t_{su}}{T} \tag{9.1}$$

where T is the length of the clock period. If we assume that the flip-flop is given some bounded amount of time, t_b, to decide whether or not to accept the newly arrived data, the probability of a synchronization failure is related to the probability that the response time, t_r, exceeds t_b. In the metastable region between t_{su} and t_h, it can be shown that the response time increases in an approximately exponential fashion. Therefore, if t_d falls in this range, the probability that $t_r > t_b$ can be expressed as follows:

$$P(t_r > t_b \mid t_d \in [t_{su}, t_h]) = \frac{1}{k + (1-k)e^{(t_b - t_{pd})/\tau}} \tag{9.2}$$

where k and τ are circuit parameters, with k being a positive fraction less than 1 and τ being a time constant with values on the order of a few picoseconds

for modern technologies. Combine Equations 9.1 and 9.2 using Bayes rule:

$$P(t_r > t_b) = P(t_d \in [t_{su}, t_h]) \cdot P(t_r > t_b \mid t_d \in [t_{su}, t_h]) \quad (9.3)$$

$$= \frac{t_h - t_{su}}{T} \cdot \frac{1}{k + (1-k)e^{(t_b - t_{pd})/\tau}} \quad (9.4)$$

If $t_b - t_{pd} \geq 5\tau$, Equation 9.3 can be simplified as follows:

$$P(t_r > t_b) \approx \frac{t_h - t_{su}}{T} \cdot \frac{e^{-(t_b - t_{pd})/\tau}}{1-k} \quad (9.5)$$

By combining constants, Equation 9.5 can be changed to

$$P(t_r > t_b) \approx \frac{T_0}{T} \cdot e^{-t_b/\tau} \quad (9.6)$$

Equation 9.6 is convenient since there is only two circuit-dependent parameters T_0 and τ that need to be determined experimentally. These parameters appear to scale linearly with feature size. Equation 9.6 has been verified experimentally and found to be a good estimate as long as t_b is not too close to t_{pd}. It is important to note that there is no finite value of t_b such that $P(t_r > t_b) = 0$. Therefore, the response time in the worst-case is unbounded.

A *synchronization error* occurs when t_r is greater than the time available to respond, t_a. A synchronization failure occurs when there is an inconsistency caused by the error. Failures occur less often than errors since a consistent interpretation still may be made even when there is an error. The expected number of errors is

$$E_e(t_a) = P(t_r > t_a) \cdot \lambda \cdot t \quad (9.7)$$

where λ is the average rate of change of the signal being sampled and t is the time over which the errors are counted. If we set $E_e(t_a)$ to 1, change t to MTBF (*mean time between failure*), substitute Equation 9.6 for $P(t_r > t_a)$, and rearrange Equation 9.7, we get

$$\text{MTBF} = \frac{T \cdot e^{t_a/\tau}}{T_0 \cdot \lambda} \quad (9.8)$$

This equation increases rapidly as t_a is increased. Therefore, even though there is no absolute bound in which no failure can ever occur, there does exist an engineering bound in which there is an acceptably low likelihood of error.

9.5.2 Reducing the Probability of Failure

Many techniques have been devised to address the metastability problem and reduce the probability of synchronization failure to an acceptable level when interfacing between synchronous and asynchronous modules. The goal of each

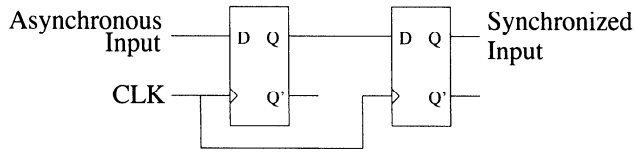

Fig. 9.9 Double latch solution to reduce synchronization failure.

of these techniques is to increase the amount of time to resolve the metastability (i.e., increase t_a). The simplest approach to achieve this is to use two (or more) latches in series as shown in Figure 9.9 to sample asynchronous signals arriving at a synchronous module. This increases the time allowed for a metastable condition to resolve. In other words, if n extra latches are added in series with an asynchronous input, the new value of t_a is given by

$$t'_a = t_a + n(T + t_{pd}) \tag{9.9}$$

where T is the clock period and t_{pd} is the propagation delay through the added flip-flops. The cost, though, is an extra n cycles of delay when communicating data from an asynchronous module to a synchronous module, even when there is no metastability. This scheme also only minimizes the probability and does not eliminate the possibility of synchronization failure, as there is still some chance that a metastable condition could persist longer than n clock cycles.

Example 9.5.1 Assume that τ is measured to be about 20 ps and T_0 about 8 ns. If the clock frequency is 2 GHz, T is 500 ps. If asynchronous inputs are coming at an average rate of 1 GHz, λ is 10^9 samples per second. Let us also assume that we can tolerate a metastability for four-fifths of the clock period or $t_a = 400$ ps. Using Equation 9.8 we find the mean time between failures to be only 30 ms! If the propagation delay through a flip-flop is 120 ps and we add a second latch, then t_a becomes 780 ps, and the mean time between failure becomes about 63 days. If we add a third flip-flop, the mean time between failure increases to over 30 million years!

9.5.3 Eliminating the Probability of Failure

To eliminate synchronization failures completely, it is necessary to be able to force the synchronous system to wait an arbitrary amount of time for a metastable input to stabilize. In order for the synchronous circuit to wait, it is necessary for the asynchronous module to be able to cause the synchronous circuit's clock to stop when it is either not ready to communicate new data or not ready to receive new data. A *stoppable clock* can be constructed from a gated ring oscillator as shown in Figure 9.10. The basic operation is that when the *RUN* signal is activated, the clock operates at a nominal rate set by the number of inverters in the ring. To stop the clock, the *RUN* signal must be deactivated between two rising clock edges. The clock restarts as soon as

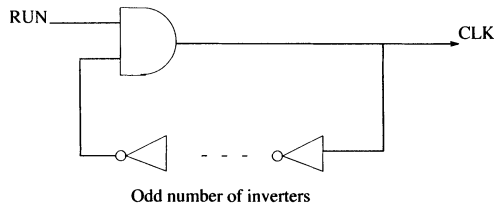

Fig. 9.10 Stoppable ring oscillator clock.

the *RUN* signal is reactivated. In other words, the clock should be stopped synchronously and is restarted asynchronously.

If the synchronous module decides when it needs data from the asynchronous module, the behavior is as follows. When the synchronous module needs data from an asynchronous module, it can request the data on the rising edge of the clock and in parallel set *RUN* low. If you want a guaranteed high pulse width, then *RUN* must be set low on the falling clock edge. When the data arrives from the asynchronous module, the acknowledgment from this module can be used to set *RUN* high. If *RUN* is set high before the end of the clock cycle, the next clock cycle can begin again without delay. If on the other hand, the asynchronous module is slow in providing data, the low phase of *CLK* will be stretched until the data arrives.

If the asynchronous module can decide when to send data, a *mutual exclusion* (ME) element is needed as shown in Figure 9.11 to guarantee that a synchronous module either receives data from an asynchronous unit or a pulse from the clock generator, but never both at the same time. If the asynchronous data arrives too close to the next clock pulse, both the data and the clock pulse may be delayed waiting for the metastability to resolve before determining which is to be handled first. An ME element has two inputs, *R1* and *R2*, and two outputs, *A1* and *A2*. It can receive rising transitions on both inputs concurrently, but it will respond with only a single rising transition on one of the corresponding outputs. There are three possible situations. The first is that the asynchronous module does not request to send data during this clock cycle. In this case, the ME simply acts as a buffer and the next rising clock edge is produced. The second case is the asynchronous request comes before the next rising clock edge is needed. In this case, the ME issues an *ACK* to the asynchronous module, and it prevents the next clock cycle from starting until *REQ* goes low. The third case is that *REQ* goes high just as *CLK* is about to rise again. This causes a metastable state to occur, but the ME is guaranteed by design to either allow the asynchronous module to communicate by setting *ACK* high and stopping the clock or by refusing to acknowledge the asynchronous module this cycle and allowing *CLK* to rise. Note that theoretically it may do neither of these things for an unbounded amount of time.

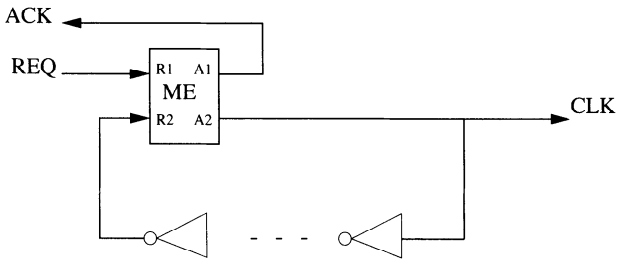

Fig. 9.11 Stoppable ring oscillator clock with ME.

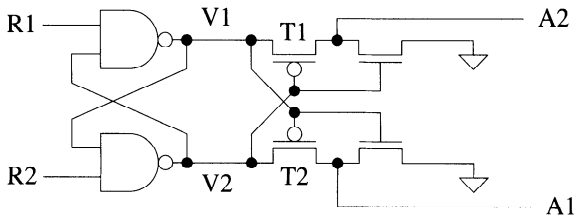

Fig. 9.12 Circuit for mutual exclusion.

A circuit diagram for a CMOS ME element is shown in Figure 9.12. When this circuit goes metastable, *V1* and *V2* differ by less than a threshold voltage, so *T1* and *T2* are off. Therefore, both *A1* and *A2* remain low. Once *V1* and *V2* differ by more than a threshold voltage, either *T1* or *T2* will turn on, pulling up its corresponding output.

A stoppable clock can be used to design a *globally asynchronous locally synchronous* (GALS) architecture. Communication between modules is done asynchronously using request/acknowledge protocols while computation is done synchronously within the modules using a locally generated clock. The basic structure of such a module is shown in Figure 9.13. The module's internal clock is stopped when it must wait for data to arrive from, or to be accepted by, other modules. If an asynchronous module can request to communicate data to a synchronous module at arbitrary times as discussed above, the stoppable clock shown in Figure 9.11 is needed. If the synchronous unit determines when data is to be transferred to/from the asynchronous modules, there is no need for a ME element, since the decision to wait on asynchronous communication is synchronized to the internal clock. In this case, the stoppable clock shown in Figure 9.10 can be used.

A *globally synchronous locally asynchronous* architecture is shown in Figure 9.14. One possible approach to increasing a synchronous, pipelined microprocessor's speed is to replace the slowest pipeline stages with asynchronous modules that have a better average-case performance, RAPPID for example. If the interfacing problem can be addressed, this allows a performance gain

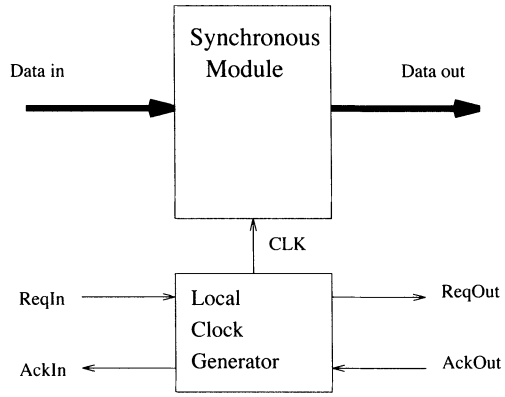

Fig. 9.13 Basic module of a GALS architecture.

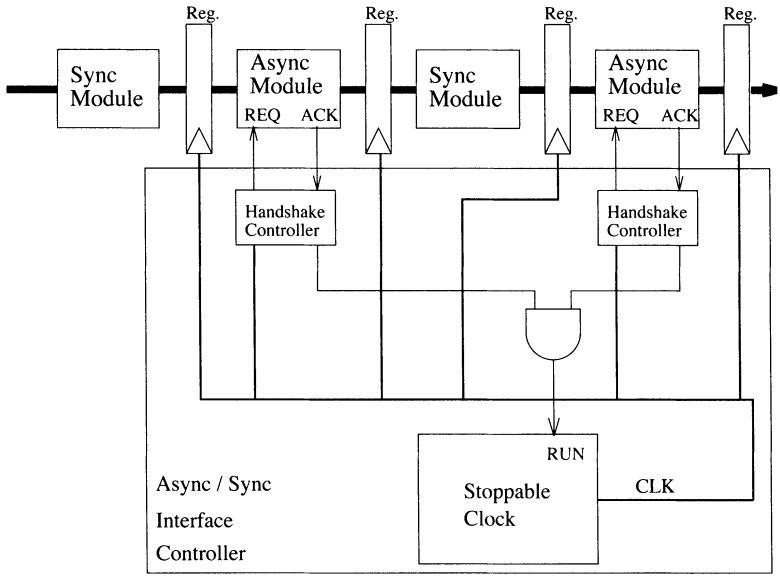

Fig. 9.14 Architecture for a globally synchronous locally asynchronous system.

without redesigning the entire chip. While the entire system communicates synchronously, one or more local modules may compute asynchronously. In other words, the system is globally synchronous locally asynchronous.

This interface methodology, while similar to the GALS architecture, allows for stages in high-speed pipelines to be either synchronous or asynchronous. The architecture shown in Figure 9.14 assumes true single-phase clocking configured in such a way that data is latched into the next stage on the rising edge of the clock. The *CLK* signal is generated using a stoppable ring

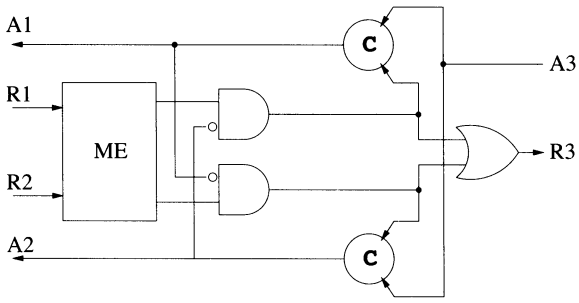

Fig. 9.15 Circuit for arbitration.

oscillator. Besides being used to sequence data between pipeline stages, the *CLK* signal is also used to generate the handshake protocol that controls the asynchronous modules. The interface controller is composed of the stoppable clock generator, one handshake control circuit for each asynchronous module, and an *AND* gate to collect the *ACK* signals to generate the *RUN* signal. The circuit behavior of the interface controller is as follows. Shortly after the rising edge of the *CLK* signal, the *RUN* signal is set low. The *RUN* signal is set high again only after all the asynchronous modules have completed their computation. Since data moves in and out of each asynchronous module with every cycle in the pipeline, no mutual exclusion elements are necessary.

9.5.4 Arbitration

A closely related problem to synchronization is *arbitration*. Arbitration is necessary when two or more modules would like mutually exclusive access to a shared resource. If the modules requesting the resource are asynchronous with respect to each other, arbitration also cannot be guaranteed to be accomplished in a bounded amount of time. If all modules are asynchronous, however, it is possible to build an *arbiter* with zero probability of failure.

An arbiter should be capable of receiving requests from multiple sources, select one to service, and forward the request to the resource requiring mutually exclusive access. An arbiter circuit for two requesting modules is shown in Figure 9.15. If this circuit receives a request on both *R1* and *R2* concurrently, the ME element selects one to service by asserting one of its two output wires. This in turn causes the resource to be requested by asserting *R3*. When the resource has completed its operation, it asserts *A3* and the acknowledgment is forwarded to the requesting module that was chosen.

Larger arbiters can be built from two-input arbiters by connecting them up in the form of a tree. These trees can be either balanced or unbalanced, as shown in Figure 9.16 (note that the symbol that looks like either a sideways "A" or an eye watching its inputs means an arbiter). If it is balanced as shown in Figure 9.16(a) for a four-way arbiter, all modules have equal priority access

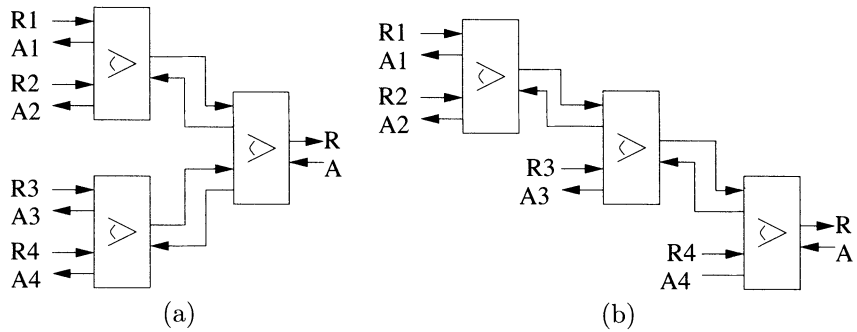

Fig. 9.16 (a) Balanced four-way arbiter. (b) Unbalanced four-way arbiter.

to the resource. If it is unbalanced as shown in Figure 9.16(b), modules closest to the end get a higher priority. In this example, if all four requests are high, *R4* has a 50 percent chance of obtaining the resource, *R3* has a 25 percent chance, and *R2* and *R1* each have only a 12.5 percent chance.

9.6 THE FUTURE OF ASYNCHRONOUS CIRCUIT DESIGN

Attempting to predict the future is an exercise that should be avoided. In reviewing the literature during the preparation of this book, papers in the 1960s and 1970s espouse many of the same advantages for asynchronous design that show up in the papers published today. Why after all these years has asynchronous circuit design not taken over the world as it has been widely predicted? The trouble is there are a lot of very smart people who have been capable of overcoming each hurdle that synchronous design has faced.

If asynchronous design is so much better, why do designers continue to struggle with synchronous design? The reason is that the synchronous design paradigm is so simple to understand and use. The trouble with this reasoning, though, is that in high-performance circuit design, the simple synchronous design paradigm does not truly exist anymore. Carl Anderson, an IBM fellow who led the design of a recent 170 million-transistor PowerPC design, stated in a talk at Tau2000, "Clocking is a major pain!" He went on to state that all future high-performance microprocessors must use multiple clocking domains. In modern technologies, the time of flight of data (not to mention the clock signal) between areas of the chip can take multiple clock periods.

So when is the asynchronous revolution going to take place? Actually, I do not believe it will. Rather, what we see happening is that as global synchrony is becoming ever more difficult, people are rediscovering ideas like GALS as evidenced by the recent IPCMOS project at IBM. Once designers start to think of communication being asynchronous between the synchronous modules they

are comfortable with, it is only a matter of time before some asynchronous modules infiltrate the ranks. This would allow designs such as RAPPID where high performance is achieved using asynchronous design on only a single module to be incorporated in a larger system. There are, however, already some domains, such as those requiring low EMI, where asynchronous design has shown its merit as evidenced by Philip's asynchronous pager.

What will we do if asynchronous design never takes over the world? We can be comforted by the fact that the issues that we face in learning how asynchronous design is to be done are important even in today's synchronous design problems. For example, the elimination of hazards has shown to be worthwhile for reducing power consumption, and it is also a necessity in several synchronous design styles. Also, the two-sided timing issues faced in the design of timed circuits are also encountered by the designers of aggressive circuit styles such as *delayed-reset* and *self-reseting* domino circuits being experimented with today.

That being said, we would still like to see it take over the world, so what is the main obstacle to this? The major hurdle that remains, in this author's opinion, is fear. Designers do not fully understand the asynchronous design problem. In the past, this has led designers to attempt asynchronous design without all the tools and background, only to become disillusioned when unforeseen problems lead to the design's failure. This can be overcome through education, and it is the hope of this author that this book will play an important role in bringing this about.

9.7 SOURCES

There is little material available about the design of the ILLIAC computer. A useful reference on the design of the ILLIAC II is an article by Brearley [46] which serves as an annotated bibliography listing 40 papers and technical reports describing the ILLIAC II design and the speed-independent design methodology used in its design. We have been unable to find any references that discuss the circuit design of the Atlas and MU-5 mainframe computers which were completed and used at the University of Manchester in 1962 and 1966, respectively. The macromodule project at Washington University in St. Louis is described in some detail in papers by Clark and others [87, 88, 302, 369]. The DDM-1 is described by Davis in [102, 103]. The only documentation of the LDS-1, the graphics system designed at Evans and Sutherland, appears to be in internal technical reports. A micropipelined dataflow processor is described in [76]. The work on data-driven processors is described in [207, 208, 373, 376, 410]. Other design work in the DSP and communications domain includes that in [6, 182, 269, 295].

The first fully asynchronous microprocessor designed at Caltech is described in [251, 257, 258]. The subsequent design of a GaAs version of this microprocessor is described in [379], and the asynchronous MIPS R3000 is

described in [259]. The Amulet microprocessors are described in [134, 135, 138, 406]. Numerous other researchers have also designed asynchronous microprocessors and special-purpose processors [10, 13, 76, 191, 288, 291, 324]. SUN's counterflow pipeline architecture is described in [364]. Experimental work from this group appears in [275, 372]. The low-power asynchronous designs from Phillips Research Laboratories are described in [38, 136, 192, 193]. The RAPPID project is described in [330, 367]. This design is also described in some detail in four patents [141, 142, 143, 144].

There have been numerous recent experiments with using asynchronous design to build functional units either to take advantage of delay variations due to data dependencies or operating conditions or to allow iterative algorithms to be carried out using a local clock. These designs include adders [80, 148, 199, 245, 256, 319, 331, 419], multipliers [74, 196, 293, 336, 419], and dividers [227, 321, 404]. Some of the most successful work has been applying self-timing to iterative division in that such a divider has been used in a SPARC microprocessor [400]. There have been several asynchronous memories designed as well [79, 127, 206, 356].

There has been some interesting work in applying novel circuit structures to asynchronous design. This includes using pass-transistor gates [399], bipolar circuit elements [403], dynamic logic [260], self-reseting gates [178, 231], threshold logic [226, 304], phased logic [236], and superconducting devices such as rapid single flux quantum (RSFQ) devices [108, 109, 110, 111, 216, 240]. An interesting technique for improving power consumption was proposed by Nielsen et al. in which the power supply is dynamically adjusted to balance power consumption and speed requirements [294].

In the domain of performance analysis, Burns uses a single delay parameter and presents techniques for determining the cycle period [65]. This performance estimate is used to guide transistor sizing. Williams developed techniques for analyzing throughput and latency in micropipelines and rings [402]. Ebergen and Berks analyzed the response time of linear pipelines [118]. Xie and Beerel present a performance analysis approach based on probabilistic delay models [408]. The stochastic cycle period analysis in Section 9.3 is from Mercer and Myers [270].

The self-checking property of asynchronous circuits has been investigated by numerous researchers [27, 162, 393]. These researchers proved that speed-independent circuits always halt under the output stuck-at fault model. Liebelt and Burgess extended these results to *excitory* output stuck-at faults (i.e., faults that take the circuit out of the valid set of states) [232]. Hazewindus demonstrated the potential for premature firing under the more general stuck-at fault model [162]. An early paper on test pattern generation is due to Putzolu [318]. Techniques for testing macromodule designs are presented by Khoche [195]. A partial-scan technique to test for delay faults is given by Kishinevsky et al. [201]. Design techniques to produce asynchronous circuits that are robust path delay fault testable are given in [194, 299]. Roncken has developed techniques to enable I_{DDQ} testing of asynchronous circuits. An ex-

cellent survey of testing issues as they relate to asynchronous design is given in [176] and forms the basis for Section 9.4.

The synchronization problem has been known for some time. The earliest paper we found on asynchronous design addresses it [239]. In Lubkin's 1952 paper, he makes the following comment about the synchronization problem:

> If an entity similar to one of Maxwell's famous demons were available to convert such "maybe's" to either "yes" or "no," even in arbitrary fashion, the problem would be solved.

He goes on to discuss how the designers of the ENIAC used additional flip-flops to allow more time to synchronize asynchronous inputs. This paper states that this technique does not eliminate the chance of error, but rather, it simply reduces its probability. This paper also presents a method of determining the probability of error. The problem appears to be largely ignored until 1966 when Catt rediscovered it and presents a different formulation of this error probability [70]. Again, it appears that the synchronization problem was not widely known or understood. Evidence of this is that several asynchronous arbiters designed in the early 1970s suffered from metastability problems if arrival times of signals are not carefully controlled [90, 91, 307, 315].

Finally, in 1973 experimental evidence of the synchronization problem presented by Chaney and Molnar appears to have awakened the community to the problem [75]. After this paper, a number of papers were published that provided experimental evidence of metastability due to asynchronous inputs, and mathematical models were developed to explain the experimental results [95, 125, 169, 200, 217, 308, 327, 394]. Pechoucek's paper also shows that the only way to reduce the probability to zero is to generate the clock locally and be able to stop the clock when metastability occurs.

One of the earliest proofs to show that metastability in a bistable is unavoidable is by Hurtado, who applies dynamical systems theory [177]. Marino demonstrates that a perfect inertial delay can be used to implement a perfect synchronizer, and he then goes on to show that several possible inertial delay designs are not perfect [246]. Thus, this casts further doubt on the existence of a perfect synchronizer. Marino later generalized and extended Hurtado's theory to arbitrary sequential systems [247]. He proved that if activity occurs asynchronously with respect to the activity of some system, metastabilibity cannot be avoided. Kleeman and Cantoni extended Marino's theory by removing the restriction that inputs have a bounded first derivative [205]. Barros and Johnson use an axiomatic method to show the equivalence of a bounded-time arbiter, synchronizer, and inertial delay [23]. Unger has recently shown a relationship between hazards and metastable states [385]. An excellent discussion of the synchronization problem, ways to reduce the probability of error, and the use of stoppable clocks is given by Stucki and Cox [368]. The mathematical treatment in Section 9.5 follows this paper.

Many designs have been proposed for synchronizers. Veendrick shows that the probability of metastability is independent of circuit noise in the synchronizer and that it could be reduced somewhat through careful design and

layout [394]. Kleeman and Cantoni show that using redundancy and masking does not eliminate the probability of synchronization failure [204]. Manner describes how *quantum synchronizers* can solve the problem in principle, but not in practice [244]. Sakurai shows how careful sizing can reduce the probability of failure in synchronizers and arbiters [335]. Walker and Cantoni recently published a synchronizer design which uses a bank of parallel rather than serial registers operating using a clock period of nT [397]. Another interesting design is published in [78] in which EMI caused by the clock is exploited to build a better synchronizer. One of the most interesting synchronizers was due to Seizovic, in which he proposes to pipeline the synchronization process [348]. *Pipeline synchronization* essentially breaks up the synchronization into a series of asynchronous pipeline stages which each attempt to synchronize their request signal to the clock. When metastability occurs in one stage, its request to the next stage is delayed. When the next stage sees the request, it will see it at a somewhat different time, and it will hopefully not enter the metastability region. As the length of the pipeline is increased, the likelihood that metastability persists until the last stage is greatly reduced. This scheme is used in the Myranet local area network. This network at the time had about 80 million asynchronous events per second with $\tau = 230$ ps. If the synchronous clock rate being synchronized to is at 80 MHz, the MTBF is on the order of 2 hours. Using an eight-stage pipeline for synchronization and a two-phase clock, a latency of 50 ns is incurred in which 28 ns is available for synchronization reducing the MTBF to about 10^{37} years.

Stoppable clocks date back to the 1960s with work done by Chuck Seitz which was used in early display systems and other products of the Evans and Sutherland company [347, 362]. Numerous researchers have developed GALS architectures based on the idea of a stoppable clock [77, 163, 328, 347, 368, 392, 423]. Some of the schemes such as those proposed in [233, 328, 368, 423] allow an asynchronous module to request to communicate data to a synchronous module at arbitrary times. The approach in [328] is based on an asynchronous synchronizer called a *Q-flop*. This synchronizer receives a potentially unstable input and a clock, and it produces a latched value and an acknowledgement when it has latched the value successfully. Q-flops are used in *Q-modules* which are essentially synchronous modules clocked using a locally generated stoppable clock. These Q-modules are then interconnected asynchronously. The schemes proposed in [77, 347] assume that the synchronous unit determines when data is to be transferred to/from the asynchronous modules. Recently, a group from IBM introduced a new GALS approach of the second kind called interlocked pipelined CMOS (IPCMOS) [341]. They implemented a test chip in a 0.18 μm 1.5 V CMOS process which consisted of the critical path from a pipelined floating-point multiplier. Their experimental results showed a typical performance of 3.3 GHz with a best-case performance of 4.5 GHz. An analysis comparing synchronous versus GALS is given in [2]. The globally synchronous locally asynchronous architecture is due to Sjogren [357].

Numerous early arbiter designs were prone to failure due to metastability. A safe arbiter is proposed by Seitz in [346]. The first published MOS design of an ME was also due to Seitz [346]. Arbiters with improved response times are presented in [183, 409]. Modular arbiter design are presented in [69, 387], which can be extended to an arbitrary number of requesters. A silicon compiler that produced circuits with arbiters is presented in [22]. In [35] it is shown that extra care must be taken when designing a three-way arbiter as a straightforward design leads to additional metastability problems.

Problems

9.1 Probability of Synchronization Failure

Assume that $\tau = 20$ ps, $T_0 = 8$ ns, the clock frequency is 4 GHz, asynchronous inputs arrive at an average rate of 2 GHz, and $t_a = 200$ ps.

9.1.1. What is the MTBF?

9.1.2. How many latches does it take to reduce MTBF to 10,000 years?

9.1.3. If no extra latchs are used, at what rate can asynchronous inputs arrive such that the MTBF is over 1 year?

9.1.4. For what value of τ does the MTBF exceed 1 year?

9.2 Probability of Synchronization Failure

Assume that $\tau = 10$ ps, $T_0 = 4$ ns, the clock frequency is 5 GHz, asynchronous inputs arrive at an average rate of 100 MHz, and $t_a = 20$ ps.

9.2.1. What is the MTBF?

9.2.2. How many latches does it take to reduce MTBF to 100,000 years?

9.2.3. If no extra latchs are used, at what clock frequency does the MTBF exceed one year? (Assume that t_a is 10 percent of the cycle time.)

9.2.4. What clock frequency would produce an MTBF of more than 1 year if t_a is 50 percent of the cycle time?