Synchronous Communication Techniques for Rationally Clocked Systems

by

Luis Francisco G. Sarmenta

B.S. Physics (1992)
B.S. Computer Engineering (1993)
Ateneo de Manila University

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author ....
Department of Electrical Engineering and Computer Science
May 25, 1995

[illegible]

Certified by ....
Stephen A. Ward
Professor of Computer Science and Engineering
Thesis Supervisor

[illegible]

Accepted by ....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Synchronous Communication Techniques for Rationally Clocked Systems

by

Luis Francisco G. Sarmenta

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

An increasingly common problem in designing high-performance computer systems today is that of achieving efficient communication between systems running at different clock speeds. This problem occurs, for example, in uniprocessor systems where the processor runs faster than the bus, and in heterogenous multiprocessor systems where processors of different kinds run at their own maximum speeds. Asynchronous solutions to this problem, which assume the systems' clocks to be completely independent, are typically inefficient because they must allow for sufficiently long synchronization delays to attain acceptable levels of reliability. Synchronous solutions, on the other hand, though more efficient and reliable, have traditionally lacked flexibility because they require the clock frequencies to be related by integer factors.

An efficient and flexible synchronous solution, called rational clocking, has been proposed that permits data to be transferred reliably and efficiently between two systems whose clock frequencies are related by the ratio of two small integers. This rational frequency constraint allows a wide variety of frequencies to be used, and at the same time assures a periodic relationship between the two clocks that can be used to determine a schedule for data transfers between the two systems.

In this thesis we present, improve, and test the rational clocking technique. We begin with the original table-based implementation, where the communication schedules are pre-computed and stored in lookup tables, and discuss some minor variations and improvements. Then, we improve the throughput of this technique with a double-buffering technique that guarantees 100% throughput efficiency for all frequency ratios and timing parameters. We also improve the space-efficiency and flexibility of the scheduling hardware dramatically by using a set of run-time scheduling algorithms that do not require lookup tables. Finally, we test all these ideas using a combination of software and hardware tools, including scheduling software, Verilog simulations, and a custom-designed CMOS VLSI chip.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Computer Science and Engineering

Acknowledgments*

This thesis was made possible by the support of many people. To them, I would like to express my deepest gratitude.

Thanks to the people I work with: my advisor, Prof. Steve Ward, for giving me a great research topic, and for his kind support and help throughout the writing of my thesis; Prof. Gill Pratt, whose remarkable insights helped lead my research in the right directions; and all the members of the NuMesh group and the 6.004 staff, for their help and friendship.

Thanks also to the people at DOST and Ateneo who made it possible for me to go to MIT, especially Fr. Ben Nebres and Mr. Arnie del Rosario.

Thanks to all my friends, who have in different ways made the stress of MIT life a little more bearable. In particular, thanks to my friends in FSA and in the TCC choir, for making me feel at-home halfway around the world from home, to my roommate Marc Ibañez for helping me relax after a long day's work, and to Cara Galang, for her prayers, and for giving me occasion to go out of MIT once in a while. Special thanks to Victor Luchangco, who not only gave me valuable feedback on my research and thesis, but also helped me with all sorts of other problems, big and small. Finally, thanks to Chê Manua, whose faith and dedication never cease to amaze and inspire me, and who, even though thousands of miles away, is always there to cheer me up and make me smile.

Of course, I would not even be here without my family, whose love, guidance, and encouragement have made all this possible. I would especially like to thank my parents, who have taught me the important things in life. I appreciate them so much more now that I am away from home.

Above all, I would like to thank God, who has been with me and sustained me through all the ups and downs of life. He has made my dreams come true, but at the same time has taught me that *His* plans for me are more wonderful than anything I can ever dream of.

*Financial support and equipment for the research in this thesis were provided in part by ARPA and MIT. My studies at MIT were supported in part by an Engineering and Science Education Project scholarship from the Philippine Department of Science and Technology, and by a Faculty Development Grant from Ateneo de Manila University.

Contents

List of Figures

Chapter 1

Introduction

A common problem in designing computer systems today is that of enabling systems running at different speeds to exchange data efficiently. This problem has traditionally been solved by having each system treat the other as completely asynchronous, and applying appropriate synchronization techniques. Such an approach, however, is needlessly inefficient. As long as the frequencies and the phase relationship between the two system clocks are both known, it becomes possible to *predict* their interactions and *schedule* communications efficiently and reliably in a synchronous manner.

This thesis presents and further develops a systematic technique called *rational clocking*, originally developed by Pratt and Ward [1, 2], that enables synchronous communication to be used between two systems whose clock frequencies are related by some known rational number. The rational frequency constraint assures the periodic relationship between the clocks necessary to provide reliable communication, while still allowing the clock frequencies to be chosen nearly independently of each other.

1.1 Applications

Rational clocking has many potential applications today, as more and more emphasis is placed on high performance and flexibility.

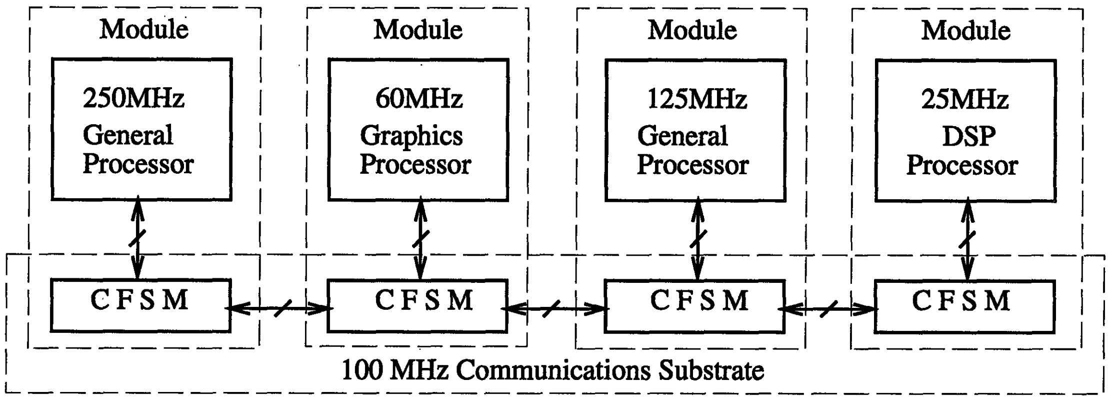


Figure 1-1: A simple NuMesh system with processors of different speeds.

1.1.1 Multiprocessor Systems

Rational clocking would be most useful in multiprocessor systems where processors might perform different functions or have different levels of performance that make it impractical or impossible to require them to run at a single speed.

A good example of such a system is MIT's NuMesh[3]. This project seeks to define a standardized communication substrate for heterogeneous parallel computers that would support highly efficient communication between modules of arbitrary and varying complexity. As shown in Figure 1-1, a NuMesh typically contains several different types of processing elements. Each processing element is connected to a communication finite-state machine (CFSM), which in turn is connected to other CFSMs in a near-neighbor mesh configuration. Together, these CFSMs form the communication substrate through which all inter-processor communication is done.

To achieve very high communication rates, the CFSMs are operated synchronously at a high common frequency. The processing elements, however, are allowed to run at their respective maximum frequencies in order to maximize the use of processors that can run faster than the communication substrate while accommodating slower processors that cannot run at the CFSM's frequency. The problem arises, therefore, of developing a mechanism that allows these processors of varying speeds to communicate efficiently with the CFSMs.

Rational clocking provides a nearly ideal solution to this problem. The processor clock can be generated from the CFSM clock such that its frequency is some rational multiple of the latter, close to the processor's maximum frequency. Then, a *communication scheduler* circuit, possibly embedded in the CFSM itself, would generate control signals that prohibit data from being transferred when timing violations would occur. In this way, flow control and synchronization of data between the processor and the CFSM can be achieved without resorting to inefficient and probabilistic asynchronous techniques such as handshaking and synchronizers.

1.1.2 Uniprocessor Systems

Rational clocking also has applications in uniprocessor systems. While improvements in technology are making it possible to develop processors of increasingly high maximum operating speeds, physical constraints are making it impossible for motherboards to catch up. Because of this, it is no longer reasonable to require the use of a single system clock frequency as has been done in the past.

Recent processors have addressed this problem by employing *clock multiplication*, a technique where the processor clock would be generated as an integer multiple of the motherboard clock. An example is the PowerPC 603, which can be clocked internally at one, two, three, or four times the system bus frequency [4].

Clock multiplication gives users the flexibility to scale systems to their own needs while giving system designers the freedom the make processors as fast as possible without having to worry as much about designing motherboards of the necessary speed. Rational clocking provides even more flexibility to the clock multiplication technique by allowing the use of non-integer factors.

1.2 Thesis Overview

This thesis is divided into seven chapters that discuss the concepts and techniques behind rational clocking. This chapter has shown the motivations for rational clocking. Chapter 2 provides background information about the problems being addressed, and also discusses previously proposed solutions to the problem and their shortcomings. Chapter 3 presents the basic rational clocking technique and some straightforward improvements on it. Chapter 4 presents the double-buffering technique which both increases throughput, and simplifies the communication scheduling algorithms. Chapter 5 discusses algorithms that allow communication to be scheduled by hardware at run-time, leading to a flexible and space-efficient implementation that makes it easy to integrate rational clocking into many applications. Chapter 6 describes the software and hardware tools used in for studying and verifying all these techniques. Finally, Chapter 7 summarizes the results of the previous chapters, and discusses possible directions for future research.

Chapter 2

Background and Previous Work

2.1 Synchronous Systems

Digital systems can be classified into two main types according to the timing of their signals: synchronous and asynchronous. In a *synchronous* system, all important events, such as the movement of data and the changing of states, happen at known times relative to a single periodic *clock* signal. This fixed relationship between events and the clock makes synchronous systems much easier to design than *asynchronous* systems, which do not have a coordinating signal like the clock. For this reason, a large majority of computer systems today are based on synchronous systems.

2.1.1 The Communication Model

Figure 2-1 shows a model of how communication between two subsystems in a synchronous system is typically done. In this model, the subsystems transfer data through modules containing an edge-triggered register (R) and combinational logic (CL). Both subsystems' clocks, Clk_m and Clk_n, are connected to a common clock, Clk, which is used to clock other subsystems in the entire system as well.

To transmit data to subsystem n, subsystem m places data at the transmit module's input, D_m. At the active edge of the clock, the register latches this data and subsequently

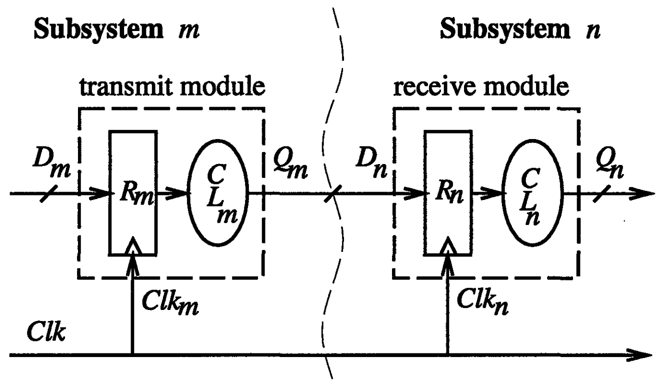


Figure 2-1: Communication in a synchronous system.

passes its value to the combinational logic. The combinational logic then performs desired computations on the data and places the result at the receive module's input. Finally, at the next active clock edge, this data gets latched by the receive module and gets passed to other parts of subsystem n.

Figure 2-2 shows an example of this data transfer process, where we assume the combinational logic blocks simply pass data without alteration. As shown, data placed at the D_m input is latched at clock edge 0 and appears at the Q_m output some time afterwards. The data is then latched by the receive module at clock edge 1 and appears subsequently at the Q_n output.

2.1.2 Timing Parameters

In practice, the registers and combinational logic blocks used in our model have certain timing properties that affect the way in which they can be used. A real module cannot instantaneously latch its input when a clock edge occurs. Switching delays within the register require us to hold the input signal stable for a certain minimum time in order for its value to be stored properly. Similarly, the output of a module cannot appear instantaneously after the clock edge either. Moreover, due to environmental factors, manufacturing variations, and data-dependent path differences through the circuitry, it is actually impossible to predict the exact time at which the module's output becomes stable and valid.

We account for the latching delay of a register by defining its *decision window*. This is

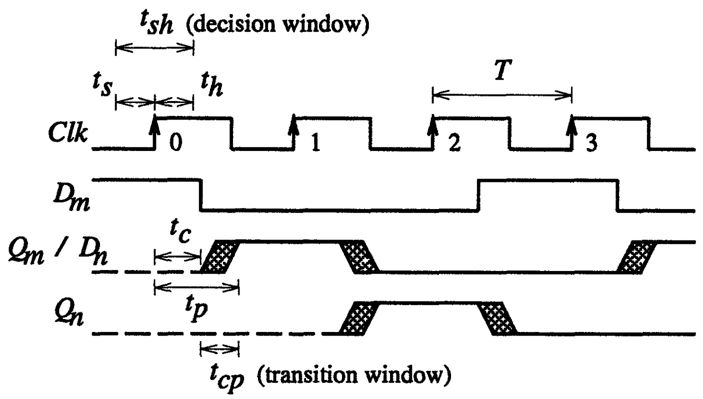


Figure 2-2: Synchronous data transfer and the timing parameters involved.

the time interval around the clock edge during which the input must be stable for correct operation of the register. It is bounded by two timing parameters: setup time (t_s) and hold time (t_h). The *setup time* is the minimum amount of time *before* the clock edge during which the input must be stable, while the *hold time* is the corresponding minimum amount of time *after* the clock edge. The point in time marked by the setup time can also be seen as the latest time that the input is allowed to be changing. The width of the decision window, t_{sh}, is equal to $t_s + t_h$. When the input changes after the setup time, we say we have a *setup time violation*. If it changes before the hold time, we say we have a *hold time violation*. These two violations are equivalent, so we shall refer to them in this paper more generally as *decision window violations*.

We account for the uncertainty in the output delay of a module by defining its *transition window*.[1] This is the time interval after the clock edge during which the output signal of a module can change. It is bounded by two timing parameters: contamination delay (t_c) and propagation delay (t_p). The contamination delay is the *earliest* time that the output signal can change, while the propagation delay is the *latest* time that the output can change. The propagation delay can also be seen as the earliest time that we can safely assume the output to contain its new value *and* be stable. The width of the transition window, t_{cp}, is equal to $t_p - t_c$.

[1]We must consider the *whole* module, not just the register, since the combinational logic is involved.

Figure 2-2 shows the effect of the timing parameters on the way data moves from one system to another. First, data to be latched at a certain clock edge must be stable within the decision window around that edge. This data then appears at the output some time within the transition window after that clock edge. For simplicity, we have assumed here that modules m and n have the same timing parameters.

2.1.3 Timing Constraints

In order for data to be transferred successfully in a synchronous system such as that in Figure 2-1, certain constraints must apply to the timing parameters and the clock period, T. First, the decision and transition windows must fit in one clock cycle. That is, for new data to be transferred every clock cycle, there must be enough time within a clock cycle for the transmitter's output signals to settle *and* for the receiver's register to latch these signals afterwards. To ensure this, the following inequalities must be true:

$$t_s < T, \qquad t_h < T, \qquad t_{sh} < T \tag{2.1}$$

$$t_c < T, \qquad t_p < T, \qquad t_{cp} < T \tag{2.2}$$

$$t_{sh} + t_{cp} \leq T \tag{2.3}$$

where t_{sh} is the decision window of the receiver, and t_{cp} is the transition window of the transmitter.[2] Second, the decision and transition windows must not overlap. If they do, then the output of the transmitter might change while the receiver is trying to latch it, resulting in a decision window violation. Thus, the following must be true as well:

$$t_p \leq T - t_s \tag{2.4}$$

$$t_c \geq t_h \tag{2.5}$$

The first of these two inequalities says that new data must be stable before the receiver starts latching its inputs. If not, the new data may not be latched properly. The second says that new data must not appear at the inputs of the receiver until data from the previous

[2]For convenience, we do not explicitly indicate the subsystem to which a timing parameter applies. It is important to remember, however, that setup and hold times are those of the receiver, while contamination and propagations delays are those of the transmitter.

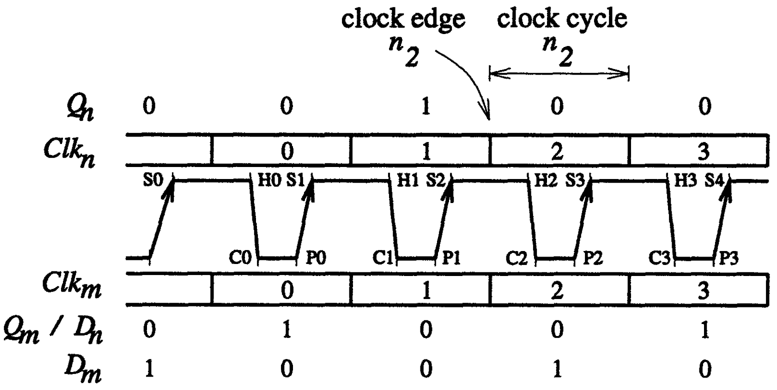


Figure 2-3: The schedule diagram of Figure 2-2.

cycle has been latched successfully. Otherwise, the old data may be lost or corrupted. When this happens, we say that the new data *contaminates* the old data.

In practice, it is often difficult or impossible for us to control the timing parameters of a system to meet these constraints. Instead, we usually control the clock period, T. By slowing down the clock, we can make T large enough to satisfy inequalities 2.1 to 2.4. Inequality 2.5 is then easily enforced, if necessary, by adding a delay element to the transmitter's output to increase t_c, and again extending the clock period to account for the accompanying increase in t_p. In this way, the timing parameters of a synchronous module determine a maximum clock frequency beyond which the module cannot properly operate.

2.1.4 The Schedule Diagram

To make it easier to describe data transfers between two systems[3] and detect violations of the inequalities above, we can use a *schedule diagram*, such as that in Figure 2-3, which depicts the data transfer operation in Figure 2-2.

In a schedule diagram, the clock cycles of each system are denoted by numbered boxes bounded by the active edges of the the system's clock. We associate each clock cycle with the active edge to its left, and refer to a specific cycle or edge of a system using the system's

[3]From here on, we shall use the term *system* to refer to subsystems as well as entire systems in general. This term becomes more appropriate when we begin to discuss systems with the different clocks.

name and cycle number. Clock cycle n_2 and its active edge are indicated in Figure 2-3 as examples. Although the two clocks, Clk_m and Clk_n, are the same in this example, they do not have to be so in general. In fact, as will be shown in Chapter 3, schedule diagrams are most useful when used with systems having different clocks.

For each clock cycle, the *final* stable values of signals of interest are written under or above the appropriate box. This makes it easy to see how data moves from one module to another without having to show their actual waveforms. During clock cycle m_0, for example, Q_m starts from 0 and changes to 1, so we write a 1 under clock cycle m_0.

The setup (S) and hold (H) times of the receiver are indicated on the schedule diagram by vertical lines surrounding each active edge of the receiver's clock. Similar lines denote the contamination (C) and propagation (P) delay points after each active edge of the transmitter's clock. We refer to these lines as the setup, hold, contamination, and propagation *edges* of the systems. Each of these edges is associated with some clock cycle, and is numbered accordingly as shown in Figure 3-4. The setup and hold edges define the decision window of the receiver (system n), while the contamination and propagation edges define the transition window of the transmitter (system m). To avoid communication errors, transfers must not be allowed to occur when these two windows overlap since this means that data at the receiver's input would not be stable while the receiver is sampling it.

We indicate the transmission of data during a certain clock cycle i (of the transmitter) in the schedule diagram by drawing an arrow from propagation edge P_i to the nearest setup edge S_j (of some cycle j), at[4] or after it. This allows us to identify the earliest clock edge j (of the receiver) at which the receiver can safely latch the transmitter's outputs for clock cycle i. The arrow from P_0 to S_1, for example, says that module m's output during clock cycle m_0 can be latched by module n at clock edge n_1. We do this for all clock cycles during which the transmitter presents new data at the transmit module's output. In Figure 2-3, arrows are drawn from all propagation edges since in this case, the transmitter always transmits new data.

For each cycle j during which the receiver latches data, we also draw a line from hold

[4]In this paper, we say that some edge P_i is *at* another edge S_j if P_i coincides in time with S_j.

edge H_j to the nearest contamination edge $C_{i'}$ at or after it. This line indicates the earliest clock edge i' (of the transmitter) at which the transmitter can transmit new data without contaminating the data being latched by the receiver at clock edge j. The line from H_1 to C_1, for example, says that module m can transmit new data during clock cycle m_1 without contaminating the data it transmitted during clock cycle m_0, which is latched by module n at clock edge n_1.

Once these lines are drawn, communication errors such as timing violations are easily detected. Violations of Inequalities 2.1 and 2.2 will cause the decision and transition windows respectively to overlap with themselves, while violations of Inequalities 2.3 to 2.5, will cause the decision and transition windows to overlap each other, and the arrows and lines to cross each other.

Schedule diagrams are useful not only in detecting timing violations, but in *avoiding* them as well. By carefully choosing to inhibit data transmissions or receptions on certain cycles such that the arrows and lines from the remaining cycles will not cross each other, we can avoid timing violations that would otherwise occur if data were transmitted on *every* cycle. This idea of *scheduling* communication between the two systems is the the basis of the rational clocking technique to be presented in succeeding chapters.

2.2 Using Different Clocks

The use of a single clock to coordinate data transfers in a synchronous system has many advantages. As we have seen, it lets us characterize the timing of signals with only a few parameters. It also allows us to avoid timing violations by simply making the clock period long enough. This ease with which we can address timing issues in single-clock systems enables us to concentrate on other aspects of system design, thus allowing us to develop complex systems that would be practically impossible to build as asynchronous systems.

Using a single clock, however, also has its disadvantages. If the system contains several modules, then the clock period must be long enough to accommodate the *slowest* module. This is clearly a problem since it is unreasonable to slow down critical processing modules,

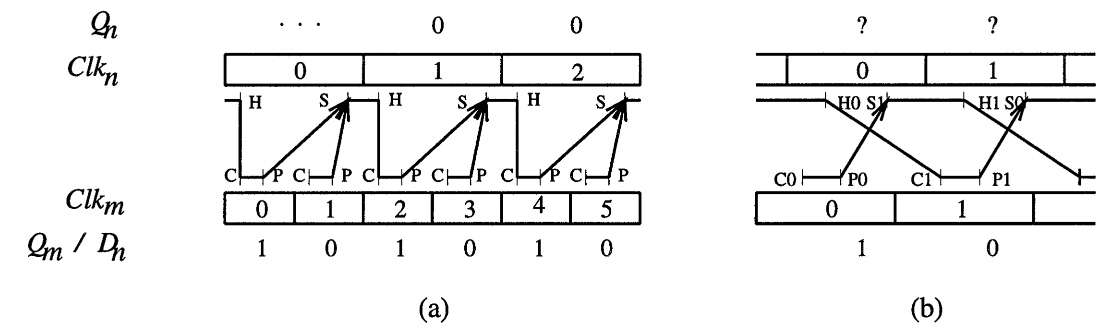


Figure 2-4: Problems with using different clocks. (a) Flow control. (b) Synchronization.

such as the CPU, just for the sake of relatively unimportant modules, such as terminals or printers. Also, it is often in fact impossible to slow down all the modules to the same speed. Some modules such as CPUs and DRAMs, have dynamic memory elements that cannot be clocked below a certain frequency without losing data. Other modules, such as monitors and modems, have industry-specified operating frequencies that must be followed.

For these reasons, many complex computer systems today are designed not as a single synchronous system driven by one clock, but as an ensemble of several synchronous subsystems driven by different clocks. When a computer system is divided in this way, new issues in how data is moved between systems arise and must be addressed.

2.2.1 Flow Control and Synchronization

There are two basic problems in establishing communication between two systems with different clocks: flow control, and synchronization. *Flow control* is concerned with the *frequency* difference between the two clocks. *Synchronization* is concerned with their *phase* difference. Examples of these two problems are shown in Figure 2-4.

Figure 2-4(a) shows an example of a flow control problem. Here, the transmitter is using a clock twice as fast as the receiver's clock. Although there are no decision window violations, data is lost because the receiver cannot latch data at the rate that the transmitter sends it. In this case, the 1's that system m sends on the even cycles are not received by system n because they are overwritten by the 0's sent on the odd cycles. Clearly, for data not to be lost in this way, system m must occasionally wait for system n before proceeding to send new data.

Figure 2-4(b) shows an example of a synchronization problem. In this example, both clocks have the same frequency, but system m's clock edges arrive earlier than system n's. As we can see, the difference in phase between the two clocks has caused the decision and transition windows to overlap, making it possible for the data being transmitted by system m to change while system n is latching it. The arrow from P_0 to S_1, for example, crosses the line from H_0 to C_1, indicating that the data system m is sending on cycle m_0 may contaminate previously sent data that system n is latching on clock edge n_0.

In general, if the two clocks are independent, then the transmitted data's transition window may occur anywhere within the receiver's clock cycle and may thus overlap with the decision window. Signals that occur at unknown times in this way are said to be ***asynchronous*** to the receiver. To ensure the proper latching of data, they must be ***synchronized*** to the receiver. That is, they must be forced to follow the timing constraints of the receiver, thus be converted into ***synchronous*** signals.

Although flow control and synchronization are fundamentally different problems, they are very closely related. Two clocks with different frequencies, for example, will have a time-varying phase difference, so synchronization must be addressed whenever flow control is necessary. Also, flow control may be useful for solving synchronization problems even if there is no frequency difference between the two systems. One way to solve the problem in Figure 2-4(b), for example, is to have system m transmit only on even cycles and have system n receive only on odd cycles. For these reasons, flow control and synchronization issues are usually addressed together.

2.2.2 Handshake Protocols

Flow control and synchronization problems arise between two systems using different clocks because each system does not know when data is latched or sent by the other system. An obvious solution to these problems, then, is to have each system tell the other when these events happen. This can be done with a ***handshake protocol.***

A typical handshake protocol works as follows:

1. The transmitter places new data on the line going to the receiver.
2. After the data is stable, the transmitter asserts a signal, *Q New*, to tell the receiver that new data is available for latching. The *Q New* signal may be sent in the same clock cycle as the data if it is guaranteed not to go high before the data is stable.
3. At each clock edge, the receiver checks if *Q New* is asserted and latches the data if it is.
4. After latching the data, the receiver asserts a signal, *D Taken*, to tell the transmitter that the data has been latched and that new data may now be sent. The *D Taken* signal may be sent in the same clock cycle that the data is being latched if it is guaranteed not to go high before the latching process is over.
5. After seeing that the *D Taken* signal has been asserted, the transmitter then lowers *Q New*, and waits for the receiver to lower *D Taken*.
6. When the receiver sees that *Q New* has been lowered, it lowers *D Taken* and prepares to latch new data when *Q New* is asserted again.
7. When it sees the *D Taken* signal lowered, the transmitter can then send new data by repeating the process from step 1.

This particular protocol is called a ***4-phase handshake*** and is used in many computer systems.[5] Another popular protocol is the ***2-phase handshake***, which uses transitions in *Q New* and *D Taken*, rather than actual values, for signaling. This protocol is faster since the systems do not have to wait for each other to lower their signals, but it is more difficult to implement and less reliable since each system has to remember the previous values of the handshake signals in order to interpret the transitions correctly.

Synchronous handshake protocols also exist for systems using the same clock. These are used when it might take more than one clock cycle for data to be produced or read. If

[5]In the literature, the names *REQ* (request) and *ACK* (acknowledge) respectively are more commonly used instead of *Q New* and *D Taken*.

a numeric coprocessor takes more than one cycle to produce the result of a floating-point operation, for example, then handshaking can be used to make sure that the CPU only reads the coprocessor output *after* the result comes out. Synchronous buses, such as the NuBus [5], often use handshaking in this way. In a synchronous handshake protocol, a system knows that its handshake signal will be received at the next clock edge and can automatically lower the signal after one clock cycle. The synchronous handshake is thus as fast as the 2-phase handshake, since it skips steps 6 and 7. At the same time, it is easier to construct and more reliable, since it only needs to interpret voltage levels and not transitions.

Whether synchronous or asynchronous, handshake protocols ensure proper flow control between the two interacting systems. Since each system must wait for a signal from the other before it latches or sends data, data is not lost or duplicated even if one system produces or reads data faster than the other.

For the most part, handshake protocols also take care of synchronization. Since the transmitter only raises *Q New* after the data is stable, and the receiver only raises *D Taken* after it finishes latching the data, the data is guaranteed not to change while the receiver is latching it. This reduces the synchronization problem to that of synchronizing *Q New* and *D Taken* — a much easier problem both because we only need to do synchronization for one signal at a time, and because we can take advantage of special properties of the handshake signals.

2.2.3 Synchronizers and Metastability

Synchronization of the handshake signals is traditionally done with a *synchronizer*. This is usually a simple flip-flop clocked by the system receiving the signal, as shown in Figure 2-5(a). The asynchronous input, D_s is fed into the input of the flip-flop. The value fed in then comes out of the flip-flop in synchrony with the clock and thus becomes a synchronous signal, Q_s, which can be used safely by the receiving system. This is shown in Figure 2-5(b).

Such a synchronizer works because a flip-flop's feedback loop is only stable at the valid high and low voltage levels, and not at invalid voltage levels. Thus, although the flip-flop

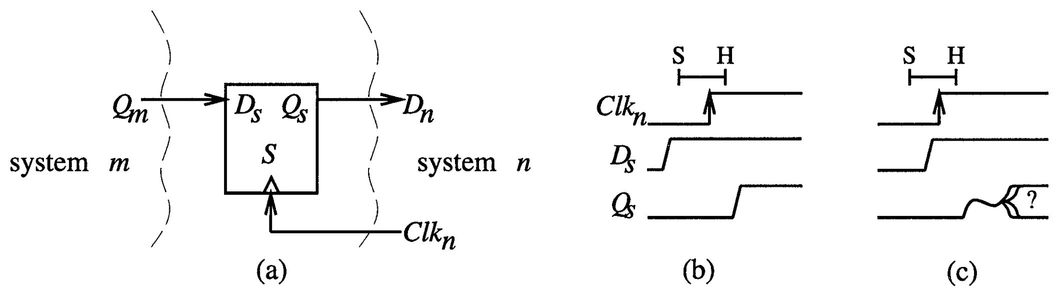


Figure 2-5: A synchronizer. (a) Implementation. (b) Normal operation. (c) Metastability.

might produce an invalid output voltage level if D_s changes within its decision window, its feedback loop will eventually pull the output to some valid voltage level. The particular value that the output falls to when this happens is not important because the handshake signals we are synchronizing do not change until their receipt is acknowledged by the system receiving them. If a handshake signal is misread the first time, it will simply remain at the same level and be read correctly on the following clock edge instead.

Unfortunately, the synchronizer has one fundamental problem: there is no upper limit to the time it takes for Q_s to become valid. This is because the output can end up perfectly balanced between the high and low voltage levels, such that pull of the feedback loop is equal in both directions. Once in such a *metastable* state, the output can remain there for an arbitrarily long amount of time until it is finally pushed away to either side by noise. When this happens, the output waveform may look like that shown in Figure 2-5(c).

We can visualize this problem with the ball and hill system shown in Figure 2-6, where we have a ball and a hill inside a well. Like a flip-flop, this system has two stable states, one on each side of the hill. If a ball is placed on any point on the hill, it will fall to one of the two stable points, and stay there. The exception is the point at the top of the hill. A ball placed there can remain balanced for an arbitrarily long time until it is pushed away by wind, earthquakes, or other random factors.

Metastability implies that no matter how much we extend the clock period, there is still a finite probability that Q_s will not be valid when the receiving system latches it at the next clock edge. When that happens, we say that we have a *synchronization failure.* A synchronization failure can cause problems because an invalid signal might be interpreted

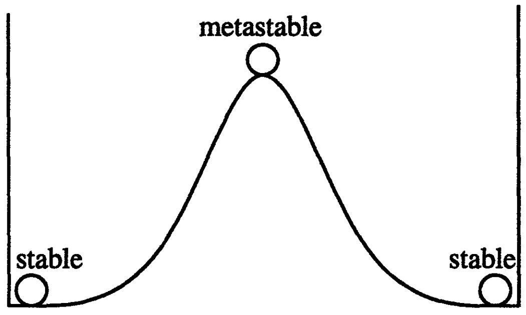


Figure 2-6: Ball and hill analogy for metastability.

differently by different parts of the receiving system. If *Q New* is invalid when the receiver checks it, for example, then it is possible for the part of the receiver that raises *D Taken* to interpret *Q New* as a 1, while the part that latches data interprets it as a 0. The receiver would then end up telling the transmitter that it has latched the data even if it has not really done so. Problems like this have caused disastrous failures in many computer systems in the past, and are still a present concern [6, 7].

There is no way to avoid landing in a metastable state other than to avoid causing decision window violations in the first place — which is impossible if the systems are asynchronous with each other. However, there *are* ways of reducing the probability of a synchronization failure so as to make it effectively negligible.

One way is to increase the gain of the flip-flop to make it tend to exit the metastable state faster, just like making the hill in Figure 2-6 steeper would make the ball on the top tend to fall more easily. However, this is not easy and cannot usually be done by a system designer. Typically, flip-flop manufacturers will specify the relevant characteristics of their flip-flops, and all a designer can do is pick a flip-flop that would satisfy the requirements of the system. In choosing a flip-flop, designers have to be very careful because a small increase in operating frequency can increase the synchronizer's failure rate by several orders of magnitude. In one case, it has been shown that the same flip-flop can have a mean-time-between-failures (MTBF) of 3.6×10^{11} seconds at 10 MHz, but have an MTBF of only 3.1 seconds at 16 MHz [7].

An easier way to reduce the probability of synchronization failure is to give the synchronizer more time to let metastable states settle to valid levels. It is known that the probability of synchronization failure is a decreasing exponential function of time [8]. Thus, given enough time, a metastable state can be resolved with high probability.

Extending a synchronizer's settling time can be done by either delaying observation of the flip-flop's output, or cascading an appropriate number of flip-flops. With enough delay or enough cascade stages, we can reduce the probability of synchronization failure to an acceptably low level — e.g., lower than the probability of flip-flop failure due to other causes. Thus, there is a trade-off between a synchronizer's reliability and its associated synchronization latency.

2.3 Previous Work

The traditional solution just described suffers from a number of problems. First, it is inefficient. The time needed for exchanging handshake signals and the time needed for letting metastable states settle result in significant overhead latency. Thus, if a continuous flow of data is to be transferred, having to handshake on every data transfer can significantly reduce throughput. Second, it is probabilistic. The probability of failure can never be completely reduced to zero. We can make the probability as small as we want, but we have to take great pains to do so, and usually at the expense of increased latency. Many improvements and alternative solutions have been proposed to get around these problems. Some of these are described in this section.

2.3.1 Pausable Clocks

One class of solutions involves the use of a *pausable clock* – a clock that can be stopped and restarted by an external signal. By strategically stopping the clock, these techniques prevent the latching of metastable signals.

Two examples of such techniques are Pěchouček's "fundamental solutions" [9]. The first uses a metastability detector connected to the synchronizer's output. When the synchronizer enters a metastable state, the detector stops the receiver's clock until the synchronizer settles into a stable state. This guarantees that the receiver does not read the synchronizer output while it is metastable. The second uses the handshake signals to stop and restart the clock in such a way that no decision window violations are possible. Since it avoids decision window violations altogether, this technique does not require the use of synchronizers and is not probabilistic. Similar schemes are presented in [10, 11].

Unfortunately, pausable clock systems suffer from several problems. First, they are difficult to implement, since they require a pausable clock generator. In some implementations, the complexity of a pausable clock system makes it even more inefficient than systems using traditional handshaking [12]. Second, they cannot be used in systems that require a clock with a constant frequency. Finally, since no other computations can be performed by a system waiting for its clock to be restarted, pausable clock systems can waste the potential capability of the faster system by effectively tying its clock to that of the slower while data transfer is being performed.

2.3.2 The Periodic Synchronizer

A novel solution proposed by Stewart and Ward [13], called the *periodic synchronizer*, notes that assuming complete asynchrony between the two system clocks is needlessly inefficient. As long they have constant frequencies, the two clocks will have a periodic phase relationship that would create repeating patterns in the positions of their interacting timing windows. The periodicity of these patterns allows us to *predict* decision window violations ahead of time and avoid them by disabling the latching of data when a violation is possible. This makes it possible to design a circuit that, given the two clock frequencies, produces a signal that identifies *in advance* the time intervals when a decision window violation can occur. This signal can then be used to inhibit a register from latching data during those times, thereby ensuring the integrity of data being passed through the register.

In such a circuit, the synchronization point is not eliminated but simply moved from the the register latching the data to the circuit producing the inhibiting signal. However, since the inhibiting signal is produced in advance, any metastable states that may result from the synchronization are given time to settle before they are needed. In fact, the periodicity of the patterns makes it possible to compute the inhibiting signal as far ahead in advance as desired, allowing us to reduce the probability of synchronization failure to an arbitrarily low level without incurring additional latency.

While conceptually promising, however, the periodic synchronizer unfortunately lacks flexibility. Since it uses time delay elements whose values must be computed for a given pair of frequencies, a particular periodic synchronizer circuit would be limited to work only for a specific pair of frequencies at one time, and selecting these frequencies would be a cumbersome process requiring the production of precise time delay elements with the required values.

An *adaptive* periodic synchronizer, which does not require prior knowledge of the two frequencies, has been proposed to solve this problem [14]. It attempts to determine the frequency and phase relationships of the clocks using digital and analog techniques. However, since it ultimately has to deal with a continuous range of possible frequencies and phases, it makes use of several approximations and assumptions. Because of this, the implementation is relatively complex and has subtle limitations that restrict its usefulness.

2.3.3 Synchronous Solutions

A limitation that has so far made flow control and synchronization difficult to solve is our lack of control over the phase difference between the two systems. This means that the receiver cannot know when signals from one system might change and must rely on a synchronizer. For this reason, even the periodic synchronizer, which takes advantage of knowledge about the frequencies of the two systems, uses a synchronizer at some point.

Often, this limitation is unnecessary. In many cases, we can satisfy the need for a variety of clock frequencies without using independent clocks by generating clocks of different frequencies *from a single source.* Clocks generated in this way will have known frequency and phase relationships, and events in one system will only occur at known times relative to any other system's clock. In short, the systems will be *synchronous* with each other even though they may be running at different frequencies. This synchronous relationship makes it possible to establish efficient and reliable communication between the systems as was done with the simple synchronous system of section 2.1.

Harmonically Clocked Systems

The simplest systems that make use of this idea are *harmonically clocked* systems — systems where the clock frequency of one system is an integer multiple, or *harmonic*, of the other. System m might be, for example, some integer M times as fast as system n.

In such systems, proper flow control is easily achieved by limiting the faster system (system m in this case) to transmit at most once every M cycles of its clock. This can be done by making a counter that outputs a pulse every M system m clock cycles, and using such a *divide-by-M counter* to trigger the sending of new data. Synchronization is then achieved by selecting one cycle out of the M that can be used in such way that no timing constraints would be violated. If none of the timing windows overlap, then flow control can also be achieved by traditional handshaking. This may not be efficient because of overhead, but it is at least non-probabilistic, and allows existing systems designed for handshaking to be used without modification.

The example Figure 2-4(a) shows a harmonically clocked system with $M = 2$. Here, a divide-by-2 counter (i.e., a toggle flip-flop) can be used to control the sending of data. Depending on the initial value of the counter, system m would transmit only on odd cycles or only on even cycles, but never both. In this case, the choice between odd and even does not matter.

Today, many high-performance systems employ harmonically clocked systems where the CPU chip runs at several times the speed of the motherboard. The ability to do this has become critical as it is has become impractical, or even impossible, to require motherboards to run at the CPU's desired speed.[15, 4]

Rationally Clocked Systems

Harmonically clocked systems form a small subset of the more general class of *rationally clocked* systems — systems whose clock frequencies are rational multiples of each other. We may have, for example, system m be M/N times as fast as system n, where M and N are integers.

In such systems, flow control is achieved by making sure that the faster system transmits data *at most* $M - N$ times out of every M cycles (assuming $M > N$). Synchronization then involves choosing which cycles to use and possibly inhibiting some of these as necessary to avoid decision window violations. Again, if none of the timing windows overlap, then flow control can be achieved reliably by handshaking. If some timing windows overlap, however, then appropriate circuitry must be used to inhibit data transfer on problematic cycles.

If the integers M and N are small, and the timing parameters are specified, designing such circuitry can be done easily on an *ad hoc* basis. Sometimes, it is also possible to find a specific group of ratios which are easily handled by simple circuitry. For example, if $N = M - 1$ and the timing windows are sufficiently small, a divide-by-M counter that *inhibits* transmissions instead of allowing them can be used for flow control between two systems with a frequency ratio of the general form $M/(M-1)$.

A number of recently-released processors already provide support for simple ratios such as 3/2, 4/3, and 5/2 [16, 17]. Among these is a processor from HP that supports frequency ratios of the form $M/(M-1)$ [17]. The techniques used by these processors are not well-documented in current literature, but it is likely that they employ the *ad hoc* methods just described, and cannot be generalized to work with arbitrary frequency ratios.

An interesting technique that attempts to describe a *general* approach that works for a wide variety of frequency ratios has been proposed by IBM [18]. This technique is based on the observation that the clock edges of two rationally clocked systems can only occur at a finite number of positions relative to each other, and that these positions are spaced in intervals of time α, where α is the greatest common factor of the two clock periods. This makes it possible to shift the clock waveforms away from potential decision window violations by delaying one of the clocks any amount necessary up to α seconds. Unfortunately, this technique only works for a restricted range of timing parameters (i.e., the widths of the timing windows must be small). Also, like the period synchronizer, it requires delay elements that must be changed for each frequency pair.

A simpler and more general technique, called *rational clocking* has been proposed by Pratt and Ward [1, 2]. This technique uses lookup tables that are consulted on every cycle to determine whether it is safe or unsafe to transmit data on the next cycle. These lookup tables can then be systematically programmed to handle *any* combination of frequency ratios and timing parameters. Rational clocking is presented in full detail in the next chapter, and developed further in the rest of this thesis.

Chapter 3

Rational Clocking

The problems with providing communication between two systems running at different speeds stem from the use of completely independent clocks and the resulting ignorance of each system about its frequency and phase relationship with the other system. Since a system does not know where in time the decision and transition windows of the other system are relative to its own, it has no choice but to treat the signals from the other system as purely asynchronous signals, and use inefficient and probabilistic asynchronous communication methods. One of the key ideas in rational clocking is that the clocks do not have to be independent. Well-known techniques can be used to generate the two clocks from a single source so as to maintain a known frequency and phase relationship between them. Knowledge of this relationship then makes it possible to avoid communication problems using efficient and reliable synchronous techniques.

In rational clocking, we use two clocks, Clk_m and Clk_n, with frequencies f_m and f_n respectively, such that

$$\frac{f_m}{f_n} = \frac{M}{N} \tag{3.1}$$

where M and N are integers, and such that their active edges are known to coincide at some point in time. The active edges of such clocks will coincide periodically with a *coincidence period*, T_{MN}, equal to:

$$T_{MN} = MT_M = NT_N \tag{3.2}$$

where T_M and T_N are the two clocks' periods, and will define *coincidence cycles* equal in length to an integer number of cycles on each system (i.e., M cycles on system m, and N cycles on system n).

When the two clocks are related in this way, the frequency ratio M/N allows us to predict the relative positions of the systems' clock edges within a coincidence cycle and generate a *schedule* of safe and unsafe cycles for data transfers. This makes it possible to ensure reliable communication by synchronously inhibiting data transfers during the predetermined unsafe cycles. Since the clock interactions are periodic, the same schedule can be applied every M clock cycles on system m and N clock cycles on system n.

This chapter presents the basic techniques involved in rational clocking, starting with techniques for generating clocks with the desired relationships. A table-based implementation of the rationally clocked communication hardware is then described, together with an algorithm for determining the communication schedule to be programmed into the lookup tables. Finally, the performance of the proposed circuit is evaluated and a number of minor improvements and variations are presented.

3.1 Clock Generation

In generating clocks for rational clocking, we have a simple objective: to generate two clocks that follow Equation 3.1 *and* have coinciding edges. This objective can be achieved by two well-known techniques: *clock division* and *clock multiplication*.

3.1.1 Clock Division

Clock division involves deriving the two system clocks, Clk_m and Clk_n, as subharmonics of a single clock of higher frequency, Clk_{high}. This is done with *frequency-dividing counters* — counters that receive an input clock and output a signal that pulses an integer number of times slower than the input clock. Such counters are easily constructed using binary counters with a presettable count limit and an output that emits a pulse whenever the count wraps-around. For example, we can use an incrementing counter with a presettable

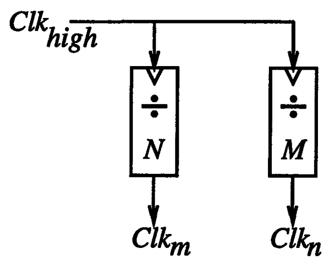


Figure 3-1: Generating rationally-related clocks through clock division.

terminal count and a zero output, Z, that goes high when the count value is 0 and stays low otherwise. If we set the terminal count to $N-1$, the Z output will emit a positive edge every N cycles as the count wraps-around from $N-1$ to 0, and we have a *divide-by-N* counter. Decrementing counters can also be used in a similar fashion. This is done in the demonstration circuitry described in Chapter 6.

Figure 3-1 shows how frequency-dividing counters can be used to generate rationally-related clocks. Here, we have

$$Nf_m = Mf_n = f_{high} \tag{3.3}$$

which is equivalent to Equation 3.1. The desired phase relationship is achieved by assigning appropriate initial values to the counters so as to force the counters to wrap-around at the same time. If incrementing counters are used, initializing both counters to their terminal counts, $M-1$ and $N-1$, would force both counters to wrap-around at the next active edge of Clk_{high}, and cause the Z outputs of the two counters to go from low to high at the same time, generating coinciding positive edges on Clk_m and Clk_n. Once initialized this way, the counters would continue to produce coinciding clock edges at the coincidence frequency.

3.1.2 Clock Multiplication

Clock multiplication involves generating one of the clocks from the other by multiplying the latter by the desired frequency ratio. This is achieved using an analog device called a *phase-locked loop*, or PLL. As shown in Figure 3-2(a), a PLL is composed of three components in a feedback loop: a phase detector (PD), a low-pass filter (LPF), and a voltage-controlled oscillator (VCO). The phase detector compares the frequency and phase of its two inputs,

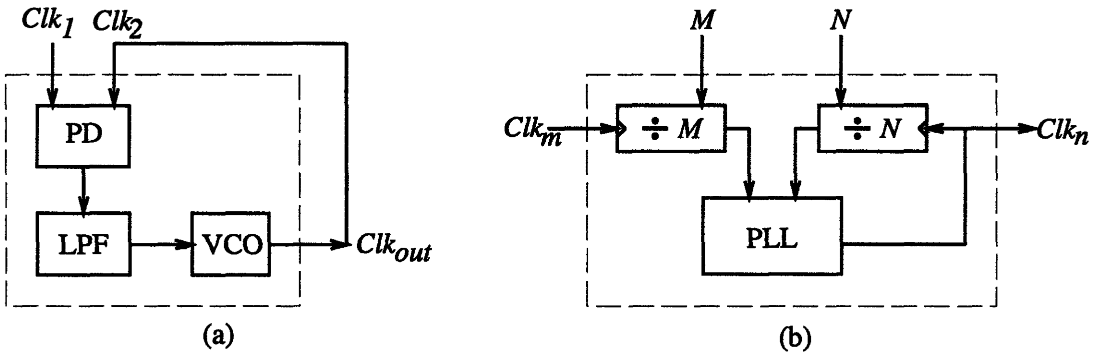


Figure 3-2: Clock multiplication. (a) A PLL. (b) A programmable clock multiplier.

Clk_1 and Clk_2, and outputs an appropriate error voltage which, after being filtered, adjusts the VCO output in such a way as to eventually force Clk_2 to become identical to Clk_1 in phase and frequency.

A PLL can be used for clock multiplication as shown in Figure 3-2(b). Here, the source clock, Clk_m, is frequency-divided by an integer M before being used as the PLL's Clk_1 input. The VCO output is similarly divided by another integer N before being fed into the Clk_2 input. Clk_n is then taken from the undivided VCO output. In this configuration, the PLL will try to control the VCO output to force its inputs, Clk_1 and Clk_2, to become identical in phase and frequency. Since $f_1 = f_m/M$ and $f_2 = f_n/N$, we have

$$\frac{f_m}{M} = \frac{f_n}{N} \tag{3.4}$$

which is again equivalent to Equation 3.1. Furthermore, since Clk_1 and Clk_2 are forced by the PLL to have coinciding edges, their multiples, Clk_m and Clk_n, have coinciding edges as well, and the coincidence period, T_{MN}, is given by Equation 3.2. Finally, by making M and N loadable from outside the circuit, we get a *programmable* clock multiplier.

3.1.3 Comparing Clock Division and Clock Multiplication

Clock division has the significant advantage of being completely implementable using only digital components. However, it requires that we generate a clock signal (Clk_{high}) that is several times faster than any of the clocks we will actually use in the system (Clk_m and Clk_n), and that we design frequency-dividing counters that can operate at this high frequency. In high-performance systems, where the frequencies of Clk_m and Clk_n would typically be very close to the maximum possible clock frequency for the available technology, this requirement is likely to be impossible to meet.

Clock multiplication avoids the problem of having to generate and use unnecessarily high frequencies but introduces a number of new problems due to the analog nature of the PLL. One problem is that the analog components of a PLL make designing an on-chip PLL difficult in digital VLSI circuits. Another problem is that the feedback loop takes time to stabilize. Thus, while the feedback loop is still unstable, the resulting Clk_n output would be invalid and should not be used. Still another problem is the limited range of frequencies that a particular PLL can accept and generate.

Fortunately, many techniques have already been developed to address these problems (for example [19, 20, 21, 22]), and clock-multiplying PLL's are fast becoming a standard component in microprocessors today [15, 16, 4, 17]. In this paper, therefore, we will just assume that adequate PLL technology is available for our clock generation needs.

3.2 Table-Based Communication Control Hardware

When the two clocks are generated using the techniques just described, it becomes possible to determine ahead of time whether it is safe or unsafe to transfer data during a certain clock cycle. Precomputing such information for each of the M system m and N system n cycles in a coincidence cycle leads to the lookup-table-based communication control hardware shown in Figure 3-3.

In this circuit, data to be transferred between the two systems is placed in a *transmit register* by the transmitter and latched into a *receive register* by the receiver. Enable controls on these registers allow us to inhibit transmission and reception of new data when necessary. For simplicity, we show these registers as distinct components of each system. In actual applications, these registers can be integrated into each system so that extra communication latency is not incurred. We may, for example, replace these registers with the modules of section 2.1.1 so that no opportunity to compute is lost even when data passes

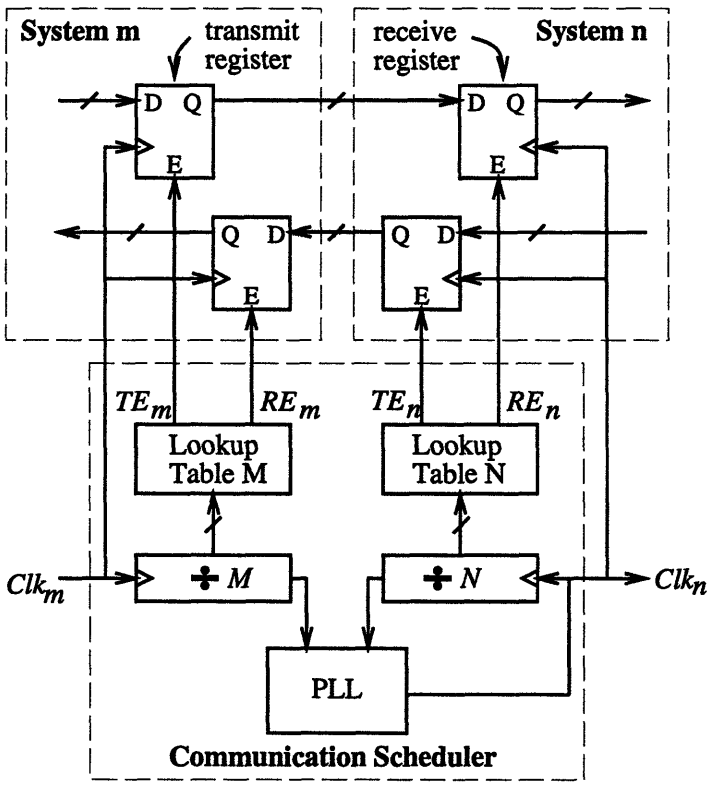


Figure 3-3: Interface hardware for rational clocking.

between systems.

As shown, Clk_n is generated from Clk_m through clock multiplication. In this configuration, the count values of the counters used for frequency generation indicate the current clock cycle (of their respective systems) within the coincidence cycle, and can be used to index two lookup tables containing the schedule of data transfers.[1] For each clock cycle number, each table generates two control signals: RE (Receive Enable) and TE (Transmit Enable). RE controls the receive register, and is asserted on the receiver's side when valid data can be received from the transmitter in the coming cycle. TE controls the transmit register, and is asserted on the transmitter's side when it would be safe for new data to appear at the output of the transmit register after the next clock edge. By programming the lookup tables appropriately, we ensure that the transmitter's output never violates the

[1]If clock division is used instead of clock multiplication, separate counters clocked by Clk_m and Clk_n must be used for indexing the tables.

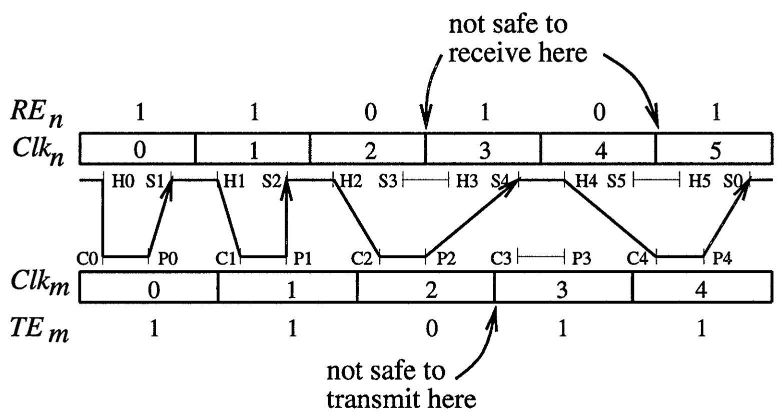


Figure 3-4: Transmitting from system m to system n where $M = 5$ and $N = 6$.

receiver's timing constraints, and effectively synchronize the transmitter's output to the receiver's clock.

We can make existing systems that rely on handshaking more efficient by using this circuit in place of synchronizers for exchanging handshake signals. Since timing violations are now impossible, metastable states cannot occur anymore, and waiting for them to settle becomes unnecessary. This can reduce handshaking latency significantly, especially in high-speed systems where the required settling times are typically several clock periods long. If it is not difficult to modify the systems' communication interfaces, we can improve communication efficiency even more dramatically by using this circuit for transferring actual data, and modifying each system to use its RE and TE signals instead of handshake signals to determine when to read and write data. This eliminates the need to exchange handshake signals between the systems every time data is to be transferred, and makes it possible to achieve very high levels of throughput for continuous data transfers.

3.3 Communication Scheduling

The values of the lookup table entries are computed using knowledge of the clock frequencies and the timing parameters of each system. These parameters are used to draw a schedule diagram as described in section 2.1.4. Figure 3-4 shows a diagram depicting system m transmitting data to system n, where $M = 5$ and $N = 6$.

The schedule diagram makes it easy to schedule data transfers between the two systems. The following graphical algorithm describes one way to use the schedule diagram:

1. Draw a schedule diagram indicating the relevant timing windows of the two systems.
2. Start with some propagation edge P_i (of some cycle m_i) and call this edge P_{begin}.
3. Move from P_i to the nearest setup edge S_j (of some cycle n_j) at or after it. Use modular arithmetic when "moving" from one edge to another. That is, if you go past the rightmost edge of the diagram, "wrap-around" to the leftmost edge. This reflects the periodic nature of the communication patterns.
4. Move from S_j to its corresponding hold edge H_j.
5. Move from H_j to the nearest contamination edge $C_{i'}$ at or after it.
6. Move from $C_{i'}$ to its corresponding propagation edge $P_{i'}$.
7. If $P_{i'}$ is *past* P_{begin} (i.e., you passed over P_{begin} in the process of moving from P_i to $C_{i'}$), then the scheduling is done.
8. If $P_{i'}$ is not past P_{begin}, then it is safe to transmit data from P_i to S_j. Draw an arrow from P_i to S_j (indicating that it is safe to transmit on cycle m_i and receive on cycle n_j) and a line from H_j to $C_{i'}$ (indicating it is safe to transmit on cycle $m_{i'}$ without contaminating the reception at clock edge n_j). Then, set lookup table entries $TE_m[(i-1) \bmod M]$ and $RE_n[(j-1) \bmod N]$ to 1 to enable the transfer.
9. If $P_{i'} = P_{begin}$, the scheduling is done. If not, make $P_{i'}$ the new P_i and go back to step 3.

To schedule data transfers in the reverse direction, repeat these steps with the roles of systems m and n reversed.

Figure 3-4 shows the result of using this algorithm with $P_{begin} = P_0$. Sampling is inhibited on clock edge n_3 since the new data being transmitted during cycle m_2 is not yet stable by setup edge S_3. Sampling can be done on clock edge n_4, but transmission is

Transmit	Receive Frequency															
Freq.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
4	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
5	100	100	100	100	100	80	100	100	100	100	100	100	100	100	100	100
6	100	100	100	100	80	100	83	100	100	100	100	100	100	100	100	100
7	100	100	100	100	100	83	100	85	85	100	100	100	100	100	100	100
8	100	100	100	100	100	100	85	100	87	100	100	100	100	100	100	100
9	100	100	100	100	100	100	85	87	100	88	88	100	100	100	100	100
10	100	100	100	100	100	100	100	100	88	100	80	80	90	100	100	100
11	100	100	100	100	100	100	100	100	88	80	100	81	81	100	100	100
12	100	100	100	100	100	100	100	100	100	80	81	100	83	83	100	100
13	100	100	100	100	100	100	100	100	100	90	81	83	100	84	84	92
14	100	100	100	100	100	100	100	100	100	100	90	83	84	100	85	85
15	100	100	100	100	100	100	100	100	100	100	90	100	84	85	100	80
16	100	100	100	100	100	100	100	100	100	100	100	100	84	85	80	100

Figure 3-5: Throughput efficiency (in %) for different frequency combinations.

inhibited on clock edge m_3 to prevent the transmit register's output from changing while system n is sampling data from cycle m_2. Finally, sampling is inhibited on clock edge n_5 since new data is not yet available by setup edge S_5. The resulting schedule thus allows four data transfers to be completed per coincidence cycle.

The choice of P_{begin} can affect the throughput efficiency of the algorithm's result. A choice of $P_{begin} = P_3$ in Figure 3-4, for example, would result in only three data transfers per coincidence cycle. In generating schedules, therefore, we must execute the algorithm for *all* possible choices of P_{begin} and pick the best resulting schedule. Schedules chosen in this way would then be optimally efficient because the algorithm we use is *greedy*. That is, it always transfers data between the systems at the soonest possible time. Any other algorithm would deviate from the greedy algorithm by postponing some data transfers, and can only produce schedules which allow an equal or lesser number of data transfers per coincidence cycle than the greedy algorithm.

3.4 Performance

The throughput efficiency of rationally clocked communications can be measured as the ratio between the number of transfers performed in a coincidence cycle and the minimum of M and N. In Figure 3-4, for example, the throughput efficiency was 4/5, or 80%. Figure

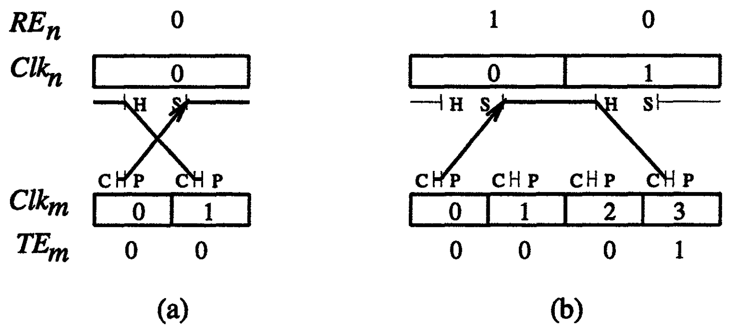


Figure 3-6: Equivalent ratios may not be equally efficient. (a) $M/N = 2/1$. (b) $M/N = 4/2$.

3-5 shows the throughput for different frequency pairs given the following typical timing constraints: setup time (S) = 20%, hold time (H) = 10%, propagation delay (P) = 20%, contamination delay (C) = 15%.

Although not apparent from Figure 3-5, equivalent frequency ratios do not necessarily result in the same throughput efficiency. Figure 3-6 shows the resulting schedules for the equivalent ratios 2/1 and 4/2 with the following timing constraints: $S = 40\%$, $H = 20\%$, $C = 30\%$, $P = 40\%$. As shown, although communication from system M to system N is impossible when $M = 2$ and $N = 1$, it becomes possible when we make $M = 4$ and $N = 2$ instead. In general, using a different but equivalent frequency ratio may enable more data transfers to be achieved by providing extra opportunities for performing transfers. Interestingly, however, making the numerator and denominator of the ratio larger is not always appropriate. Figure 4-1 in the next chapter, for example, shows that using the ratio 6/3 would result in lower efficiency than using 4/2. To achieve maximum throughput, therefore, we should compute the throughput efficiency of all equivalent ratios within the range of the clock generating circuitry and use the ratio that results in the highest throughput efficiency.

3.5 Variations and Improvements

3.5.1 Generic Lookup Tables

When programming the lookup tables, it is useful to specify the timing parameters in fractions of a clock cycle rather than in absolute time units. This makes it possible to create

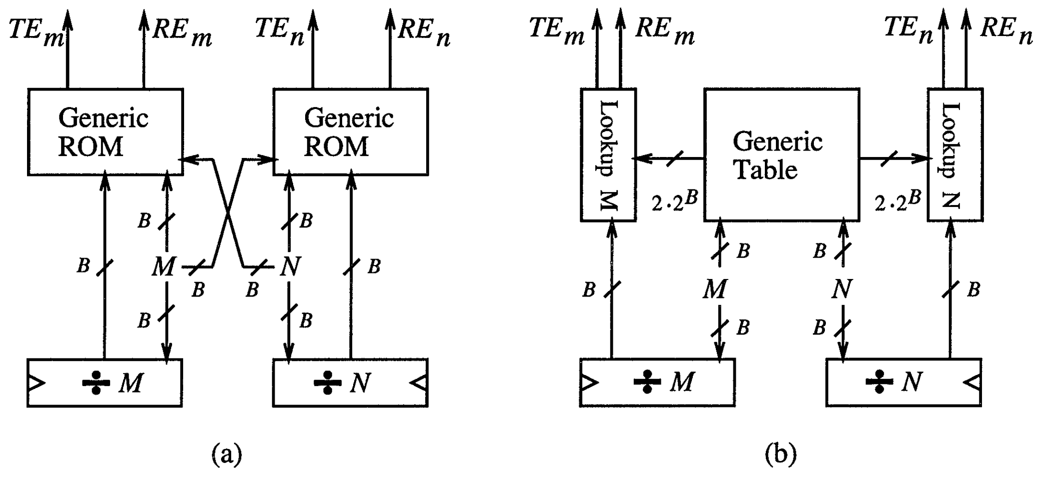


Figure 3-7: Generic lookup tables. (a) Using two ROMs. (b) Using a single table.

lookup tables that are useful regardless of the actual frequencies being used, presuming that the timing parameters scale with the minimum clock period. A single *generic* lookup table might take M and N as inputs and produce the signals needed for the frequency ratio M/N. This generic table can be used together with presettable frequency-dividing counters to construct a single "off-the-shelf" rational clocking circuit that supports different frequency ratios as needed in applications.

Figure 3-7 shows two possible implementations of this idea. In Figure 3-7(a), we use two "generic" ROMs that take M, N, and the current cycle count as address inputs, and output the corresponding control signals. If M and N are B-bit integers, then each ROM would take $3B$ bits of address input and output 2 bits for TE and RE, and would thus have 2×2^{3B} bits of memory. If we assume that the timing parameters specifications of the two systems are *proportionally symmetric* — that is, that the fraction of a system clock cycle covered by each timing parameter is the same for the two systems (e.g., the setup times of the two systems both represent 20% of their respective systems' clock periods) — then the two ROMs would be exactly identical, and simply crossing the roles of M and N as shown in the figure allows us to generate the control signals appropriate for each system.

Figure 3-7(b) shows an alternative circuit which uses a single generic table that takes M and N as inputs and outputs the bits contained in the appropriate lookup tables. For B-bit M and N values, this table would take $2B$ address input bits and output 4×2^B

bits, and would have $(4 \times 2^B) \times 2^{2B} = 4 \times 2^{3B}$ bits of memory. The output bits are fed into additional lookup circuitry that takes the count value and produces the desired TE and RE signals for each system. This configuration may not be practical for board-level implementations (because of the generic table's large number of outputs), but it becomes useful for integrated implementations where it is possible to combine the generic table and the lookup circuitry on one chip. By collecting all the scheduling information in a single place, this configuration allows us to reduce redundancy in the generic tables in ways not possible with the configuration in Figure 3-7(a).

3.5.2 Reducing the Generic Table Size

One way to reduce redundancy in the generic table is to note that if the timing parameters are proportionally symmetric, then the table's output bits for reciprocal frequency ratios (e.g., $M/N = 5/6$ and $M/N = 6/5$) would be exactly the same and would only differ in which system gets which set of bits. Thus, it is possible to design the table to only accept one of each pair of reciprocal ratios, and then use an extra bit to direct each set of bits to the appropriate system. The generic table may, for example, be designed to produce the appropriate bit patterns only for cases where M is less than N. Such a structure, when implemented as a PLA instead of a ROM, would only be half (or even less) the size of the original generic table since not all possible input addresses are used. To handle cases where M is greater than N, an extra bit can be used to control selection circuitry that would interchange the output bits as necessary.

The size of such selection circuitry grows linearly with the length of the lookup table, while the amount of space it saves grows as the cube of the table's length. That is, for numerators and denominators ranging from 1 to N, the size of the selection circuitry grows as $O(N)$, while the amount of space saved grows as $O(N^3)$. Thus, except when M and N are limited to be very small, the use of the configuration in Figure 3-7(b) is more space-efficient than the use of the two ROMs in Figure 3-7(a).

Another significant reduction in size can be achieved by noting that it is only necessary to have one set of entries in the generic tables for all equivalent forms of a particular ratio.

Thus, we can choose the form which results in the highest throughput, and then design the PLA to only accept this specified form. In the case where M and N range from 1 to 16, for example, there are 159 distinct ratios out of 256. Thus, we only need a table 62% as large as the original one. If the timing parameters are proportionally symmetric, we can further cut the table size in half using the first technique, and get a table only 31% as large as the original one.

3.5.3 Programmable Lookup Tables

Another way to support different frequency ratios is to use *programmable* lookup tables. We might, for example, have RAM-based tables on-chip, and then have these loaded *serially* at bootup time from an off-chip generic table. Such a setup requires very few pins and would be useful in space-constrained applications. Furthermore, the small size of the RAM-based tables compared to that of the generic table makes them capable of operating at higher frequencies. Programmable tables also make it possible to forego the ROM altogether and instead *compute* the lookup table contents at bootup using software or dedicated circuitry. Chapter 6 describes a prototype rational clocking chip that implements this RAM-based approach, and Chapter 5 proposes circuitry that can be used to compute the lookup tables entries at bootup.

3.5.4 Reaction Latency Compensation

As described, the communication scheduler assumes that new data is made available at the input of the transmit register in the same cycle that TE is enabled in order for the data to be transmitted at the next clock edge. In practice, however, most systems require at least one cycle to react to TE, and would thus need a *time-advanced* copy of TE. Fortunately, the periodicity of TE allows us to generate this copy by simply *rotating* the lookup table entries for TE by an appropriate amount. The same idea can also be applied to the RE signal if necessary.

Figure 3-8 shows three ways to implement this reaction latency compensation. In Figure 3-8(a), we simply create two new lookup table outputs, $Q\ New$ and $D\ Taken$, which are

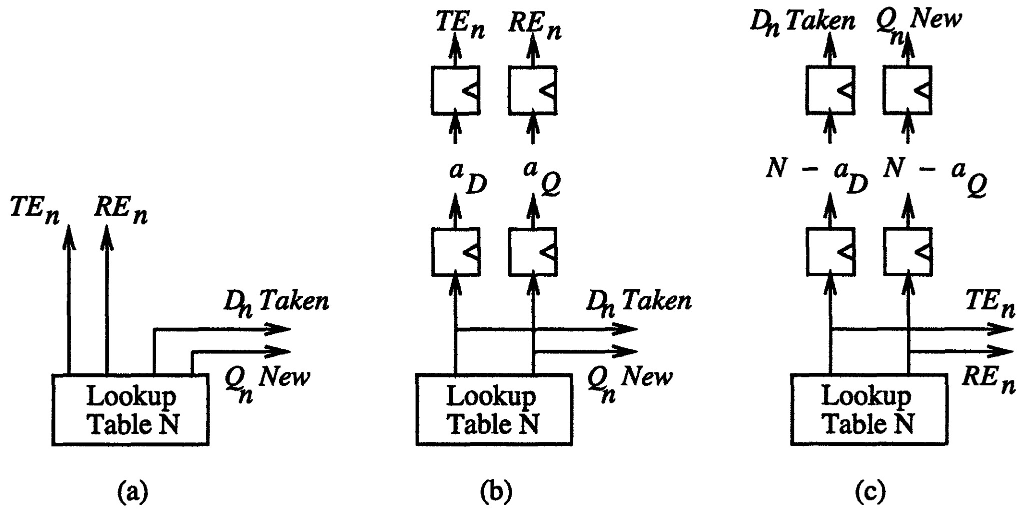


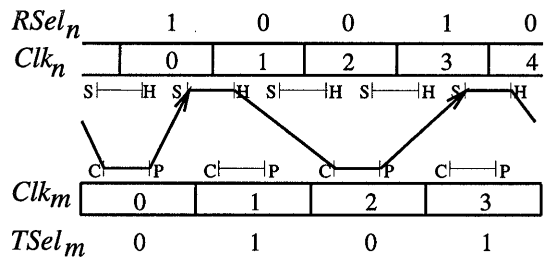
Figure 3-8: Interface hardware with reaction latency compensation.

time-rotated copies of RE and TE respectively. As their names imply, these signals serve the same function as the handshaking signals described in section 2.2.2 — i.e., $Q\ New$ tells the receiver that new data is available, and $D\ Taken$ tells the transmitter that the receiver has received its previous transmission and that new data must now be transmitted. Unlike in handshaking, however, these signals are synchronous and can be time advanced as necessary. Thus, they do not incur any additional latency.

The disadvantage of the circuit in Figure 3-8(a) is that it requires two additional bits of data in the ROM for each address. This results in an additional space requirement which grows as $O(N^3)$. The circuit in Figure 3-8(b) avoids this problem by deriving RE and TE from $Q\ New$ and $D\ Taken$ using shift registers. As shown, if a_Q and a_D are the number of clock cycles that $Q\ New$ and $D\ Taken$ respectively are advanced, then we simply insert a_Q and a_D flip-flops to the corresponding lookup table outputs to produce the RE and TE signals respectively. This circuit is significantly more space-efficient than the previous one since its space requirement only grows as $O(N)$ instead of $O(N^3)$. [2]

Figure 3-8(c) shows an alternative circuit similar to Figure 3-8(b) but where $Q\ New$ and $D\ Taken$ are derived from RE and TE instead. Here, we take advantage of the periodicity of the signals to simulate the necessary advances using delays. As shown, advances of a_Q

[2]Note that the worst-case magnitude that a_D and a_Q can have is $N-1$ since a delay a with magnitude greater than N is equivalent to a delay of $a \bmod N$ cycles.



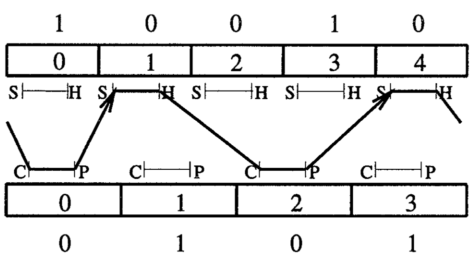


Figure 3-9: Using rational clocking with non-coincident clocks. (a) Clock skew can introduce a phase difference. (b) Respecifying the timing parameters.

and a_D are implemented as delays of $N - a_Q$ and $N - a_D$ respectively. This circuit has the advantage that the lookup tables can be programmed without knowledge of how much reaction latency compensation is needed by the target application. Different amounts of delay needed for *Q New* and *D Taken* can be produced as necessary by tapping the shift register at the appropriate points.

3.5.5 Non-coincident Clocks

In some applications, it may be not be possible to generate clocks that have exactly coinciding edges. Clock skew, for example, can introduce a phase difference between the two clocks. As long as the phase difference is known, however, rationally clocked communications can still be achieved between the two systems by first respecifying the timing parameters, and then applying the appropriate techniques as if the clocks had coinciding edges.

Figure 3-9 shows an example of this *retiming* procedure. Here, a simple shift in the timing parameter specifications allows us to schedule communications as we would for clocks with coinciding edges. If the phase difference is not exact as in Figure 3-9 but can take on a range of values (as might happen if there is phase jitter due to the PLL), then retiming would entail widening the timing windows to reflect the uncertainty introduced into the clocks' phase relationship. Appendix A presents some general rules for retiming.

Retiming can occasionally result in unconventional timing parameter values — i.e., values that are negative or larger than the clock period. In Figure 3-9(b), for example, the setup time has become negative. In some cases, it is possible to retime *both* systems, instead

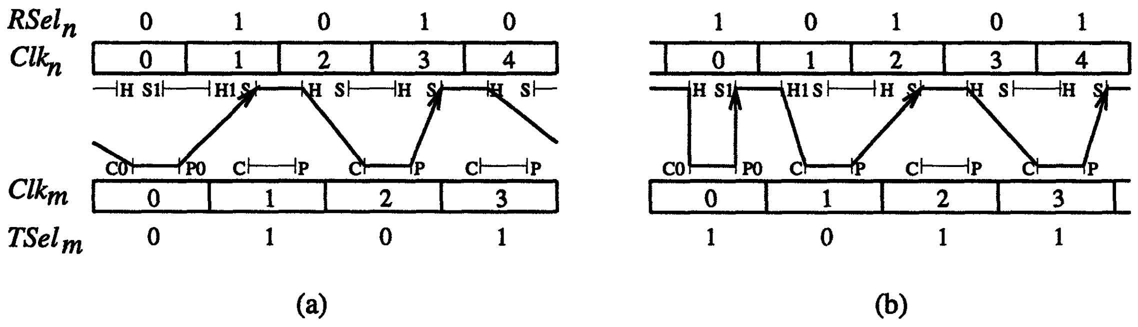


Figure 3-10: Using non-coincident clocks to increase throughput. (a) Throughput with coincident clocks is 50%. (b) Throughput with one clock delayed appropriately is 75%.

of just one, so that the timing parameters of each system remain within conventional constraints. If this is not possible, however, then we must make sure that our implementation of the scheduling algorithm works as described in section 3.3 even when given unconventional timing parameters.

It is sometimes possible to increase throughput efficiency by adding an appropriate phase difference and then retiming as necessary. For example, in Figure 3-10, the throughput efficiency is raised from 50% to 75% by delaying Clk_n enough to move S_1 past P_0. This technique is similar to the IBM approach described in section 2.3.3, and is useful when accurately delaying one of the clocks is not difficult to do.

Chapter 4

Double-Buffering

As Figure 3-5 shows, rational clocking works quite well given "typical" timing constraints. However, as shown in Figure 4-1, doubling the timing parameters (i.e., $S = 40\%$, $H = 20\%$, $C = 30\%$, $P = 40\%$) results in a significant drop in efficiency for a number of frequency pairs. This problem becomes relevant in high-speed systems where the timing margins are intentionally made narrow to get as much performance as possible. In these systems, the "typical" constraints would be considerably tighter than we have initially assumed and the efficiency of the current implementation may deteriorate significantly.

Since one system is faster than the other, it may seem that the slower one should never have to skip (i.e., not transmit or receive on) any cycles since the faster one can always "catch-up". The problem is that the non-zero widths of the transition and decision windows make it possible for one transmission to *contaminate* a previous transmission. This happens in Figure 3-4, for example, where system m cannot transmit on clock edge m_3 because doing so results in contaminating the data transmitted on clock edge m_2.

Fortunately, this problem can be solved with a simple hardware extension: a *second* transmit register. Whenever transmitting during the current cycle would result in contamination of the transmission from the previous cycle, we place the data in the second transmit register. When the receiver is ready to receive the new data, it then reads from the second register as well. Though simple, this *double-buffering* technique, is highly effective and, in almost all cases, guarantees 100% throughput efficiency regardless of the specific frequency

Transmit	Receive Frequency															
Freq.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	0	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
3	100	100	100	66	66	100	100	100	100	100	100	100	100	100	100	100
4	100	50	66	100	75	100	100	100	100	100	100	100	100	100	100	100
5	100	100	66	75	100	60	80	80	100	100	100	100	100	100	100	100
6	100	100	33	100	60	100	66	66	100	66	100	100	100	100	100	100
7	100	100	100	75	60	66	100	71	71	71	71	100	100	100	100	100
8	100	100	100	50	60	66	71	100	75	75	62	100	87	100	100	100
9	100	100	100	75	80	100	71	62	100	66	77	66	77	77	66	100
10	100	100	100	100	40	66	57	75	66	100	70	60	70	80	100	80
11	100	100	100	100	80	83	57	62	66	60	100	63	63	72	63	72
12	100	100	100	100	100	50	71	100	66	60	63	100	66	66	75	66
13	100	100	100	100	80	83	85	62	55	70	63	66	100	69	69	76
14	100	100	100	100	100	100	42	75	55	60	72	66	69	100	64	71
15	100	100	100	100	100	100	85	75	66	100	63	75	61	64	100	66
16	100	100	100	100	100	100	71	50	77	60	54	66	61	71	66	100

Figure 4-1: Throughput efficiency when the timing parameters are doubled.

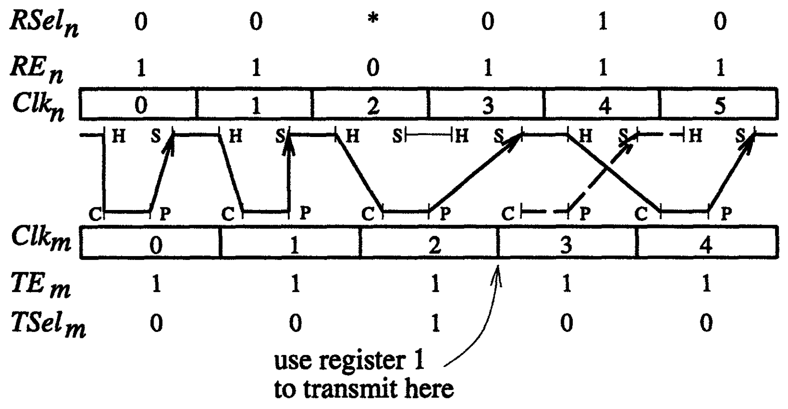


Figure 4-2: Getting 100% throughput efficiency using a second transmit register.

ratios and timing parameters.

Figure 4-2 shows how double-buffering can be used to achieve 100% throughput efficiency in the example from Figure 3-4. Here the dashed arrow from P_3 to S_5 shows that a second register can be used to transmit on clock edge m_3 and receive on clock edge n_5. This register is selected by two new control signals from the lookup tables, $RSel$ and $TSel$. As shown, $TSel_m[2]$ is set to 1 to tell the interface hardware to use register 1, instead of register 0, for transmitting on clock edge m_3. Correspondingly, $RSel_n[4]$ is set to 1 as well.

This chapter discusses the double-buffering technique in detail, beginning with a proof of its effectivity. New scheduling algorithms are discussed, followed by their hardware implementations. A few variations on double-buffering are also presented.

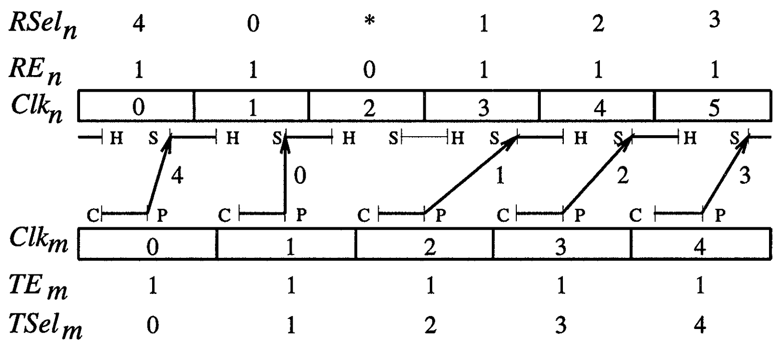


Figure 4-3: Avoiding contamination by using a new register for every transmission.

4.1 Why Double-Buffering Works

It is easy to show that double-buffering provides 100% throughput efficiency in almost all practical applications. Consider first the case where the transmitter, system m, is slower than the receiver, system n. If we have an unlimited supply of registers, we can eliminate the possibility of contamination by using a new register for each transmission, as shown in Figure 4-3 (the register used is indicated beside each arrow). Since the clock period of the receiver is shorter than that of the transmitter, there will always be a setup edge between any two propagation edges. Thus, the transmitter can transmit on every cycle and the receiver can always receive the transmission without losing data — that is, we can always achieve 100% throughput efficiency.

We can get the same result with a limited number of registers by *reusing* registers. In general, if R registers are used in sequence and then reused in the same sequence, there would be R system m clock cycles between when a register is used and when it is reused. Thus, by making R large enough to guarantee that a data transfer using a register will always complete within $R \cdot T_M$ seconds, we can successfully emulate an unlimited supply of registers and achieve 100% throughput efficiency.

Figure 4-4 shows how we determine a sufficiently large value of R. In this figure, t_{pc} is the maximum amount of time that a register can take to complete a data transfer without being contaminated by a new transmission using the same register. As shown in the figure, it is the time between the propagation edge of a cycle when a register is used and the

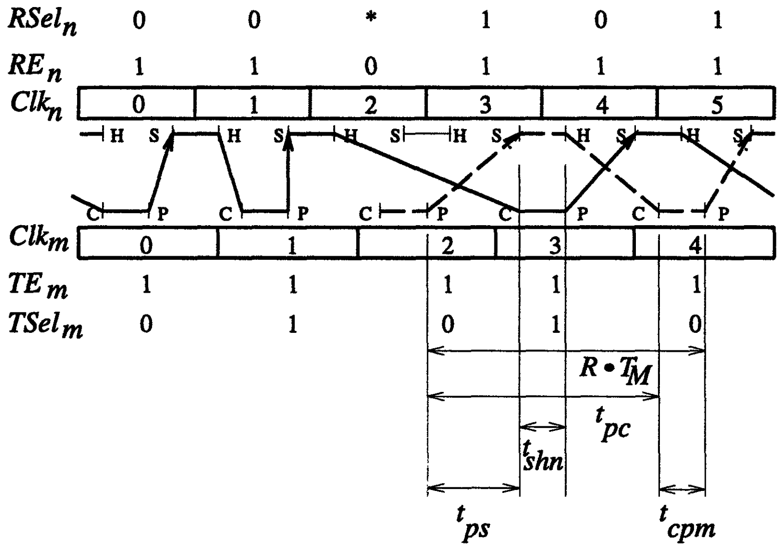


Figure 4-4: Determining R in the slower transmitter case.

contamination edge of the cycle when the register is reused. That is,

$$t_{pc} = R \cdot T_M - t_{cpm} \tag{4.1}$$

where t_{cpm} is the width of the transition window of system m. The actual time that a register takes to complete a data transfer is given by $t_{ps} + t_{shn}$, where t_{ps} is the time between the propagation edge used for the first transmission and the next available setup edge, and t_{shn} is the width of the decision window of system n.

In order simulate an unlimited supply of registers, we must ensure that contamination is not possible. That is, that

$$t_{pc} \geq t_{ps} + t_{shn} \tag{4.2}$$

Substituting the definition of t_{pc} from Equation 4.1, and rearranging terms, we get:

$$R \cdot T_M \geq t_{ps} + t_{shn} + t_{cpm} \tag{4.3}$$

We can simplify this by noting that even though t_{ps} changes from cycle to cycle, it is always less than T_N. Thus, we have

$$t_{ps} < T_N < T_M \tag{4.4}$$

and Inequality 4.3 will be satisfied if

$$R \cdot T_M \geq T_M + t_{shn} + t_{cpm} \tag{4.5}$$

Rearranging terms, we get the following *master* inequality for determining R:

$$R \geq \frac{t_{shn} + t_{cpm}}{T_M} + 1 \tag{4.6}$$

The master inequality shows that R depends directly on the restrictions placed on the widths of the decision and transition windows. In the *ideal case*, we have

$$t_{shn} = t_{cpm} = 0 \tag{4.7}$$

and *one* register would be sufficient. In the *worst* case, we know that these windows cannot be wider than the clock periods of the systems they belong to — otherwise, communication will not be possible even within the systems themselves. Thus, we have:

$$t_{shn} \leq T_N < T_M, \quad t_{cpm} \leq T_M \tag{4.8}$$

and *three* registers would be needed to eliminate contamination. In *most* cases, however, the timing parameters of the registers would be such that if we clock system n with system m's clock, the two systems would be able to transfer data through the registers in a conventional synchronous manner. Specifically, the transmitter's transition window and the receiver's decision window would both fit in one clock period of the slower system, giving us

$$t_{shn} + t_{cpm} \leq T_M \tag{4.9}$$

and allowing us to guarantee 100% throughput efficiency with only *two* transmit registers.

The analysis for the case where the receiver is slower (i.e., system m is receiving) is similar. The time intervals involved are shown in Figure 4-5. Instead of t_{pc}, we now consider t_{hs}, the maximum amount of time that the transmitter can take to put new data in the register and allow the data to settle before the receiver expects new data to be available (i.e., if the new data is not available and stable within t_{hs} seconds, the receiver will have to skip a cycle and will thus not get 100% throughput). Using the actual time taken by the transmitter, $t_{hc} + t_{cpn}$, and similar steps, we arrive at a master inequality analogous to Inequality 4.6, and the conclusion that two transmit registers are enough to guarantee 100% throughput efficiency in the slower receiver case as well.

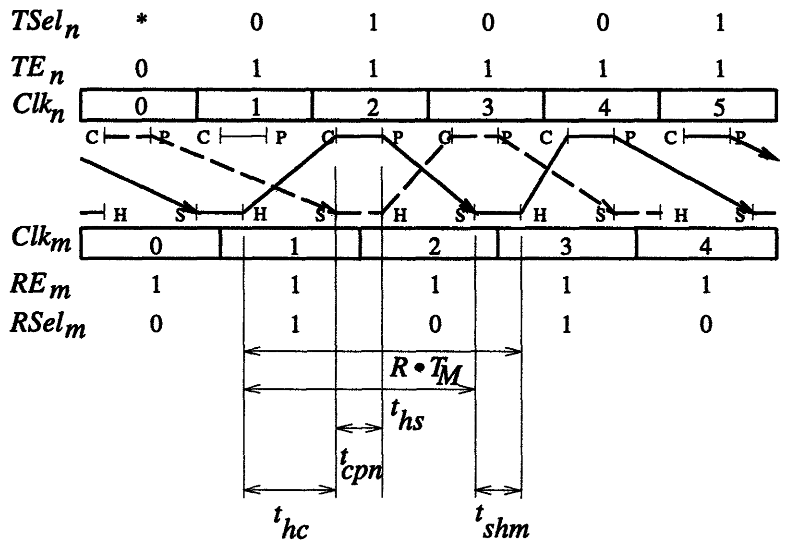


Figure 4-5: Determining R in the slower receiver case.

4.2 Communication Scheduling

One way to schedule double-buffered communications is to first use the algorithm for the single-buffered hardware, and then do a second pass through the cycles that failed to check if data transfer is possible using a second transmit register. This was done in Figure 4-2. The elimination of contamination problems through double buffering, however, creates a new possibility: if we set $RSel$ and $TSel$ to alternate between using the two registers, then scheduling each transaction on the slower system can be done *independently* of the transaction immediately following it. A number of such *contamination-free* algorithms are discussed in this section.

4.2.1 Slower Transmitter Case

For the slower transmitter case, the algorithm is simple: we draw arrows from each propagation edge P_i to the next setup edge S_i at or after it, and then set $TSel$ and $RSel$ so that the two registers are used alternatingly. Figure 4-6(a) shows an example where $M = 4$ and $N = 5$. (In this figure RE and TE are 1 except during cycles where $RSel$ and $TSel$ respectively are marked with an asterisk.)

Note a potential problem in scheduling communications with double-buffering — if M

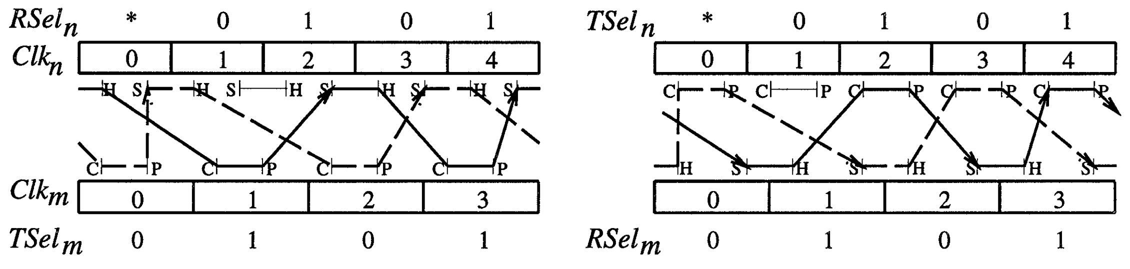


Figure 4-6: Scheduling for double-buffering. (a) Slower transmitter case. (b) Slower receiver case (lazy algorithm).

is odd, we cannot alternate perfectly between the two registers. This problem occurs in Figure 4-4. This is important to note since double-buffering is only guaranteed to work if no register is ever used for two consecutive cycles. Otherwise, simply scheduling transactions independently from one another may result in contamination. Fortunately, as in the case shown in Figure 4-4, contamination does not always occur, and we can often get away with scheduling transactions independently even if M is odd. If contamination does occur, however, the problem is easily solved by multiplying both M and N by two and making lookup tables based on the resulting equivalent ratio instead.[1]

4.2.2 Slower Receiver Case

For the slower receiver case, there are two basic algorithms: the *lazy* and *greedy* algorithms.

The lazy algorithm is similar to the slower transmitter algorithm except that it uses the H and C edges instead of the P and S edges respectively. That is, we draw a line from some edge H_i of the receiver to the next available edge C_j and then set the lookup tables to tell the transmitter to transmit at clock edge n_j using the same register used in receiving at clock edge m_i. Since $RSel_m$ is always alternating, we then know that we will receive the data on edge m_{i+2}. We call this a "lazy" algorithm since although it provides 100% throughput efficiency, it can introduce unwanted latency because the receiver always waits until clock edge m_{i+2} to receive even if it can already receive at clock edge m_{i+1}. This

[1]Multiplying M and N by two may make the PLL less stable. In this case, it may be good to use separate counters for clock generation and for indexing the lookup tables.

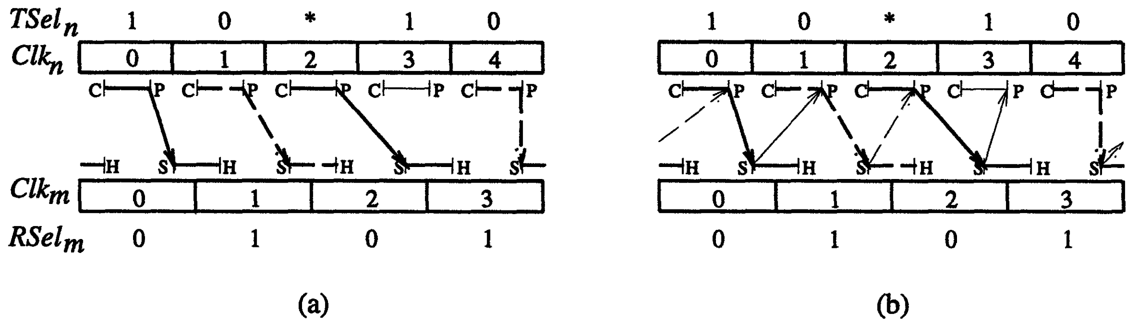


Figure 4-7: The greedy slower receiver algorithm. (a) In reverse time. (b) In normal time.

algorithm was used in Figure 4-5. An example of this algorithm being used is shown in Figure 4-6(b).

The greedy algorithm is the opposite of the lazy algorithm — it produces a schedule where the receiver gets new data as soon as possible after the transmitter sends it. If we are allowed to work in *reverse time*, then this algorithm is easy to describe — we simply draw reverse arrows from an S edge to the nearest P edge at or *before* it. If reverse time scheduling is not possible, as with the run-time methods to be presented in the next chapter, then implementing the greedy algorithm becomes slightly more complicated. First, we draw *temporary* arrows from each setup edge S_i of the receiver to the nearest propagation edge P_j of the transmitter after *but not at* it. Then, we schedule a transfer from P_{j-1} to S_i. Since P_j was chosen to be the nearest edge *after* S_i, then P_{j-1} must be the nearest edge *at or before* S_i. Thus, this implementation is exactly equivalent to the reverse time version, but has the advantage of only requiring a time shift instead of a complete time reversal. Figure 4-7 shows the two ways to implement the greedy algorithm. In Figure 4-7(b), the temporary arrows are indicated by thin arrows, while the actual transfers are indicated by the thick arrows.

Note that the greedy algorithm is actually only "greedy" from the receiver's perspective. It is not greedy from the transmitter's view because it does not always let the transmitter transmit data at the earliest possible time. In fact, given the choice of several system n clock edges from which to transmit data that is to be received at a particular system m clock edge, the greedy algorithm would always choose the *latest* of these edges since this results in the receiver receiving the data the soonest after it has been transmitted. This

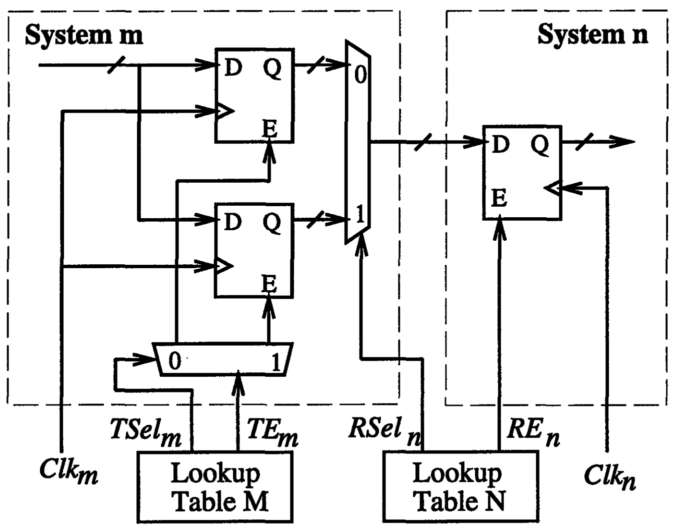


Figure 4-8: Double-buffered hardware (only one direction shown).

extra latency, however, is actually an advantage rather than a disadvantage, since it allows the transmitter more time to perform computation before being required to produce the next piece of data to be passed to the receiver.

If ease-of-implementation is not an issue, then it is generally better to use the greedy algorithm than the lazy algorithm. Aside from having lower latency for the receiver and allowing greater compute time for the transmitter, the greedy algorithm is also more likely to work in situations where M (the slower system's frequency) is odd. Although neither algorithm is guaranteed by the proof in section 4.1 to work correctly if M is odd, the lazy algorithm is more likely to fail because it explicitly assumes that $RSel$ always alternates.

4.3 Hardware Implementation

4.3.1 Register Selection

Figure 4-8 shows one way to implement double-buffering in hardware (only one direction is shown in this figure). Data to be transmitted is fed into the inputs of two transmit registers, but only at most one of these registers latches the data at a particular clock edge. The transmit register to be used is specified by the $TSel$ control signal from the lookup table. $TSel$ controls a demultiplexer that passes the value of TE to the desired register and disables the other. As in the original hardware of Figure 3-3, the registers here may be replaced by the modules of section 2.1 if reducing latency is important. Note, however, that any combinational logic in the modules must be duplicated.

The transmit register from which the receiver will sample data is selected by having $RSel$ control a multiplexer placed before the input of the receive register. This multiplexer is built within the transmitter (system m), as shown in the figure, in order to avoid having to run two sets of data wires between the two systems. (Alternatively, we can build the two transmit registers into the receiver.) Note that the effective setup and hold times of the receiver are affected by the delay through the multiplexer, and must be recomputed accordingly. Also note that when implementing this circuit, it is important to constrain $RSel$ not to change in such a way that would cause a setup or hold time violation in the receive register that would not have otherwise happened in the single-buffered configuration. Fortunately, since $RSel$ is synchronous with the receiver's clock, this restriction is not hard to meet.

4.3.2 Communication Scheduling

Double-buffering allows us to make some simplifications in the scheduling hardware for the slower system. First, since the slower system (assume that it is system m) never has to skip any cycles, TE_m and RE_m are always equal to 1 and need not be taken from a lookup table. Similarly, the $RSel_m$ and $TSel_m$ signals system are both simple alternating sequences of 1's and 0's and can be derived from the least significant bit (LSB) of the cycle count value. Thus, as shown in Figure 4-9, lookup tables are unnecessary for the slower system.

If we further assume that the slower system always has an even number of clock cycles within a coincidence cycle (i.e., M is even), then we can simplify the faster system's hardware as well by using a toggle (T) flip-flop to generate $TSel_n$ and $RSel_n$, as shown in Figure 4-9.

In this configuration, RE_n controls the enable input of the T flip-flop such that the flip-flop only toggles after the register indicated by the current value is used. The flip-flop is initialized to the precomputed value of $RSel[1]$ through a synchronous load controlled by the Z output of the index counter. This value is either fed directly by the user, or taken

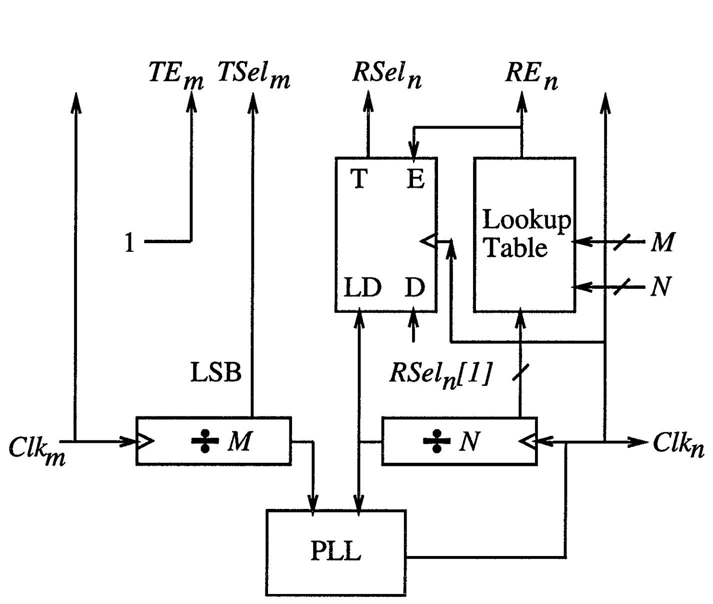


Figure 4-9: Modified scheduling hardware (only one direction shown).

from a table containing corresponding $RSel[1]$ values for different frequency ratios. (Note that if $RE[1]$ is 0, then $RSel[1]$ must be made equal to $RSel[2]$.) Alternatively, we can assume that $RSel[1]$ is always 0, and then adjust $TSel$ if necessary by taking it from the negation of the LSB. In this way, the T flip-flop need only support synchronous clear, which is simpler to implement than synchronous load.

Note that if M is not even, the circuit shown may not produce the correct sequence because of the forced discontinuity in the LSB sequence that occurs when the index counter wraps-around. The sequence for $TSel_n$ in Figure 4-5, for example, cannot be produced by the hardware in Figure 4-9. If desired, this problem can be solved by using a T flip-flop for the slower system as well. However, the best solution is still to multiply M and N by two to make M even.

4.4 Variations

4.4.1 Triple and Quadruple Buffering

As shown in section 4.1, some applications may require three transmit registers to eliminate contamination. The same scheduling algorithms can still be used for these cases except that *TSel* and *RSel* must now repeatedly cycle from 0 to 2, instead of just alternating between 0 and 1. This means that these signals must each be made two bits wide, and must be generated using modulo-3 counters. Also, M must be a multiple of three to strictly guarantee that contamination is eliminated.

A more interesting variation of the double-buffering technique uses *four* transmit registers. Although the analysis in section 4.1 shows that we never actually need more that three buffers to guarantee 100% throughput efficiency, having four buffers becomes useful if the timing parameters of the systems are *unknown*. In particular, it allows us to generate a "generic" schedule that works with different timing parameters by allowing us to assume *worst-case* values for the timing parameters — that is, by allowing us to specify timing parameters that are conservative enough to cover the timing parameters of *any* real system.

Consider, for example, the slower transmitter case where system M is transmitting to system N. The worst-case values of the four timing parameters in this case are:

$$t_{sn} = T_N, \quad t_{hn} = T_N \tag{4.10}$$

$$t_{cm} = 0, \quad t_{pm} = T_M \tag{4.11}$$

where t_{sn} and t_{hn} are the setup and hold times of system n, and t_{cm} and t_{pm} are the contamination and propagation delays of system m. Given these parameters, we have

$$t_{shn} = 2T_n < 2T_m \tag{4.12}$$

$$t_{cpm} = T_m \tag{4.13}$$

Substituting these into the master inequality (4.6), we see that in this case, *four* transmit registers are needed to eliminate the possibility of contamination. A similar argument with the same result can be made for the slower receiver case.

This result means that given four transmit registers, we can generate a single schedule that will work regardless of the actual systems' timing parameters. Figure 4-10 shows an example of such a schedule for $M = 4$ and $N = 5$. In this figure, the register used by a data transfer operation is indicated beside the arrow corresponding to the operation.

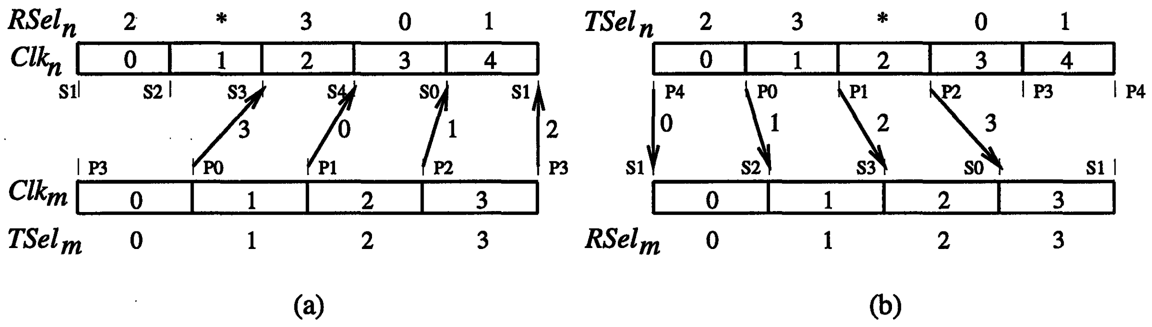


Figure 4-10: Using four buffers. (a) Slower transmitter case. (b) Slower receiver case (greedy algorithm).

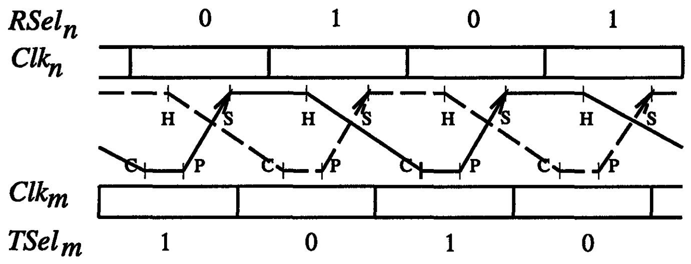


Figure 4-11: Using double-buffering for communication between out-of-phase systems.

4.4.2 Using Double-Buffering for Out-of-Phase Communications

Another interesting application of double-buffering is in providing communications between two systems that are running at the *same* frequency but cannot otherwise communicate because of a phase difference between their clocks. Since we allowed t_{ps} to assume any value from 0 to T_N in Inequality 4.4, the analysis in section 4.1 applies regardless of the phase difference between the two clocks.

One possible application is the case shown in Figure 2-4(b). As previously discussed, the phase difference between the two clocks in this example prevents the systems from performing data transfers every clock cycle. Blindly transmitting and sampling every cycle results in contamination, while employing flow control results in 50% throughput efficiency. With double-buffering, however, the problem disappears and 100% throughput efficiency can be achieved as shown in Figure 4-11.

Note that double-buffering does *not* solve the synchronization problem. The problem has just been converted to that of assigning the right values of $TSel$ and $RSel$ for each cycle. If the phase difference is known, then these values can easily be precomputed.[2] If the phase difference is unknown, then determining these values requires some form of synchronization. We may perhaps be able to take advantage of the fact that there are only two possible functionally-distinct assignments and at least one of these is guaranteed to work. Since this paper is concerned primarily with synchronous techniques, we do not develop this idea further. It may be worthwhile, however, to make this idea a subject of future research since it provides an alternative to conventional techniques which force the clocks to have coinciding edges. Similar ideas are described in [23, 24, 25].

[2]It is only necessary to compute the values of $TSel$ and $RSel$ for one particular clock cycle on each system. These values are used to initialize a T flip-flop that will generate the desired alternating sequence of 1's and 0's.

Chapter 5

Run-Time Communication Scheduling

The table-based implementation of rational clocking suffers a major drawback: it requires a large amount of space. A lookup table for a particular ratio M/N and a given set of timing parameters needs $O(\max(M, N))$ bits of memory. A generic lookup table which allows the frequency ratio numerator and denominator to vary from 1 to N needs $O(N^3)$ bits. A generic lookup table which further allows the four timing parameters to be varied independently needs as much as $O(N^7)$ bits. This tremendous growth rate makes generic table-based implementations impractically slow, costly, and inflexible.

Fortunately, it is possible to exploit the regularity of the clocks' interactions to produce the appropriate control signals using significantly less space. Since the difference between the two clock frequencies is constant, the interacting timing windows will move relative to each other at a constant rate. Thus, it is only necessary to compute their initial positions — from there, subsequent positions can be computed on-the-fly using counters.

This chapter discusses how run-time scheduling can be implemented. We begin with the algorithms and hardware for the simpler case of contamination-free (i.e., double-buffered) communications, and move on to a general algorithm that accounts for contamination. This leads to a general hardware scheduler capable of producing schedules for both single and double-buffered communication.

5.1 The Basic Algorithm

The algorithms used for run-time scheduling are based on a graphical algorithm similar to *Bresenham's algorithm* for drawing lines on raster displays [26]. In this section, we describe a *basic* algorithm for the simpler task of scheduling contamination-free (i.e., double or triple-buffered) communication. In section 5.3, we modify this algorithm and come up with a *general* algorithm that works for single-buffered communication as well.

5.1.1 Slower Transmitter Case

The run-time scheduling algorithm makes use of a two-dimensional version of the schedule diagram called the *schedule plot.* Figure 5-1 shows a schedule plot being used to schedule receptions on system n in the double-buffered slower transmitter case.

The schedule plot is drawn on a grid with axes spaced in units of time $\Delta t = T_{MN}/MN$. The occurrences of system m's and system n's clock edges are indicated on the y and x axes respectively by horizontal and vertical lines spaced N and M grid units apart, as shown. (The spacing is derived from Equation 3.2.) The function, $f(t)$, equal to the time that system m's transmit register must clock-in new data in order for system n to properly sample the data by clocking the receive register at time t, is plotted on the grid. As shown,

$$f(t) = t - (P_m + S_n) \tag{5.1}$$

where P_m and S_n are the propagation and setup times expressed in units of Δt. This is because system n must wait P_m time units for the receive register's input to stabilize after system m clocks data into the transmit register, and an additional S_n time units of setup time before it can clock its receive register. The example in Figure 5-1 has both P_m and S_n equal to 2, and thus has $f(t) = t - 4$.

With the schedule plot, determining the clock edges on which system n can receive new data is straightforward: for each vertical line x, we plot a dot on the nearest horizontal line

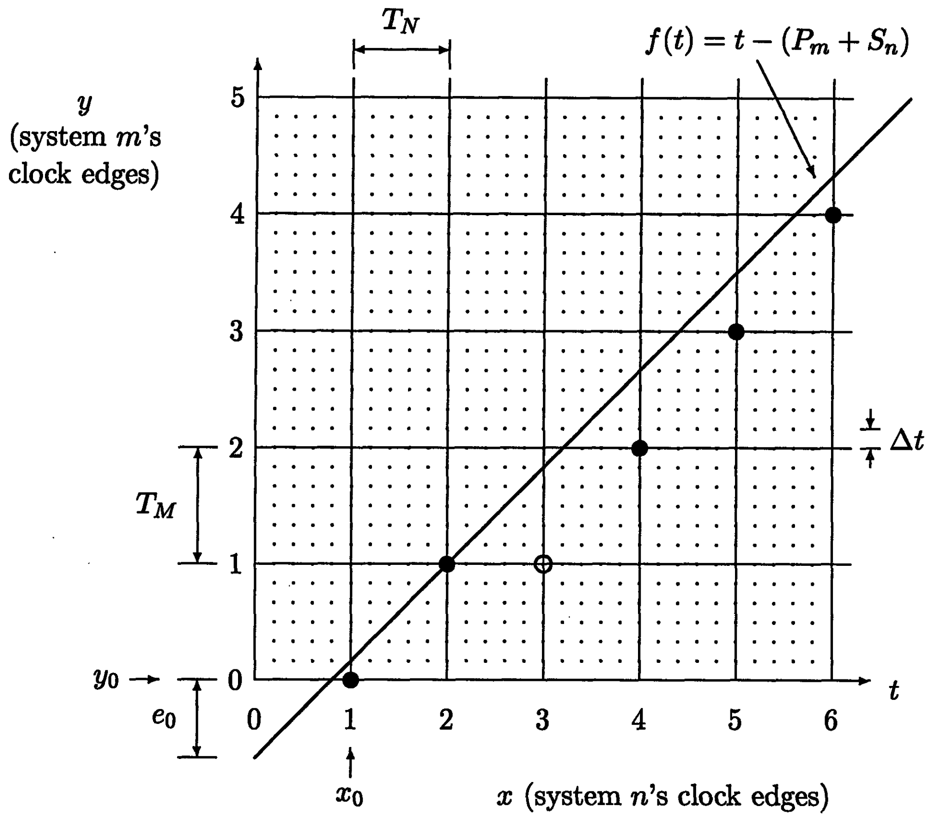


Figure 5-1: A schedule plot for the case where $M = 5$ and $N = 6$.

at or below $f(t = M \cdot x)$. This dot indicates the last system m clock edge, y, whose data system n can safely receive at time t. If the dot moves up, system n can receive new data. If the dot stays on the same line, then no new data is available yet, and system n should inhibit reception.

By thus identifying the earliest x at which $f(t)$ is at or above the horizontal line y, we effectively identify the nearest system n setup edge S_x at or after each system m propagation edge P_y. That is, this algorithm is equivalent to the greedy slower transmitter algorithm from section 4.2, and can be used to schedule communications in the slower transmitter case. The following algorithm shows how this is done:

```
1. Initialize all entries of $TSel_m$, $TE_m$, $RSel_n$, and $RE_n$ to 0.
2. $x = 1$                              ▷ $x_0 = 1$
3. $f_0 = \lfloor -(P_m + S_n) \rfloor$              ▷ get $f(t)$ at clock edge $x_0 - 1$
4. $y = f_0$ div $N$                      ▷ $y_0$ = first line above $f_0$
```

```
5.  if ($f_0 \geq 0$) or ($f_0 \bmod N = 0$) then
        $y = y + 1$                              ▷ adjust $y$
6.  $e_{old} = (f_0 \bmod N) - N$                ▷ $e_0$ = distance from line $y$ to $f_0$
7.  $R = 0$                                      ▷ use register 0 first
8.  do $N$ times
9.      $e = e_{old} + M$                        ▷ compute error term for current $x$
10.     if ($e \geq 0$) then                     ▷ is new data available?
11.         $TSel_m[(y-1) \bmod M] = R$          ▷ yes, transmit data
12.         $TE_m[(y-1) \bmod M] = 1$
13.         $RSel_n[(x-1) \bmod N] = R$
14.         $RE_n[(x-1) \bmod N] = 1$
15.         $R = \overline{R}$                   ▷ use other register next time
16.         $y = y + 1$                          ▷ wait for next system $m$ clock edge
17.         $e = e - N$                          ▷ make $e$ relative to new $y$
18.     $x = x + 1$                              ▷ get next $x$
19.     $e_{old} = e$                            ▷ get next $e_{old}$
```

Here, the variable x indicates the system n clock edge (vertical line) currently being considered. It is initialized to $x_0 = 1$ in line 2, and incremented at the end of each loop iteration.[1] The variable y indicates the nearest system m clock edge (horizontal line) from which data has *not* yet been received. It is initialized in lines 4 and 5 so that it points to the first horizontal line above $f(t = 0)$, and incremented in line 16 after a system n clock edge that can receive the data has been found. In terms of the graphical algorithm, y indicates the nearest horizontal line which does not have a dot plotted on it yet.

For each x, the availability of new data can be determined by checking the *error term*, e, equal to the distance from the horizontal line y to $f(t = M \cdot x)$. This error term is equal to the time difference between setup edge S_x and propagation edge P_y, and indicates how early or late system n would be if it tries to receive data from clock edge m_y on clock edge

[1]The choice of $x_0 = 1$ means that the case for $x = 0$ is not considered until the last iteration of the loop, when $x = N$. Although seemingly unintuitive, this choice, as will be shown in section 5.1.2, simplifies analysis and implementation by allowing us to use $f(t = 0)$ in computing e_0 and y_0.

n_x. If e is zero, then system n would be exactly on time, and the transmitted data would be stable just at the setup edge of clock edge n_x. If e is positive, then system n would be late, and the data would be stable for $e\Delta t$ seconds before the setup edge. If e is negative, then system n would be too early, and the data would not be stable until $|e|\Delta t$ seconds after the setup edge. Being late is not a problem (at least not with contamination-free communication), but being too early requires reception to be postponed by at least $|e|\Delta t$ seconds to give data enough time to be available and stable.

The initial error, e_0, is computed in line 6 relative to y_0, the first horizontal line above $f(t=0)$. Then, at each iteration, the current error e is computed *incrementally* by adding M (i.e., the length of time between system n clock edges) to the previous error, e_{old}. As soon as e becomes nonnegative, data transfer is enabled from clock edge m_y to clock edge n_x by setting the appropriate entries of RE and TE to 1, and setting $RSel$ and $TSel$ to use the register indicated in the variable R. After these signals are set, R is toggled to avoid contamination, and y is incremented so that it indicates the next system m clock edge. To make it relative to the new y, e_{old} is then set to $e - N$. This adjustment also keeps the value of e_{old} negative and ensures that the condition in line 10 is satisfied if and only if the current S_x is the nearest setup edge at or after P_y.

5.1.2 Computing y_0 and e_0

Given an initial system n clock edge, x_0, the initial values of y and e_{old} must be computed such that e_{old} is negative and equal to the error term at vertical line $x_0 - 1$. This can be done by first computing the initial function value, $f_0 = \lfloor f(t = M \cdot (x_0 - 1)) \rfloor$, equal to

$$f_0 = \lfloor M \cdot (x_0 - 1) - (P_m + S_n) \rfloor \tag{5.2}$$

(where the floor function is used to accommodate non-integer timing parameters as will be discussed in the next section), and then choosing y to be the first horizontal line above f_0, and e to be the distance from line y to f_0.

Figure 5-2 shows the three possible cases that need to be considered. (Note that the vertical and horizontal lines in this figure are spaced at intervals of $M\Delta t$ and $N\Delta t$ units

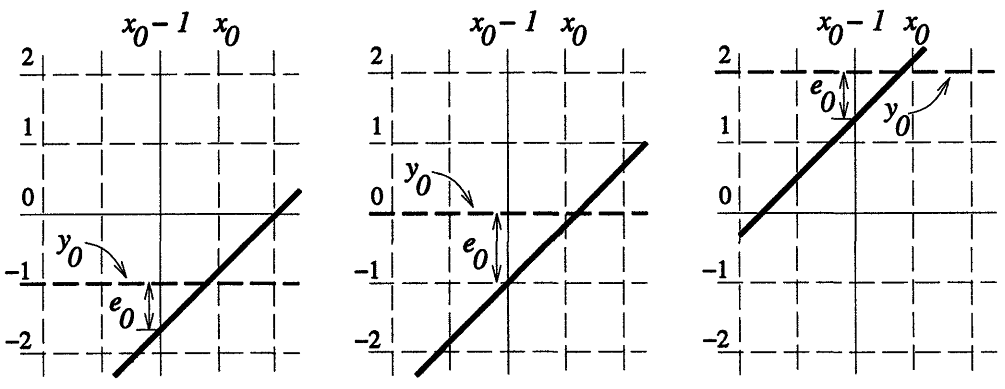


Figure 5-2: Three cases in computing y_0 and e_0. (a) $f_0 < 0$. (b) $(f_0 \bmod N) = 0$. (c) $f_0 \geq 0$

respectively.) If $f_0 < 0$, and f_0 does not land exactly on a horizontal line, then y_0 is simply

$$y_0 = f_0 \text{ div } N \tag{5.3}$$

If f_0 is negative but lands exactly on a horizontal line, then we must add 1 to y to make it point to the line *above* f_0. Finally, if $f_0 \geq 0$, then we must also add 1 to y since the div operator will give us the line *below* f_0. Considering these cases leads to the two-step computation done in lines 4 and 5 of the basic algorithm, where y_0 is first computed according to Equation 5.3 and then adjusted as necessary. Computing e_0 is simpler — in all three cases, the following formula correctly computes e_0:

$$e_0 = (f_0 \bmod N) - N \tag{5.4}$$

As will be shown in section 5.2.4, although y_0 and e_0 can be computed in this way for any given x_0, choosing $x_0 = 1$ limits the range of f_0, and makes it easier to compute y_0 and e_0 in hardware.

5.1.3 Slower Receiver Case

The slower transmitter algorithm can be used to implement the lazy slower receiver algorithm if we appropriately change the names of the control signals being set, and make the following transformations:

$$\begin{aligned} P_m &\leftarrow H_m \\ S_n &\leftarrow -C_n \end{aligned} \tag{5.5}$$

where H_m and C_n are the hold and contamination times expressed in units of Δt. These transformations give us

$$f(t) = t - (H_m - C_n) \tag{5.6}$$

and enable the algorithm to determine the nearest contamination edge C_x at or after each hold edge H_y. The assignments in lines 11 to 14 (with the appropriate name changes) correctly set the control signals so that system n transmits at clock edge n_y using the same register used in receiving at clock edge m_x.

The normal-time greedy algorithm can also be implemented by using similar but slightly more complicated transformations. First, we use the transformations

$$\begin{aligned} P_m &\leftarrow -S_m \\ S_n &\leftarrow -P_n \end{aligned} \tag{5.7}$$

to get

$$f(t) = t + (S_m + P_n) \tag{5.8}$$

and have the algorithm determine the *temporary* arrows to be drawn from the system m setup edges to the system n propagation edges: Then, since the data transfer corresponding to a temporary arrow drawn from some edge S_y to some edge P_x actually transmits data on P_{x-1} instead of P_x, we initialize x to $x_0 - 1$ to get a one-cycle time-advance in the signals computed for system n.

We must also change the condition in line 10 to $(e > 0)$, to cover cases where a propagation edge P_j coincides exactly with a setup edge S_i. That is, we must make sure that in such cases, the greedy algorithm does *not* draw a temporary arrow from S_i to P_j yet, but draws one from S_i to P_{j+1} instead.

Finally, we must make y_0 and e_0 consistent with this new condition by making y_0 indicate the line *at or above* f_0, and allowing e_0 to be zero. This entails modifying the conditions in line 5 appropriately, and having an adjustment for e_{old} after line 6 that would set e_{old} to zero if it turns out to be $-N$.

Note that the modified condition may be harder to check for since we cannot simply look at the sign bit anymore. Thus, a hardware implementation of a strict greedy slower receiver algorithm may be slightly more complex than that for the lazy algorithm. If strict greediness is not necessary, however, then the original condition in line 10 can be kept so as to avoid the additional cost.

5.1.4 Non-integer Timing Parameters

In concept, the run-time scheduling algorithm itself does not place any restrictions on the values that can be taken on by the timing parameters. In practice, however, the hardware used to implement the algorithm would only be able to handle integers and would thus require that y_0 and e_0 both be integers, or equivalently, that f_0 be an integer. Fortunately, any non-integer f_0 will have a corresponding f_0 that will produce exactly the same communication schedule.

To see this, note that adding a small number, ϵ, such that $0 \leq \epsilon < 1$, to some integer value of f_0 does not change the resulting pattern of dots and its corresponding schedule. Thus, if f_0 is not an integer, we can still produce a correct schedule by simply taking the *floor* of f_0 (i.e., the greatest integer less than or equal to f_0) as done in line 3. This rounding step is equivalent to adjusting the timing parameters *conservatively* so as to make the resulting f_0 an integer while ensuring that the original timing windows are covered by the new ones.

5.2 Hardware Implementation

Figure 5-3 shows a circuit that implements the basic algorithm in hardware. It uses three different modules: *R Gen*, *E Gen*, and *LD Gen*.

5.2.1 The *R Gen* Module

The *R Gen* module stores the value R, which is used to set $TSel$ and $RSel$. It is simply a T flip-flop with enable (E) and clear (Clr) inputs. The E input is used to prevent R from

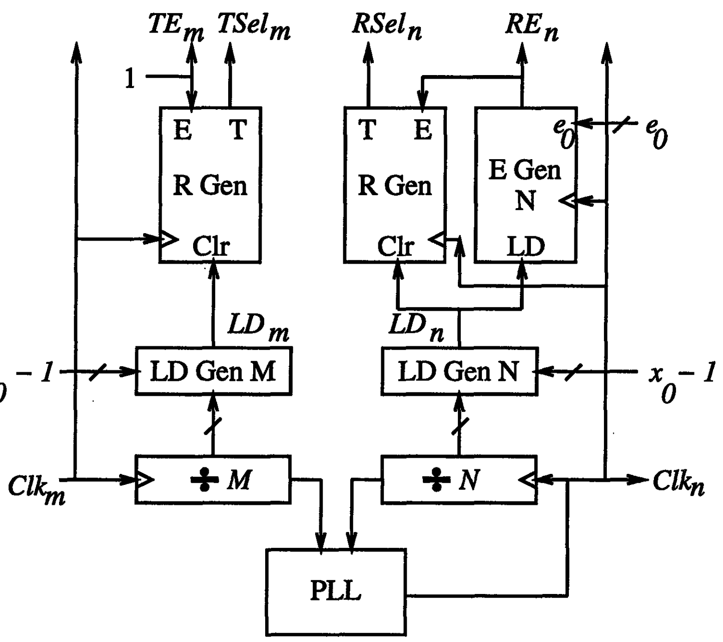


Figure 5-3: Basic run-time scheduler. (Only one direction shown.)

being toggled when no data transfer is being done. The *Clr* input is used to initialize *R*. It is a synchronous input which, when asserted, clears the flip-flop on the *next* clock edge.

5.2.2 The *E Gen* Module

The *E Gen* module generates the RE_n and TE_n signals for the faster system (in the slower system, these signals are simply tied to logical 1). As shown in Figure 5-4, this module is a direct implementation of the loop in lines 8 to 19 of the algorithm.[2] A register stores the value of e_{old}, which is used to compute the new values of e and RE_n. The *LD* input is used for initialization and causes e_{old} to be set to e_0 at the clock edge *after LD* is asserted.

In actual applications, we can make the last two stages of the *E Gen* module more space-efficient and possibly faster by replacing them with the circuit shown in Figure 5-5(a), where the NOR module is an array of NOR gates that NOR each bit of *N* with the

[2]The *E Gen* module shown works for both the slower transmitter algorithm and the lazy slower receiver algorithm. Section 5.3.3 shows the modifications necessary to make it implement the greedy slower receiver algorithm.

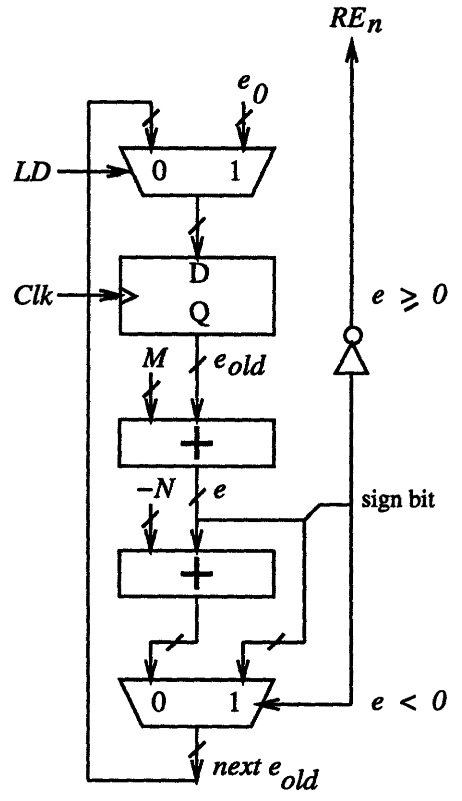


Figure 5-4: The *E Gen* module.

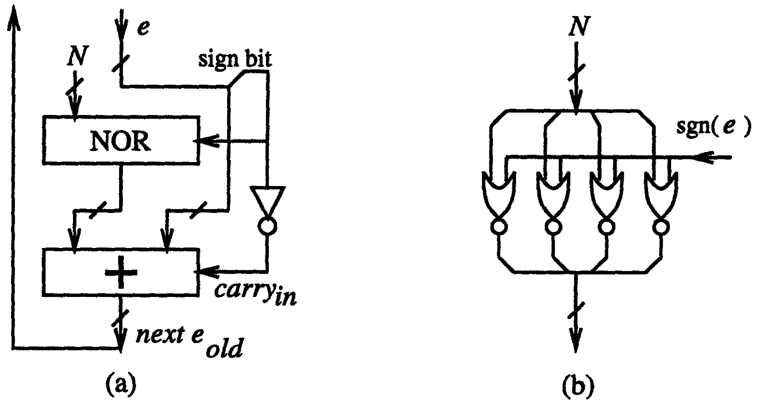


Figure 5-5: Using a NOR module instead of a mux in computing the next e_{old}.

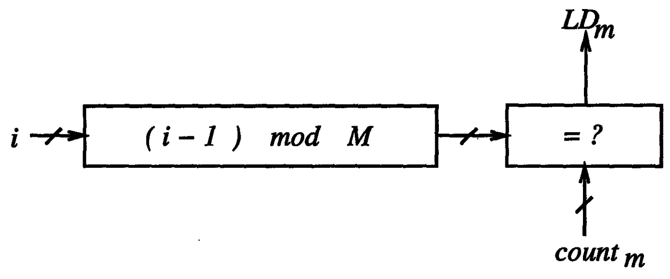


Figure 5-6: The *LD Gen* module.

sign bit of e, as shown in Figure 5-5(b). In this circuit, the adder will subtract N from e if e is nonnegative, and add 0 otherwise.

It is interesting to note that the *E Gen* module is actually a *rate multiplier* — a special counter that when clocked at one frequency, produces a sequence of pulses at a lower frequency by appropriately skipping cycles. Rate multipliers are commonly used in applications such as line drawing and digital wave synthesis. Oberman [27, 28] provides a good discussion of different ways to implement rate multipliers.

5.2.3 The *LD Gen* Module

The *LD Gen* module is used for periodically initializing the *R Gen* and *E Gen* modules in a way that causes the control signals to be generated at the correct times. It takes an input i and outputs a load signal (*LD*) that causes the connected components to initialize at the ith clock edge of every coincidence cycle. If we are using incrementing counters, then this can be done with a comparator that checks if the index counter value is equal to $(i-1) \bmod M$, as shown in Figure 5-6 (for system n, N is used instead).

As shown in Figure 5-3, the *LD Gen* modules of systems m and n are set to initialize on clock cycles $y_0 - 1$ and $x_0 - 1$ respectively. On system m, for example, $i = y_0 - 1$ would normally be -1, so *LD Gen M* causes the *R Gen* module to initialize (i.e., set R to 0) on clock edge $M - 1$. This then tells system m to use register 0 when transmitting on edge m_0. The *LD Gen* module can also be used for reaction latency compensation by adjusting x_0 and y_0 to reflect the necessary time advance.

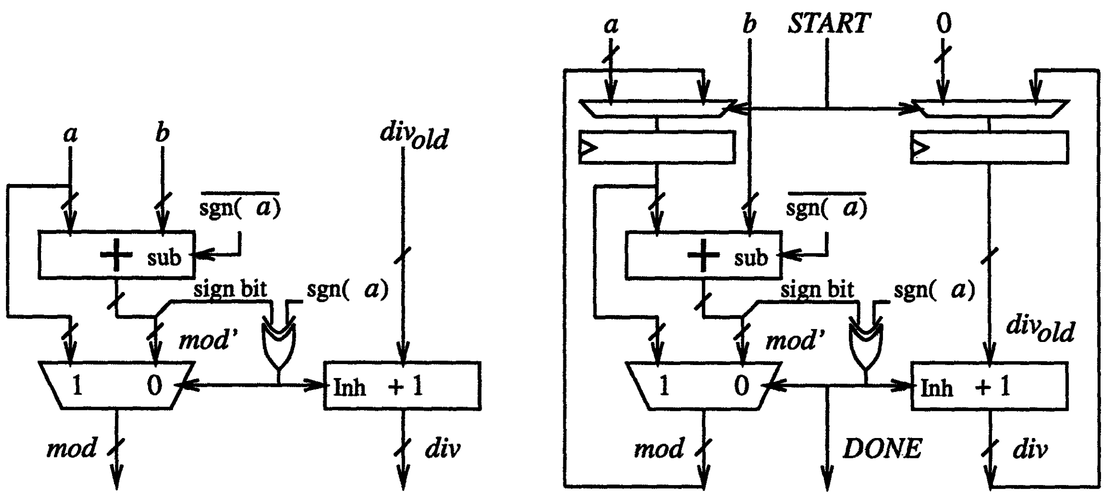


Figure 5-7: Computing mod and div. (a) The *MOD* module. (b) A sequential circuit.

5.2.4 Computing Initial Values

The scheduling hardware expects the values e_0, $(y_0 - 2) \bmod M$, and $(x_0 - 2) \bmod N$ to be available as constants. In the simplest implementation, these constants can be taken as external inputs which the system designer must compute manually for a certain configuration. It is not difficult, however, to design a more user-friendly circuit that takes M, N, and the timing parameters as inputs and uses these to computes the required constants in hardware.

The main problem in computing these constants is that of performing the mod and div operations. These operations can be done with the combinational *MOD* module shown in Figure 5-7(a). This module subtracts or adds b from or to a in an attempt to change a's sign (i.e., it subtracts if a is nonnegative, and adds if a is negative). If the sign changes, it sets the *mod* output to the value of a before the subtraction or addition, and sets *div* to div_{old}. Otherwise, it passes the new value *mod'*, to *mod*, and sets *div* to $div_{old} + 1$. Assuming that $-2b < a < 2b$, the *mod* output would be equal to

$$\begin{array}{ll} (a \bmod b), & \text{if } a \geq 0 \\ (a \bmod b) - b, & \text{if } a < 0 \end{array} \qquad (5.9)$$

and the *div* output would be

$$\begin{aligned} &(a \text{ div } b), && \text{if } a \geq 0 \\ -&(a \text{ div } b), && \text{if } (a < 0) \text{ and } (a \bmod b \neq 0) \\ -&(a \text{ div } b) - 1, && \text{if } (a < 0) \text{ and } (a \bmod b = 0) \end{aligned} \tag{5.10}$$

These outputs can then be converted into $a \bmod b$, and a div b using appropriate circuitry.

The sequential circuit shown in Figure 5-7(b), which uses registered feedback paths from the *MOD* module's *mod* and *div* outputs to its (original) a and div_{old} inputs respectively, allows us to perform these operations for a wider range of inputs. After the registers are initialized by the *START* signal, this circuit proceeds to subtract b from (or add it to) the register's contents until the sign of the adder's output, *mod'*, changes sign. Then, the *DONE* line goes high, causing the register's contents to be fed back unchanged into the register, and holding the register's contents constant at its last value before the sign change. This holds the *DONE* line high indefinitely, and holds the *mod* and *div* outputs constant at values given by Equations 5.9 and 5.10 respectively.

If there is a known limit to how large a can be with respect to b, then it is also possible to unroll the loop and make a *combinational* circuit by cascading several *MOD* modules. A cascade of k *MOD* modules would work for any integer a in the range $-(k+1)b < a < (k+1)b$. If $x_0 = 1$, for example, then we would have $-2N \leq f_0 \leq 2N$ for both slower transmitter and slower receiver cases, and would need a cascade of *two* *MOD* modules with initial inputs $a = f_0$ and $b = N$.

Such a cascade is shown in Figure 5-8, together with additional circuitry that derives e_0 and y_0 from *mod* and *div*. In this circuit, the two *MOD* modules produce final *mod* and *div* outputs given by Equations 5.9 and 5.10 (with $a = f_0$ and $b = N$), and the circuitry in the dashed box computes e_0 and y_0 from *mod* and *div* depending on the original sign of f_0. If f_0 is nonnegative, the circuit computes e_0 as $mod - N$, and computes y_0 by forming a 1 at the left input of the adder and adding it to *div* to get $(f_0 \text{ div } N) + 1$. If f_0 is negative, the circuit simply uses *mod* for e_0, and computes y_0 by subtracting *div* from zero to get $(f_0 \text{ div } N) + 1$ if $(f_0 \bmod N = 0)$, and $(f_0 \text{ div } N)$ otherwise.

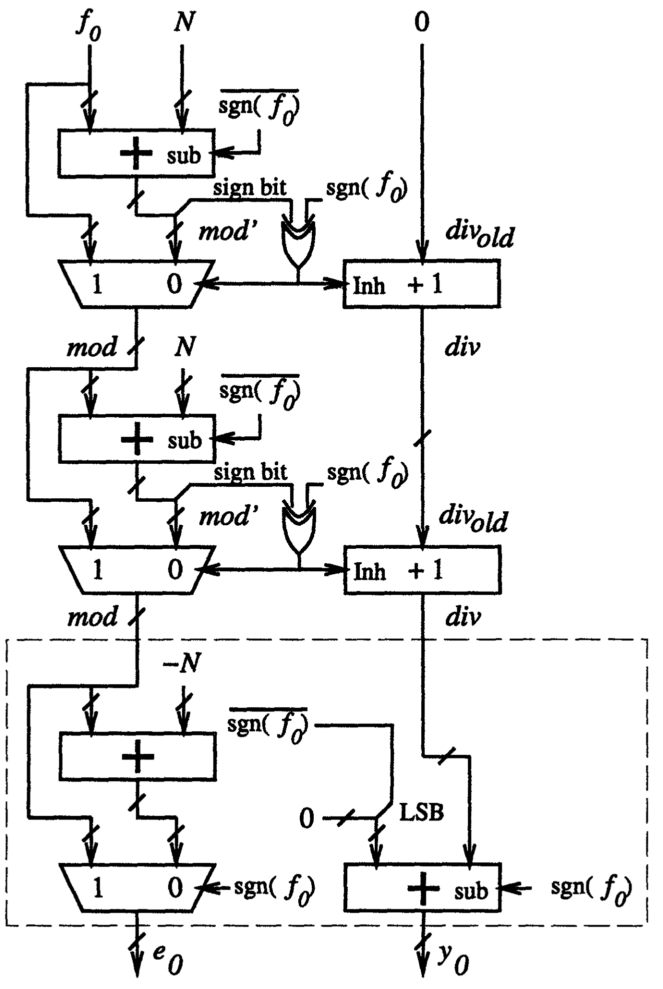


Figure 5-8: Combinational circuit for computing y_0 and e_0.

So far, the constants needed by the run-time scheduling hardware can all be computed combinationally. In section 5.3, however, the computation of some constants involve div and mod operations where the dividend (a) is not bounded with respect to the divisor (b). In these cases, the constants must be computed at bootup with a sequential circuit, and any circuitry that depends on the values of these constants must wait until the *DONE* signal goes high before using them.

5.2.5 Performance Issues

The run-time scheduling hardware is remarkably space-efficient. It requires only $O(\lg N)$ bits of memory instead of the $O(N^7)$ required by the table-based hardware to handle different frequency ratios and timing parameters. This makes the run-time scheduler very flexible, and makes it ideal for designing an "off-the-shelf" rational clocking chip, or for integrating rational clocking capability directly into the application chips themselves.

One drawback of the run-time scheduler, however, is its speed, which is limited by the delay required to compute the next value of e_{old} in the *E Gen* module. Though usually small enough not to matter in typical applications, this delay may be critical in very high performance systems. In such systems, the best solution is probably to have fast RAM-based tables that are loaded at bootup by run-time scheduling hardware clocked at a sufficiently small fraction of the desired frequency.

5.3 Generalized Run-Time Scheduling

The basic algorithm makes two assumptions. First, it assumes that communication is contamination-free so transmissions can be scheduled independently of each other. Then, it assumes that two buffers are sufficient to guarantee this first assumption.

The second assumption is made only for convenience and simplicity. The basic algorithm can easily be made to work with applications requiring more buffers by assigning more bits to R and having its value cycle through the available buffers instead of just being toggled between 1 and 0. In hardware, this is done by replacing the T flip-flop in the *R Gen* module with a mod-r counter, where r is the number of buffers available.

The first assumption, on the other hand, is a necessity, and changing the algorithm to make it work even when this assumption is not valid (e.g., as with single-buffered hardware) is not as easy. To avoid contamination, the algorithm must be modified to keep track of the amount of time that has passed since the last data transfer, and make sure that data is only transferred again after enough time has passed. Furthermore, the slower system would now also need scheduling hardware since it cannot just transmit on every cycle anymore.

In this section, we develop a *general* algorithm that detects contamination and, depending on the available resources, either disables data transfer, or uses a second register to avoid contamination. With this algorithm, it becomes possible to develop a single circuit that can be used for scheduling both single-buffered and double-buffered communication.

5.3.1 The Algorithm (Slower Transmitter Case)

As mentioned in section 5.1.1, the error term e at some x indicates how early or late system n would be if it tries to receive data from system m on clock edge n_x. In the basic algorithm, we were only concerned about prohibiting system n from receiving when it was too early, and assumed that receiving late would not pose a problem. This assumption, however, is only valid if contamination-free communication is guaranteed. Otherwise, it is possible for system n to be *too* late such that it would not be able to finish latching the data by the time the next transmission from system m contaminates the receive register's input.

As discussed in section 4.1, such a contamination problem would occur if the condition

$$t_{pc} \geq t_{ps} + t_{shn} \tag{5.11}$$

(from Inequality 4.2) is violated. For the single buffered case, $t_{pc} = T_M - t_{cpm}$, and this inequality can be restated as

$$t_{ps} \leq T_M - (t_{cpm} + t_{shn}) \tag{5.12}$$

Recalling from section 5.1.1 that the error term e is the time difference between edge S_x and edge P_y, we can replace t_{ps} with e, and express this inequality in units of Δt as

$$e \leq N - (CP_m + SH_n) \tag{5.13}$$

where $CP_m = (P_m - C_m)$ and $SH_n = (S_n + H_n)$. Thus, potential contamination problems can be detected at each x by checking if e is greater than the maximum safe value, e_{max}, equal to

$$e_{max} = N - (CP_m + SH_n) \tag{5.14}$$

Figure 5-9 shows two examples from the case being scheduled in Figure 5-1. Here, the schedule diagram has been marked in units of Δt to make the lengths of the time intervals easier to read. As shown in this figure, there is no contamination problem involving the

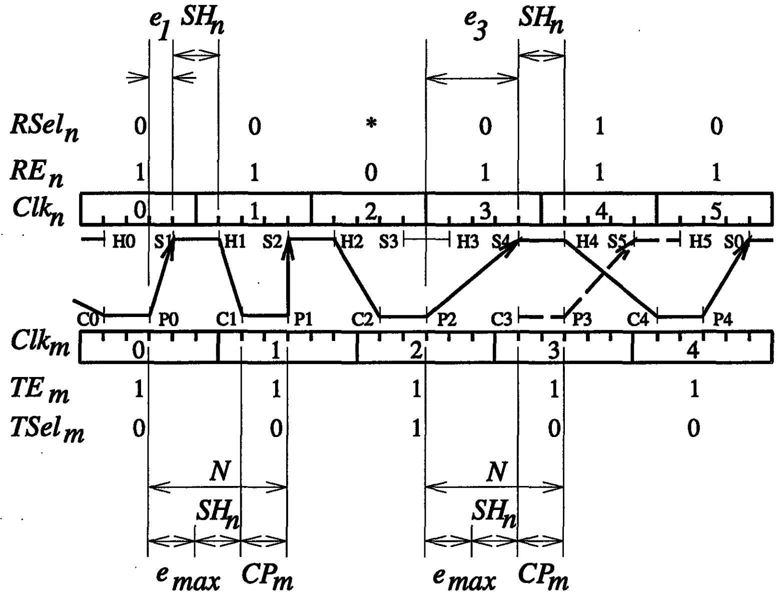


Figure 5-9: Checking for contamination using e.

transmission from P_0 to S_1, since e_1 is less than e_{max}. The transmission from P_2 to S_4, however, can be contaminated by that from P_3 to S_5 since e_3 is greater than e_{max}.

By checking if $e > e_{max}$ each time we transmit data from some clock edge m_y, we can determine if transmitting on clock edge m_{y+1} would result in contamination, and disable transmission on m_{y+1} accordingly. Alternatively, instead of simply disabling the offending transmission, we can set $RSel$ and $TSel$ so that the transmission on edge m_{y+1} uses a second register. This produces a schedule that can be used by *both* single-buffered and double-buffered systems. A double-buffered system would simply use the control signals as they are, while a single-buffered system would disable any transmission using the second register. In Figure 5-9, for example, a double-buffered system would use register 1 to transmit from P_3 to S_5 to avoid contamination. A single-buffered system would simply disable this transmission, and get the same schedule shown in the example in Figure 3-4.

Implementing these ideas into the basic algorithm gives us a set of two *general* algorithms: one for the faster system, and another for the slower system.

Algorithm for the Faster System

The general algorithm for the faster system can be described as follows:

```
1.  Initialize all entries of $TSel_m$, $TE_m$, $RSel_n$, and $RE_n$ to 0.
2.  $e_{max} = N - (CP_m + SH_n)$                ▷ get largest safe value of $e$
3.  $x = 1$                                       ▷ $x_0 = 1$
4.  $f_0 = \lfloor -(P_m + S_n) \rfloor$          ▷ get $f(t)$ at clock edge $x_0 - 1$
5.  $y = f_0$ div $N$                             ▷ $y_0$ = first line above $f_0$
6.  if $(f_0 \geq 0)$ or $(f_0 \bmod N = 0)$ then
        $y = y + 1$                               ▷ adjust $y$
7.  $e_{old} = (f_0 \bmod N) - N$                 ▷ $e_0$ = distance from line $y$ to $f_0$
8.  $R = 0$                                       ▷ use register 0 by default
9.  do $N$ times
10.     $e = e_{old} + M$                         ▷ compute error term for current $x$
11.     if $(e \geq 0)$ then                      ▷ is new data available?
12.         $TSel_m[(y-1) \bmod M] = R$           ▷ yes, transmit data using reg $R$
13.         $TE_m[(y-1) \bmod M] = (R < regs)$    ▷ but, only if reg R exists
14.         $RSel_n[(x-1) \bmod N] = R$
15.         $RE_n[(x-1) \bmod N] = (R < regs)$
16.         if $(R = 0)$ then                     ▷ if we're using reg 0,
17.             $R = \overline{(e \leq e_{max})}$ ▷    use reg 1 if necessary
18.         else                                  ▷ if we're using reg 1,
19.             $R = 0$                           ▷    use reg 0 next time
20.         $y = y + 1$                           ▷ wait for next system $m$ clock edge
21.         $e = e - N$                           ▷ make $e$ relative to new $y$
22.     $x = x + 1$                               ▷ get next $x$
23.     $e_{old} = e$                             ▷ get next $e_{old}$
```

This algorithm uses register 0 by default, and only uses register 1 if using register 0 would lead to contamination. Every time register 0 is used to transmit data, the current e is compared with e_{max} to check for contamination. If $e > e_{max}$, then contamination is avoided by setting R to 1 to let the *next* transmission use register 1. Otherwise, R is set to 0 so that register 0 is used again. Every time register 1 is used, R is simply set back to 0 — whether or not $e > e_{max}$. Checking e is unnecessary in this case because assuming double-buffering to be sufficient (i.e., assuming Inequality 4.9), we know from Inequality 4.6 that register 0 can always be safely reused after two clock cycles. The value of *regs* in lines 13 and 15 indicates the number of registers available and allows the algorithm to be used for both double-buffered and single-buffered systems. When $regs = 1$, lines 13 and 15 only enable data transfers that use register 0, and disable data transfers using register 1.

Given the same parameters, this generalized slower transmitter algorithm and the original graphical algorithm in section 3.3 generate exactly the same schedule on a single-buffered system. To see this, first note that since both algorithms match propagation edges with the *nearest* setup edge, any arrows they draw (i.e., transmissions they allow) are all chosen from the same set of P to S pairs. That is, if S_j is the nearest setup edge at or after P_i, then a transmission from clock edge m_i, if any is allowed at all, would always be matched with a reception on clock cycle n_j, regardless of which of the two algorithms is used. Note further that given a successful transmission from clock edge m_i to n_j, the original algorithm disables transmission on the next clock edge, m_{i+1}, if and only if the hold edge H_j is past the next contamination edge C_{i+1}, while the generalized algorithm disables transmission on clock edge m_{i+1} if and only if $e > e_{max}$ during cycle m_i. Since these two conditions (i.e., $e > e_{max}$, and H_j being past C_{i+1}) are equivalent, we can conclude that both algorithms skip exactly the same transmissions and produce the same schedule.

Algorithm for the Slower System

The slower system can detect and avoid contamination in the same way as the faster system, but needs to keep track of e in a different way. The following algorithm, written from the point-of-view of the slower system, m, shows how this is done:

```
1.  Initialize all variables and compute constants as before
2.  $\Delta NM = N \bmod M$                ▷ $\Delta NM$ = rate of change of $e$
3.  $e = e_0 \bmod M$                      ▷ get $e$ of first transmission
4.  $R = 0$                                ▷ use register 0 first
```

```
5.  do $M$ times
6.      $TSel_m[(y-1) \bmod M] = R$            ▷ yes, transmit data using reg $R$
7.      $TE_m[(y-1) \bmod M] = (R < regs)$     ▷ but, only if reg R exists
8.      if $(R = 0)$ then                      ▷ if we're using reg 0,
9.          $R = \overline{(e \leq e_{max})}$  ▷     use reg 1 if necessary
10.     else                                   ▷ if we're using reg 1,
11.         $R = 0$                            ▷     use reg 0 next time
12.     $e = e - \Delta NM$                    ▷ compute next error $(e-N) \bmod M$
13.     if $(e < 0)$ then                      ▷     by computing
            $e = e + M$                        ▷     $(e - \Delta NM) \bmod M$
14.     $y = y + 1$                            ▷ consider next system $m$ clock edge
```

In this algorithm, each iteration in the loop moves us from one horizontal line (system m clock edge) y to the next, and considers the times when e becomes nonnegative and the condition in line 11 of the faster system's algorithm is satisfied.

The initial value of e in this algorithm represents the first nonnegative value that results from repeatedly adding M to the negative number e_0, and can be computed as $e_0 \bmod M$ as done in line 3. Similarly, the value of e at the next iteration represents the next nonnegative value of e after repeated addition of M to $(e-N)$, and so can be computed as $(e-N) \bmod M$.

This latter modulo operation can further be simplified if the condition $0 \leq e < M$ is maintained through the course of the algorithm. Then, $(e - N) \bmod M$ can be expressed as $(e - \Delta NM) \bmod M$, where $\Delta NM = N \bmod M$, and can be computed by the two-step process in lines 12 and 13. This simplification is necessary when implementing the algorithm in hardware since the size of $(e - N)$ relative to M is not restricted, and so prevents the mod operation from being done combinationally using the techniques of section 5.2.4. Unrestricted modulo operations are still necessary for computing the constants ΔNM and $(e_0 \bmod M)$, but these are one-time computations that can be done off-line by the user, or computed iteratively at bootup using a sequential modulo circuit.

Non-Integer Timing Parameters

The generalized algorithms shown assume that the timing parameters are all integer multiples of Δt. If they are not, then they are rounded *conservatively* to the nearest integer — i.e., S, H, and P are rounded up, while C is rounded down. Unfortunately, rounding the parameters in this way can result in lower throughput efficiency, since doing so widens the timing windows, decreasing e_{max} and increasing e_{min}. One solution would be to use a sufficiently small fraction of Δt as the unit of time instead of Δt. For example, we can use units of $\Delta t/4$ by making the components two bits wider, and by replacing N and M in the algorithm by $4N$ and $4M$ respectively. Another potential solution is to first compute e_{max} and e_{min} using unrounded parameters, adjust their values appropriately depending on the fractional part of the initial value of e, and round the result conservatively to the nearest integer. The details of this latter approach still need to be worked out, but the approach seems plausible and not exceedingly complex.

5.3.2 The Hardware

The hardware for the two algorithms are shown in Figures 5-10 and 5-11. Like the original *E Gen* module on which they are based, they are straightforward implementations of their respective algorithms. In both systems, the T flip-flop in the *R Gen* module has been replaced by a D flip-flop with enable (E) and clear (Clr) controls. In this configuration, the new value of R is computed according to the algorithm, and latched at the next clock edge every time e becomes nonnegative. The input signal SB is used to indicate that the target application is single-buffered. When SB is set to 1, data transfers attempting to use register 1 are disabled by setting RE and TE to 0.

Note that there are only three structural differences between the two circuits: the source of e, the source of the E input of the D flip-flop, and the use of an inverter on the control for the last multiplexer. Thus, by adding a multiplexer and a few appropriate gates, we can come up with a single circuit that can be used by either system. The only difference between the systems would then be in the input values fed into the circuit.

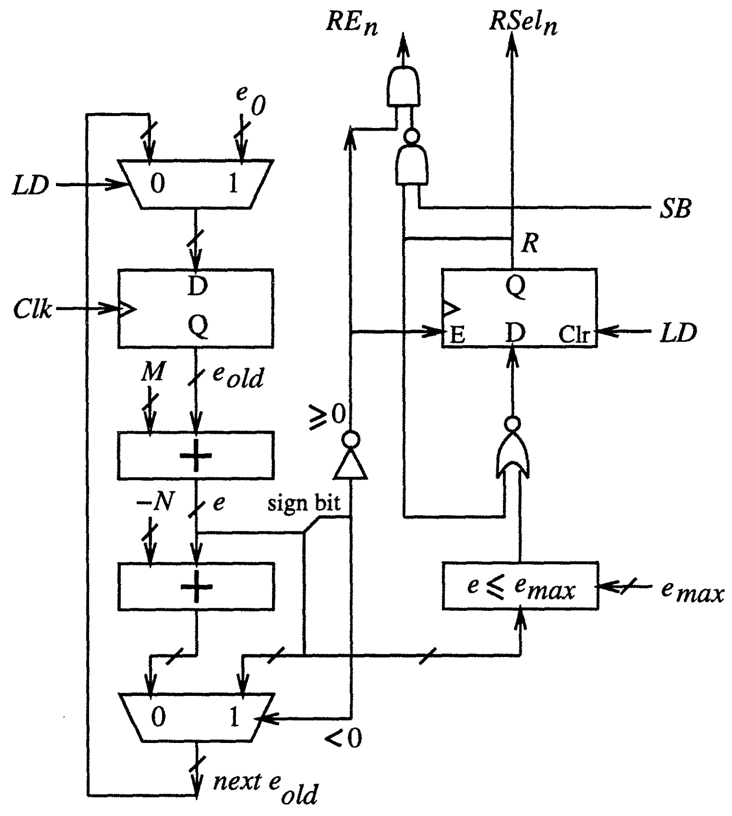


Figure 5-10: Scheduling hardware for the faster system.

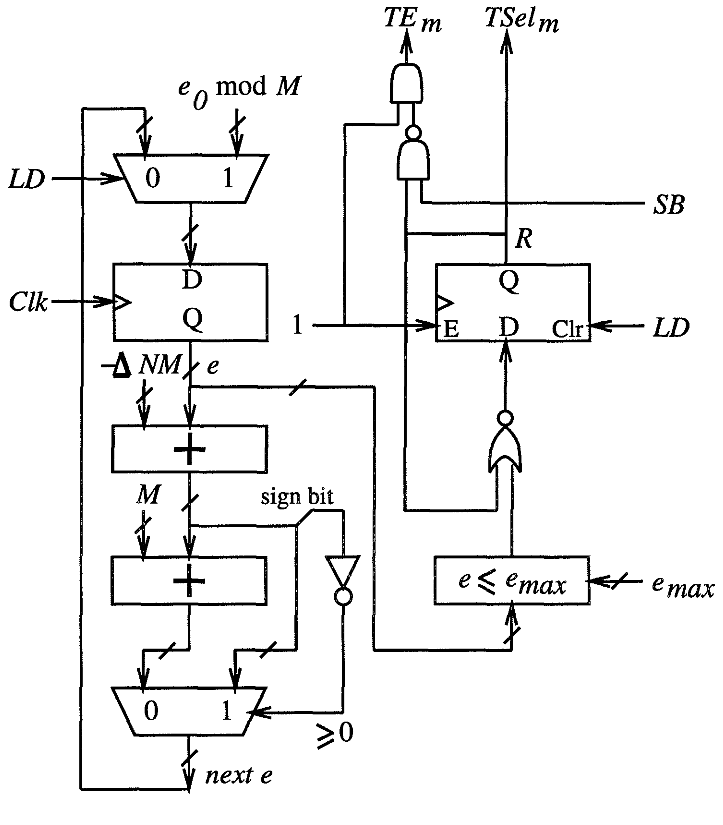


Figure 5-11: Scheduling hardware for the slower system.

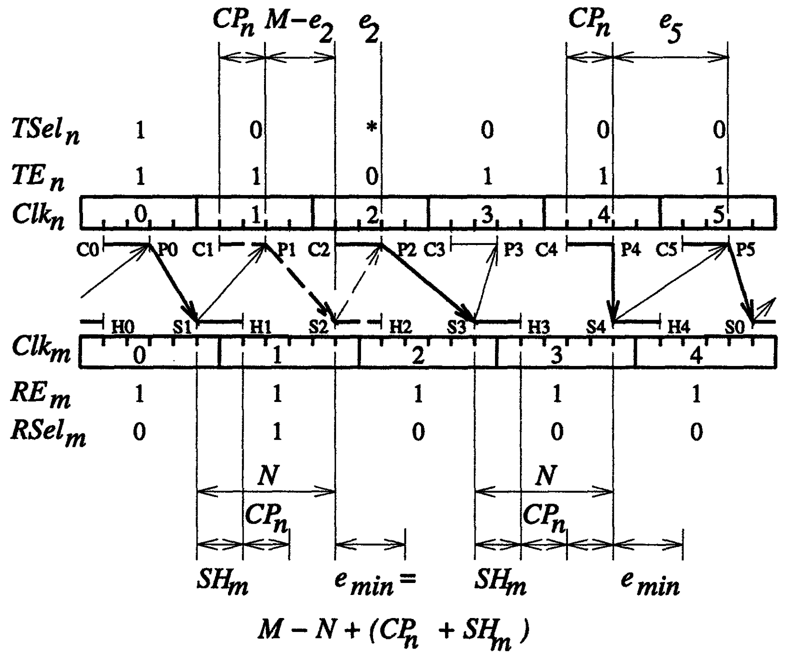


Figure 5-12: Checking for contamination in the slower receiver case.

5.3.3 Slower Receiver Case

The Algorithms

The general algorithms for the slower receiver case are based on the greedy slower receiver algorithm. Section 5.1.3 discusses how the greedy slower receiver algorithm can be implemented. To generalize this algorithm, we first apply the changes discussed in that section to the general algorithms of section 5.3.1. We then modify the slower system's algorithm by changing the condition in line 13 to $(e \leq 0)$, and adding M to the initial e after line 3 if $(e_0 \bmod M)$ is equal to zero. These changes are necessary since changing the condition in line 11 of the faster system's algorithm to $(e > 0)$ implies that e must be positive (not 0) during the course of the slower system's algorithm.

Figure 5-12 shows how contamination is detected and avoided in the slower receiver case. Here, the meaning of the error term e has changed. It is now $M - e$ which indicates how early or late system n would be if it tries to transmit data to be received at clock edge m_y. Correspondingly, the contamination condition is now

$$M - e \leq N - (CP_n + SH_m) \qquad (5.15)$$

This indicates when system n is transmitting data too *early* such that it would contaminate data being received by system m during the *previous* cycle, m_{y-1}. To detect contamination, therefore, we check if e is less than the minimum safe value, e_{min}, equal to

$$e_{min} = M - N + (CP_n + SH_m) \tag{5.16}$$

In Figure 5-12, for example, there is no contamination problem involving the transmission from P_4 to S_4, where $M - e_5 = 0$, since e_5 is greater than e_{min}. The transmission from P_1 to S_2, however, can contaminate that from P_0 to S_1 since e_2 is greater than e_{min}.[3]

When contamination is detected, it can be avoided by disabling the *current* data transfer, or using a second register for it as done in the slower transmitter algorithms. Since the original algorithms disable the *next* data transfer, however, we must modify them by moving the calculations for R ahead of the control signal assignments. Then, to ensure that the first data transfer is done using register 0, we initialize R to 1 instead of 0. In hardware, these changes mean taking $RSel_n$ from the output of the NOR gate instead of the D flip-flop, and having LD set the flip-flop instead of clear it.

Unlike the slower transmitter algorithm described in 5.3.1, this slower receiver algorithm does *not* necessarily produce the same single-buffered schedules as the algorithm of section 3.3. As discussed in section 4.2.2, the greedy slower receiver algorithm, on which this generalized algorithm is based, lets the transmitter transmit data at the *latest* possible time before a setup edge. The algorithm of section 3.3, on the other hand, lets the transmitter transmit at the *earliest* possible time after the preceding hold edge.

Fortunately, this difference in scheduling strategy does not affect throughput efficiency — like the algorithm of section 3.3, the generalized slower receiver algorithm is optimally efficient. This is because it considers all possible occasions for transferring data (i.e., it goes through all of the slower system's setup edges one-by-one without skipping), and only disables a data transfer if transmitting at the *latest* possible time results contamination. It may be possible to transmit data at an earlier time without causing contamination, but if

[3]It is interesting to note that if e_{min} is negative, then contamination cannot occur. Thus, a sufficient (but not necessary) condition for guaranteeing 100% throughput efficiency on a *single*-buffered system is: $N - M > CP_n + SH_m$

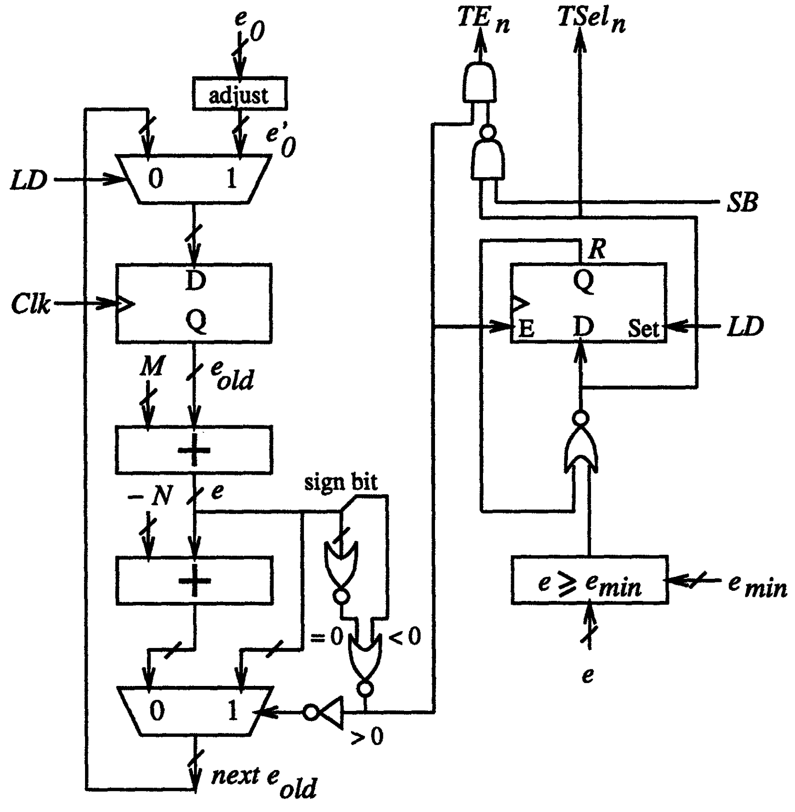


Figure 5-13: Faster system's hardware for the greedy slower receiver algorithm.

transmitting at the latest possible time causes contamination, then transmitting any earlier will also cause contamination.

The Hardware

Figures 5-13 and 5-14 show the hardware for general run-time scheduling in the slower receiver case. As shown, the circuitry that controls the multiplexer which selects the next e_{old} has been modified to implement the necessary transformations. Similarly, the e_0 and ($e_0 \bmod M$) inputs to the top multiplexer of both circuits are first passed through *adjust* modules to adjust their values as required by the transformations. These adjust modules are slightly different from each other, and are shown in Figure 5-15. Note that the slower system's adjust module takes $e'_0 \bmod M$ as input, where e'_0 is the output of the faster system's adjust module.

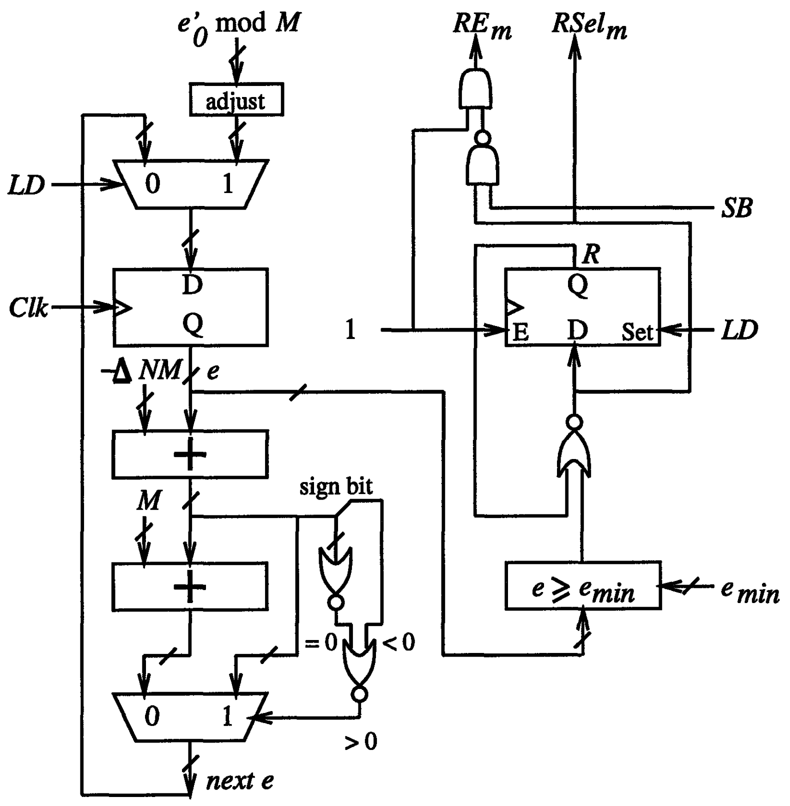


Figure 5-14: Slower system's hardware for the greedy slower receiver algorithm.

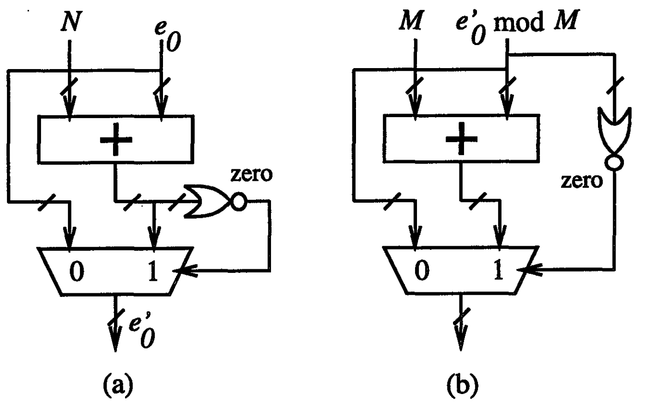


Figure 5-15: Adjusting initial errors. (a) For the faster system. (b) For the slower system.

As shown, the two circuits are still sufficiently similar to each other and to those for the slower transmitter case that it is still possible to come up with a single circuit that can be used for any of the four roles by simply feeding it the right inputs.

5.3.4 Choosing P_{begin}

By initializing R to 0 in the general algorithms presented, we guarantee that the first data transfer will not be disabled. This is equivalent to choosing a specific P_{begin} in the scheduling algorithm of section 3.3. For the slower transmitter case, $P_{begin} = P_{y_0}$, while for the slower receiver case $P_{begin} = P_{x_0} = P_0$.

While choosing P_{begin} in this way works in many cases, there are some potential problems. One problem, as mentioned in section 3.3, is that the choice of P_{begin} can affect throughput efficiency. Thus, it may occasionally be better to choose a different P_{begin}. This can be done by changing x_0 to achieve the desired starting point, and then applying the techniques discussed in 5.1.2. This can also be done with less hardware by precomputing the schedule for a desired P_{begin}, and initializing R to the value it would have during cycle m_{y_0}, for the slower transmitter case, and cycle n_{x_0}, for the slower receiver case.

A worse problem is that forcing register 0 to be used for the first data transfer may cause contamination problems with the last data transfer in the coincidence cycle. This would happen if the last transfer incorrectly assumes that it can safely use register 0. Ideally, this problem can be avoided by initializing R once, and then never again forcing a particular data transfer to use register 0. However, this approach is not reliable since the communication schedule can easily be broken if one of the flip-flops flips erroneously due to noise. A more practical solution is again to initialize R to a value that is known not to cause this kind of contamination problem.

Although initializing R to an appropriate value solves these problems, it requires precomputation and defeats the purpose of run-time scheduling. One way to avoid precomputation is to make hardware that detects, before initializing R, if a transmission on first cycle would contaminate a transmission from the last cycle, and initializes R to 1 if contamination is possible, so that the transmission from the first cycle uses register 1 instead of register 0.

There are two related problems with this approach, however. First, if single-buffering is being used, then throughput efficiency would become sub-optimal in the case where the transmission from the last cycle ends up being disabled, since the transmission from the first cycle would then be disabled unnecessarily. Second, if double-buffering is being used, then it is possible that the algorithm could incorrectly assume that the transmission from the last cycle can safely use register 1, and thus cause contamination problems with the transmission from the first cycle. That is, we could just get the original problem back in a different form.

This problem of initializing R to an appropriate value has not yet been solved, and should be a subject of future research. Meanwhile, the solution just described can be used for single-buffering if getting guaranteed optimal performance is not a requirement. For double-buffering, we can simply use the basic algorithm, which has none of these problems.

5.3.5 Generalizing to More Buffers

The algorithms and hardware presented in this section can be extended to work with more buffers by providing some means to keep track of which buffers are available for use at a certain time. Using these means, the algorithm would schedule communications on the lowest numbered buffer available at any time.

Unfortunately, while this idea is conceptually simple, its implementation is disproportionately complex. In the double-buffering case presented, a single variable, R, was sufficient to keep track of buffer usage because we assumed that a buffer cannot be in use for more than two cycles. Thus, if one register is used to transmit data on some clock edge m_i, then we know that the other register can be used to transmit data on edge m_{i+1} regardless of which register was used to transmit on edge m_{i-1}. In triple-buffering, there are now three possible buffers, and each buffer can be in use for three cycles. A general algorithm for this case would require considerably more variables, more comparisons, and more computations than before. This disproportionate complexity quickly offsets any advantages gained from the convenience of having a single generalized circuit. Since only a small fraction of applications are expected to require three buffers, it is probably more practical to just use the two-buffer-capable generalized circuit most of the time, and then use the basic circuit with triple-buffering when necessary.

Chapter 6

Simulation and Implementation

The techniques and ideas presented in this thesis were developed and tested with the help of different software and hardware tools. First, communication scheduling software was written to generate lookup tables for different frequency ratios, timing parameters, and scheduling algorithms. This software was used to measure throughput and to test the different algorithms. Then, a CMOS VLSI integrated circuit using programmable lookup tables was designed and fabricated to facilitate actual testing of the rational clocking technique. This chip was used in a breadboard-based circuit that verified and demonstrated the usefulness of the rational clocking techniques and algorithms. Finally, a model of the run-time scheduling was constructed and simulated using Verilog to verify the correctness of the proposed hardware.

This chapter provides functional descriptions of these tools, and discusses various issues that came up during their development and use. Further details about their implementation, including source code and a VLSI layout, are provided in the Appendices.

6.1 Communication Scheduling Software

The communication scheduling software is a text-based menu-driven program written in C that takes timing parameters and frequency ratios as inputs, and produces corresponding data transfer schedules. It can be used for studying scheduling algorithms, and for generating source files for the lookup table ROMs. This section provides a functional description of the different features of the software, and presents some recommendations for improvements. A complete listing of the source code and more detailed discussion about different issues involving the design of the software are available in Appendix B.

Figure 6-1 shows the scheduling software's menu tree. At the top level, the user can either enter the timing parameters, generate output, or change the program options. The meaning of the timing parameters in the first case, and the format of the output generated in the second case, are determined by the options available under the Options Menu. These options are discussed in the following subsections.

6.1.1 Timing Parameter Units

In the scheduling software, time is expressed in integer units of *dt*, defined as:

$$dt = \Delta t/d \tag{6.1}$$

where

$$\Delta t = T_{MN}/MN \tag{6.2}$$

as defined in Chapter 5, and *d* is some integer specified by the user. Using these units and a sufficiently large value for *d*, we can express time with arbitrarily fine precision without the use of floating-point numbers (assuming *d* does not get so large that the times we want to express do not fit in a 32-bit "long" integer). As discussed further in Appendix B, this not only allows us to avoid rounding-errors associated with floating-point numbers, but also makes it easier to implement the scheduling algorithms.

As Figure 6-1 shows, the software allows the user to express the timing parameters in five different ways. The first two ways (dt and Δt) are self-explanatory. The third ($1/dt$) allows the user to express the timing parameters as fractions of a clock period. For example, if system m's propagation delay, P_m, is 20% of the minimum clock period, and d is equal to its default value of 1000, then the user should enter 200 when asked for P_m. As discussed in section 3.5.1, expressing the timing parameters in this way is useful if we

Main Menu

- Enter Parameters
- Generate Output
- Options Menu

Options Menu

- Parameter Units
  - dt
  - Δt
  - $1/d$ of a clock period
  - Absolute Units with Clock Division
  - Absolute Units with Clock Multiplication
- Scheduling Algorithm
  - Original Single-Buffered
  - Generalized Run-Time (2 buffers)
  - Basic Run-Time Double-Buffered (Greedy)
  - Basic Run-Time Double-Buffered (Lazy)
- Compute Mode
  - Pair
  - Table
- Output Format
  - Text
  - LaTeX schedule diagram
  - LaTeX schedule plot (not implemented)
  - ROM
- Output Options
  - Log file name
  - Show intermediate results?
  - Draw text schedule diagram?
  - Use with prototype chip?

Figure 6-1: The communication scheduling software's menu options.

assume that rational clocking will be used with systems whose timing parameters scale with their minimum clock periods.

The last two ways to express timing parameters, are useful when the timing parameters are fixed, or *absolute*, and do *not* scale with the clock period. The fourth option assumes that clock division is being used, and asks the user for integers representing the timing parameters and the generating clock's period, T_{high}, in some base unit of time (e.g., picoseconds, nanoseconds, etc.). The fifth assumes that clock multiplication is used, and asks the user for T_M instead of T_{high}, where T_M is assumed to be the generating clock's period (i.e., we assume that Clk_n is generated from Clk_m so that T_M remains fixed while T_N varies). These two ways of expressing the timing parameters are useful in situations where the same hardware is to be used at different frequencies. In programming the ROM used in the demonstration circuit of section 6.3, for example, the timing parameters were specified using absolute units with clock division.

6.1.2 Scheduling Algorithms

Four scheduling algorithms were implemented in the program: the original greedy algorithm for single-buffering, the greedy and lazy versions of the run-time scheduling algorithm for double-buffering, and the generalized run-time scheduling algorithm. These algorithms were slightly modified to make them work better with the software's representation scheme for time. More details regarding these modifications can be found in the discussion and source code contained in Appendix B.

6.1.3 Computation Modes

The scheduling software works in one of two computation modes: pair, and table. In pair mode, the software simply uses the currently selected algorithm to generate bidirectional schedules (i.e., two schedules, one for each direction) for the frequency ratio specified by the user through the variables M and N. In table mode, the software computes schedules for all frequency ratios with numerators and denominators ranging from 1 to $\max(M, N)$, and outputs the throughput efficiency in percent for each of these ratios in a two-dimensional table format. Table mode is also used to generate object code files for programming ROMs that work with different frequency ratios.

6.1.4 Output Formats and Options

The scheduling software can produce output in a number of formats, and has a few options available for each of these formats.

Text Output

The simplest output format is text, where the program just emits text to the standard output. This provides a quick and portable (i.e., it works on standard TTY terminals) way to look at the generated output. When using this format, the program also allows the user to keep a log of all generated text output in a separate file. This makes it possible for the user to do several computations in one run through the program, and then review the results at a later time by looking at the log file.

When using text format in pair computation mode, the program shows the schedules for both directions by listing the values of the control signals for each clock cycle, as shown in Figure 6-2. The user has the option of displaying these control signals in a simple character string format as in Figure 6-2(a), or in a formatted text version of a schedule diagram as in Figure 6-2(b). The latter gives a better sense of the timing relationship between the two systems, and is preferable to the former as long as M and N are not so large that the text schedule diagrams become wider than the screen. The program also reports the throughput efficiency in each of the two directions, although this is not shown in Figure 6-2.

Since we cannot draw arrows to explicitly indicate data transfers, we instead show the values of two data lines in the circuit, TD (transmitted data) and RQ (received data). The value of TD during a certain cycle is the value presented by the transmitter at the input of the transmit register during that cycle. If TE is also 1 during that cycle, then TD gets latched into the transmit register at the *next* clock edge, and gets transmitted to the receiver. The value of RQ during a certain cycle is the value at the output of the receive

```
M -> N
-------
N.RE    : 110101
M.TE    : 11011
-------
N.Rsel  : 00x0x0
M.Tsel  : 00x00
-------
N.RQ    : 301x2x
M.TD    : 12x30

N -> M
-------
N.TE    : 010111
M.RE    : 10111
-------
N.Tsel  : x0x000
M.Rsel  : 0x000
-------
N.TD    : x1x230.
M.RQ    : 30x12
-------
```

(a)

```
M -> N
-------
N.RE    : | 1 | 1 | 0 | 1 | 0 | 1 |
M.TE    : |  1  |  1  |  0  |  1  |  1  |
-------
N.Rsel  : | 0 | 0 | x | 0 | x | 0 |
M.Tsel  : |  0  |  0  |  x  |  0  |  0  |
-------
N.RQ    : | 3 | 0 | 1 | x | 2 | x |
M.TD    : |  1  |  2  |  x  |  3  |  0  |

N -> M
-------
N.TE    : | 0 | 1 | 0 | 1 | 1 | 1 |
M.RE    : |  1  |  0  |  1  |  1  |  1  |
-------
N.Tsel  : | x | 0 | x | 0 | 0 | 0 |
M.Rsel  : |  0  |  x  |  0  |  0  |  0  |
-------
N.TD    : | x | 1 | x | 2 | 3 | 0 |
M.RQ    : |  3  |  0  |  x  |  1  |  2  |
-------
```

(b)

Figure 6-2: Sample text outputs in pair mode. (a) Without text schedule diagrams. (b) With text schedule diagrams.

```
Freq.   1     2     3     4     5     6
1       0     0     0     100   100   100
2       0     50    50    100   100   100
3       0     50    33    66    66    100
4       100   100   66    100   75    100
5       100   100   66    75    100   80
6       100   100   100   100   80    100
```

Figure 6-3: Sample text output in table mode.

register. If RE was 1 during the *previous* cycle, then RQ represents new data just latched into the transmit register at the most recent clock edge. As with determining RE and TE, we use modulo arithmetic when determining which clock cycle receives the transmitted data. That is, the data transfers "wrap-around" the right edge of the diagram.

When using text format in table computation mode, the program generates a simple table of throughput efficiency values with M designating the rows and N designating the columns. The throughput efficiency reported is the throughput efficiency when transmitting data from system m to system n. Figure 6-3 shows an example where the timing parameters are defined as $S = 1$, $H = 1$, $C = 1$, and $P = 3$, in units of Δt.[1] Note that since units of Δt are being used here, the fraction of the clock cycle covered by a timing parameter actually becomes larger as M and N become smaller. That is why the throughput efficiency starts out low with small values of M and N and becomes higher as M and N increases. A similar effect can be seen when using absolute units with clock division.

In both pair or table computation modes, the user has the option of looking at the results of intermediate computations while the program is running. This is useful for debugging code, and also for just gaining a better understanding of how the algorithm works. When using this "peek mode", a log file can become useful since the intermediate computations usually generate a large amount of output.

LaTeX Output

The scheduling software also allows the user to generate files that can be formatted by the LaTeX document preparation system [29]. It can generate graphical schedule diagrams in pair computation mode, and formatted throughput efficiency tables in table computation mode.

In pair computation mode, the program generates a LaTeX file, *sched.tex*, which when compiled, produces schedule diagrams such as that in Figure 6-4.[2] Unfortunately, arrows

[1]These are the same parameters used in the original example in section 3.3, in the examples in Chapter 5, and in the example in Figure 6-2. These are *not* the same as the parameters used in Figures 3-5 and 4-1.

[2]The figure only shows one direction of data transfer. The actual *sched.tex* file contains a separate schedule diagram for each of the directions.

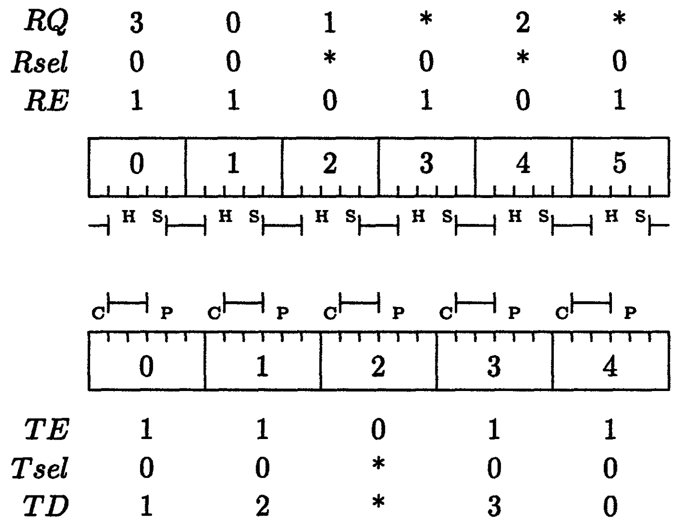


Figure 6-4: A LaTeX format schedule diagram.

cannot be drawn in the schedule diagram because LaTeX's `picture` environment, which is used to generate these diagrams, has no provision for drawing lines of arbitrary slope.

In table computation mode, the program generates a LaTeX `tabular` environment which can be inserted into LaTeX documents to produce tables such as those in Figures 3-5 and 4-1. Figure 6-5 shows another table generated using the timing parameters of the demonstration circuit discussed in section 6.3.

Since generating schedule diagrams in pair computation mode and generating text output can be done independently, the software does both. That is, it displays the results in text format on the standard output, and at the same time generates *sched.tex*. Similarly, when computing in table mode, the program uses text format for the standard output, while using LaTeX format for the output file. In this case, however, the output file is simply the log file, so the user may first have to clip the desired `tabular` environment from the log file and save it into a separate file before being able to use it.

One feature that was planned but not yet implemented is the ability to generate LaTeX format schedule plots such as the one in Figure 5-1. This feature can be implemented using the same techniques used to implement the LaTeX schedule diagrams are implemented.

Transmit	Receive Frequency															
Freq.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	0	50	50	100	100	100	100	100	100	100	100	100	100	100	100	100
3	100	50	100	66	100	100	100	100	100	100	100	100	100	100	100	100
4	100	100	66	100	75	100	100	100	100	100	100	100	100	100	100	100
5	100	100	100	75	100	80	100	100	100	100	100	100	100	100	100	100
6	100	100	100	100	80	100	83	100	100	100	100	100	100	100	100	100
7	100	100	100	100	100	83	100	85	100	100	100	100	100	100	100	100
8	100	100	100	100	100	100	85	100	87	100	100	100	100	100	100	100
9	100	100	100	100	100	100	100	87	100	88	100	100	100	100	100	100
10	100	100	100	100	100	100	100	100	88	100	90	100	100	100	100	100
11	100	100	100	100	100	100	100	100	100	90	100	90	100	100	100	100
12	100	100	100	100	100	100	100	100	100	100	90	100	91	100	100	100
13	100	100	100	100	100	100	100	100	100	100	100	91	100	92	100	100
14	100	100	100	100	100	100	100	100	100	100	100	100	92	100	92	100
15	100	100	100	100	100	100	100	100	100	100	100	100	100	92	100	93
16	100	100	100	100	100	100	100	100	100	100	100	100	100	100	93	100

Figure 6-5: LaTeX format efficiency table for the demonstration circuit's timing parameters.

ROM Output

The scheduling software provides the capability to generate output in the Intel MCS-86 16-bit Hexadecimal Object file record format [30]. To produce an object file for a "generic" ROM similar to that described in section 3.5.1, the user simply specifies table computation mode and ROM Output format together. This tells the program to output an object file containing not only the appropriate lookup table contents for different frequency ratios, but also those for different algorithms. In this way, the user can specify the scheduling algorithm using the most significant address bits of the ROM. The program also gives the user the option to generate a ROM file specifically for use with the prototype rational clocking chip. When this option is used, the program automatically remaps the ROM to account for certain idiosyncrasies of the rational clocking chip. More details on the address mapping of the ROM, as well as its output format, are provided in section 6.3.4.

When using ROM output format, the program writes the output to both the standard output and to the log file. The ROM format is a text file, not a binary file, so the user can easily edit the log file as necessary. Furthermore, the object file is generated in such a way that the tables produced by different algorithms are stored under separate *segments* [30]. This makes it easy for the user to select only desired algorithms when all of them together would not fit in the ROM.

6.1.5 Results and Recommendations

The communication scheduling software successfully generated correct schedules for different timing parameters, frequency ratios, and algorithms, and gave valuable insights that helped refine the scheduling algorithms. Furthermore, it was successfully used to generate a ROM for use with the prototype rational clocking chip in the demonstration circuit.

Despite its usefulness, however, the software can still benefit from several improvements. Improved graphical output, for example, would be very useful in helping users visualize the communication schedules. This can be achieved by using Postscript instead of the LaTeX `picture` environment. Automatic conversion between different timing parameter units would be another useful feature. Ideally, the user should just be able to enter numbers taken from a data book, and then have the software automatically perform such computations as finding appropriate values for d, and performing conservative rounding when necessary.

Perhaps the most useful feature to have at this point, however, would be the capability to check the correctness of the generated schedules. Right now, the program does not actually check that the schedule generated by an algorithm is error-free. Using the run-time double-buffered lazy algorithm for the slower receiver case, for example, frequently results in erroneous data transfers being done when the smaller of M and N is not even To detect these problems, we might write an independent function that takes a given schedule, attempts to perform data transfers according to it, and reports problems such as setup and hold time violations.

6.2 The Prototype Rational Clocking Chip

The prototype rational clocking chip is a full-custom CMOS VLSI circuit that implements the RAM-based scheduler described in section 3.5.3, and is designed to be flexible enough to be used in a number of experimental configurations. A block diagram of the prototype chip is shown in Figure 6-6. The chip contains three major components: the counters, the lookup tables, and the circuitry for loading the tables on bootup. This section gives a

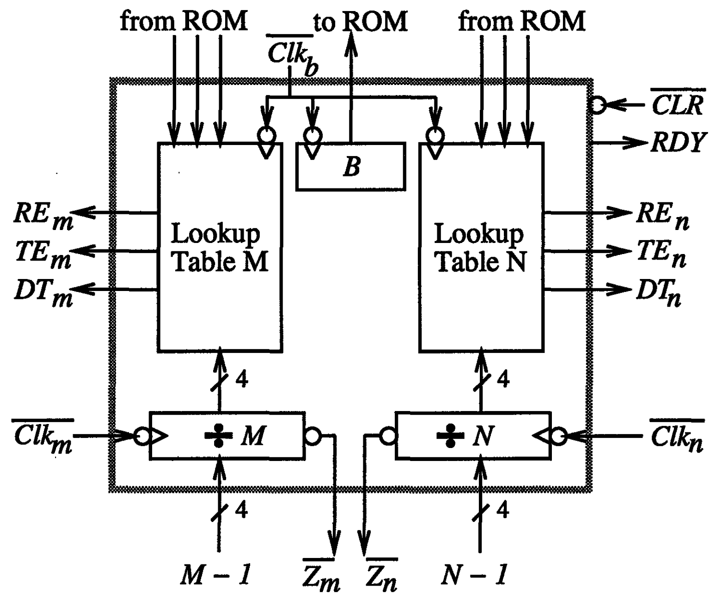


Figure 6-6: The prototype rational clocking chip.

functional description of these components, and presents the results of the fabrication and testing process. A layout of the chip can be found in Appendix C.

6.2.1 The Counters

Each counter in the chip is a 4-bit negative edge-triggered decrementing counter with a terminal count input, and a combinational active-low zero output, $\overline{Z}$, which goes low whenever the count value is zero. To use it as a divide-by-N counter, we set its terminal count to $N-1$. This makes the counter count down from $N-1$ to zero, and causes it to emit a one cycle pulse on $\overline{Z}$ once every N cycles. We can then implement clock multiplication by connecting the counters to an external PLL as in Figure 3-2. We can also use the chip with clock division by simply taking the clock inputs from an external clock division circuit like that in Figure 3-1, and ignoring the $\overline{Z}$ outputs. The external clock division circuit in turn can be implemented using another rational clocking chip.

The counters can be used to generate rationally related clocks with ratios where M and N range from 2 to 16. They cannot be used for ratios where either M or N is 1, however, since the $\overline{Z}$ output would then stay low and not pulse. This problem can be solved easily by either manually connecting the undivided clock directly to the appropriate system, or doubling both M and N.

6.2.2 The Lookup Tables

Within the chip, each counter is used to index a 16-entry, 3-bit wide lookup table. The tables are only 3 bits wide instead of the ideal 4 bits for double buffering (i.e., one bit each for TE, $TSel$, RE, and $RSel$) because the chip was designed before double-buffering was developed. Thus, it only has bits for TE and RE, plus an extra bit for a time-advanced copy of TE should reaction latency compensation be necessary. Fortunately, for experimentation purposes, we can easily get 4 bits by using two copies of the rational clocking chip.

The lookup tables are implemented as serial-in parallel-out (SIPO) shift registers that are loaded at bootup time with the appropriate entries by shifting-in values from a 3-bit input. With the appropriate interface, the lookup tables can be used with a serial ROM, a combinational ROM, a processor running scheduling software, or a run-time scheduling circuit.

6.2.3 The Bootup Control Circuitry

The chip has an asynchronous active-low reset line, $\overline{CLR}$, which clears the tables and counters, and signals the start of the bootup sequence. The chip waits until $\overline{CLR}$ goes high, then proceeds to load the lookup tables on the following 16 clock edges, shifting-in entry 15 first and entry 0 last. The lookup table outputs are invalid during these first 16 cycles, and should not be used. A ready signal, RDY, goes high after the last entry has been loaded to indicate when we can start using the lookup tables.

The lookup tables and the bootup control circuitry use a "bootup" clock, Clk_b, which is independent of the two system clocks. This separate clock allows us to load both systems' lookup tables from a single ROM. It also allows us to load the lookup tables at a slower rate, as may be required if we have a slow ROM, slow run-time scheduling hardware, or are using software to load the lookup table.

Using an independent bootup clock has the disadvantage of making the *RDY* signal asynchronous to the two system clocks. This problem is easily solved by ignoring *RDY* altogether, and instead manually determining some minimum number of equivalent system clocks it takes to load the tables and making sure that the systems wait at least that long before attempting to transfer data. Alternatively, a better solution is to use the $\overline{Z}$ output of one of the counters as the bootup clock. This clock would run at the coincidence frequency, and would be synchronous to both systems. We can also use a frequency-divider on the $\overline{Z}$ output if a slower bootup clock is desired.

To make the chip easy to use with conventional, combinational ROM chips, an on-chip bootup counter, B, is provided. This counter, which uses the bootup clock, emits a decreasing 4-bit number that can be used to index a combinational ROM, allowing the ROM to output the appropriate entries as the lookup table is loaded. Ideally, the bootup counter should count from 15 to 0 during the first 16 clock cycles so that the lookup table entries are loaded in the proper sequence. However, since the counter is cleared on reset[3] it actually starts at 0 first before going to 15 and counting down, causing a shift in the lookup table entry sequence. That is, entry 15 of the lookup table would be loaded with entry 0 from the ROM, entry 14 loaded with entry 15 from the ROM, and so on. This shift is easily accounted for, however, by having the software that programs the ROM store the intended contents for lookup table entry x in ROM entry $(x + 1) \bmod 16$ (for some x).

6.2.4 Fabricating and Testing the Chip

The prototype chip was designed using Mentor Graphics' GDT CAD tools. After extensive simulation, the chip design was sent to MOSIS for fabrication and packaging. Approximately two months later, MOSIS sent back four copies of the chip, each packaged in a 40-pin DIP. These chips were then tested on a breadboard under two configurations.

The first test configuration was simple: the three clocks were connected together to a pulse generator, and the lookup table inputs were connected to the bootup counter outputs. In this way, the lookup tables and counters could be tested without the need for other chips.

[3]The bootup counter and the index counters are built from the same design.

	Measured (ns)		Simulated (ns)	
	rising	falling	rising	falling
RE_m	28	28	16	21
$\overline{Z_m}$	24	35	12	20
bit 0 of $count_B$	22	22	10	10

Figure 6-7: Measured and simulated clock-to-Q propagation delays of some chip outputs.

	Measured from	Measured (ns)	Simulated (ns)
Rise time	0 V to 3.2 V	4	3
	0.5 V to 4.5 V	4	1.7
	0 V to 5 V	15	4
Fall time	5 V to 1.8 V	3	3.7
	4.5 V to 0.5 V	3.5	3.7
	5 V to 0 V	5	4.3

Figure 6-8: Measured and simulated rise and fall times of the chip's counter outputs.

A second test configuration involved two copies of the chip, with the first copy generating the two system clocks of the second copy through clock division. In this way, any unwanted glitches on the $\overline{Z}$ output of the first chip would be detected, and the ability of the second chip to take three independent clocks would be tested.

Of the four chips, one was damaged by a human error during testing, while the rest worked as expected. Timing measurements were taken using one of the chips, and were compared with the simulation results. As shown in Figure 6-7, the measured propagation delays were generally higher than their simulated values, but are roughly within a factor of two of their simulated values. This is not abnormal for prototype chips fabricated by MOSIS. The measured and simulated rise and fall times are shown in Figure 6-8. Comparing these values, however, would probably not be meaningful since the simulated times were taken from *inside* the chip (i.e., without including the pin driving circuitry), and thus does not necessarily represent the rise and fall times outside the chip.

The chip was able to operate successfully in the lab at clock speeds up to 25 MHz. It could not be tested above this frequency because the available pulse generator could not go faster. From the timing measurements in Figure 6-7, however, it seems unlikely that the chip's actual maximum frequency would be significantly higher than 25 MHz. Since the chip uses dynamic memory elements, it was also tested at low frequencies. It operated

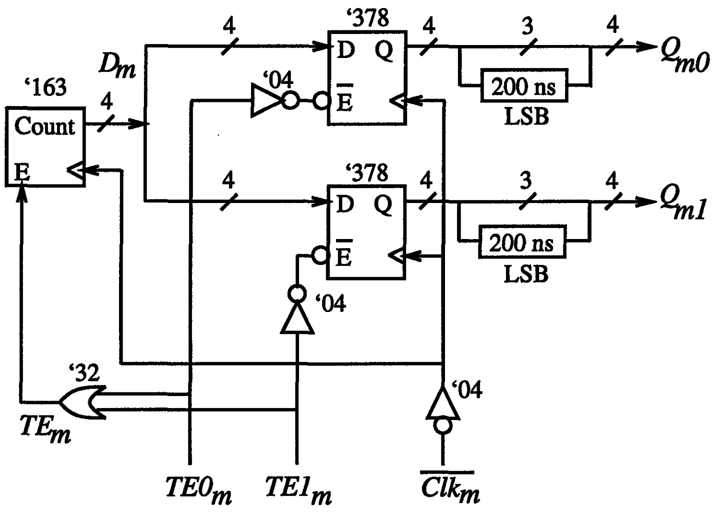


Figure 6-9: The transmitter's circuitry.

successfully at 333 Hz, the pulse generator's lower limit.

6.3 The Demonstration Circuit

A demonstration circuit using the prototype chip was built to test the rational clocking technique and demonstrate its usefulness. This circuit was built on a breadboard and uses off-the-shelf digital components from the 74Fxx family. It employs clock division to generate the system clocks, and implements a unidirectional 4-bit double-buffered data channel from system M to system N. The lookup tables are loaded from a single ROM large enough to allow the user to choose among different tables generated by several algorithms. This section describes the various parts of the demonstration circuit as well as the results of the experiments done on it.

6.3.1 The Transmitter's Circuitry

In the demonstration circuit, system M transmits to system N using a double-buffered data channel. Figure 6-9 shows the transmitter's circuitry. As shown, two 4-bit transmit registers (74F378) are used as in the double-buffered hardware shown in Figure 4-8. Instead of using $TSel$ and TE, however, we use two separate enable signals, $TE0$ and $TE1$, defined as follows:

$$\begin{aligned} TE0 &= TE \cdot \overline{TSel} \\ TE1 &= TE \cdot TSel \end{aligned} \tag{6.3}$$

By programming the lookup tables to generate these signals instead of $TSel$ and TE, we avoid the need for a demultiplexer, and save on hardware and wiring.

For a data source, we use a 4-bit counter (74F163) with its enable control connected to the TE signal derived by ORing $TE0$ and $TE1$. Configured in this way, the counter uses TE as the D *Taken* signal. That is, whenever TE is 1, the counter knows that its current value will be latched by one of the transmit registers on the coming clock edge, and that it is thus safe to place the next value on D_m during the next cycle. Thus, in this case, reaction latency compensation is unnecessary.

Using the manufacturer's specifications for the different components, and making the necessary adjustments to account for the delay through the clock's inverter, we arrive at the following propagation and contamination delays for the two transmit registers (see Appendix A for more details):

$$\begin{aligned} t_{cm} &= 4.5\text{ns} \\ t_{pm} &= 15.5\text{ns} \end{aligned} \tag{6.4}$$

To make decision window violations more detectable, we connect the LSB of the transmit register's 4-bit output to a 200 ns delay element. This widens the transition window by increasing t_{pm} while keeping t_{cm} the same. That is, we now have:

$$\begin{aligned} t_{cm} &= 4.5\text{ns} \\ t_{pm} &= 215.5\text{ns} \end{aligned} \tag{6.5}$$

Furthermore, this scheme makes it possible to simulate the effect of metastable states, which are very hard to catch in practice. By delaying the LSB, we cause an out-of-sequence number to appear at the receiver's input during the transition window. Thus, if the receiver tries to latch its inputs any time during the transition window, it would get a wrong value — even if no metastable states are actually generated.

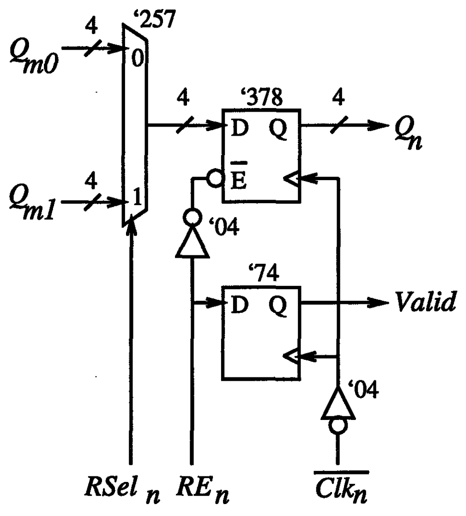


Figure 6-10: The receiver's circuitry.

6.3.2 The Receiver's Circuitry

The receiver's circuitry, shown in Figure 6-10, is simply a direct implementation of the receiver's circuitry in Figure 4-8. An additional register (74F74) is used to latch RE to produce a "valid" bit that tells the receiving system whether the current value of Q_n represents new data, or if it is old and should be ignored.

To calculate the equivalent setup and hold times of the receiver, we must take account not only of the delay through the clock inverter, but also of the additional delay through the multiplexer. Using the information in Appendix A, we arrive at the following setup and hold times:

$$\begin{aligned} t_{sn} &= 9.5\text{ns} \qquad (6.6) \\ t_{hn} &= 4.0\text{ns} \end{aligned}$$

6.3.3 Clock Generation and Reset Circuitry

For simplicity, the demonstration circuit uses clock division instead of clock multiplication. This allows us to avoid using a PLL and having to deal with its analog requirements. As shown in Figure 6-11, we implement clock division by taking both system clocks of a rational clocking chip, RCC A, from a single clock, $\overline{Clk_{high}}$, and using the $\overline{Z_m}$ and $\overline{Z_n}$ outputs of

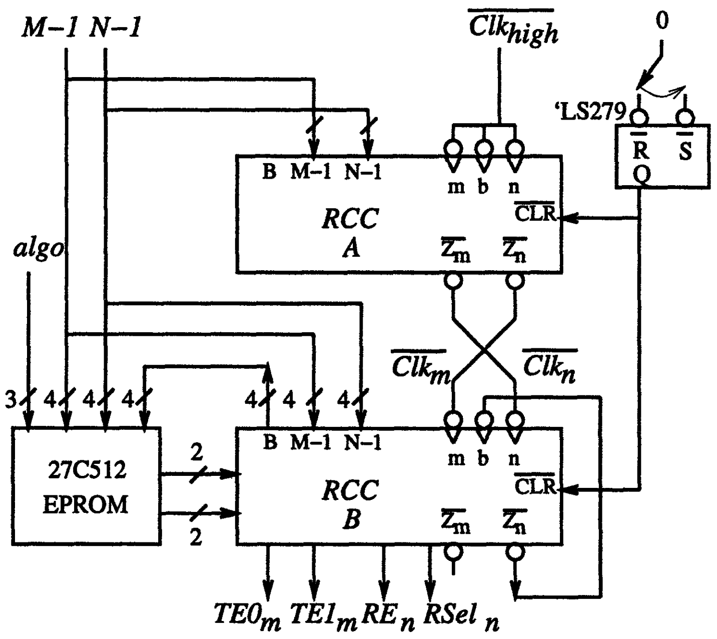


Figure 6-11: The clocking and communication control circuitry.

this chip as the $\overline{Clk_n}$ and $\overline{Clk_m}$ inputs respectively of a second rational clocking chip, RCC B.[4] In the lab, $\overline{Clk_{high}}$ was taken from a pulse generator running at 10 MHz. The timing relationships of the resulting clock waveforms are shown in Figure 6-12. For clarity, the counter values that determine each $\overline{Z}$ output are also shown. Note that by clearing all the counters simultaneously, the asynchronous $\overline{CLR}$ input makes the active edges of the two clocks, $\overline{Clk_m}$ and $\overline{Clk_n}$, coincide naturally when clock division is implemented.

A reset signal for the two rational clocking chips is generated by a TTL R-S flip-flop (74LS279) acting as a debounced switch [31]. To reset the circuit, the user first grounds the $\overline{R}$ input, then disconnects it and grounds the $\overline{S}$ input. This has the effect of first clearing, then setting the flip-flop without generating glitches due to switching mechanics. A TTL chip is required here so as to guarantee that an unconnected input is read as a logical 1. This prevents both $\overline{R}$ and $\overline{S}$ from being 0 at the same time.

[4]In the circuit diagram, a bar over a signal name indicates that the signal is active low. For clock signals, this means that the negative edge is considered to be the active edge.

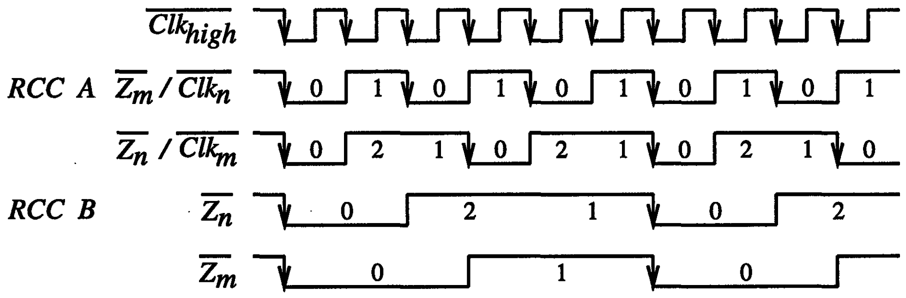


Figure 6-12: The demonstration circuit's clock waveforms.

6.3.4 The Communication Control Circuitry and the ROM

A second copy of the rational clocking chip, RCC B, is used to generate the control signals. As shown in Figure 6-11, RCC B generates the control signals needed to allow system M to transmit to system N. To implement bidirectional transfer, we can simply use a third copy of the chip to generate the other set of control signals.

RCC B is loaded from a combinational ROM (27C512) which is programmed by the communication scheduling software. It takes 15 bits of address input and outputs 8 bits, as shown in Figure 6-13. The lowest-order 4 bits of the address are taken from RCC B's bootup counter, and indicate the current clock cycle whose corresponding control signals are to be read. The next 8 bits specify M and N. The highest order 3 bits allow the user to switch between tables generated by different algorithms. At least one of the 8 possible settings corresponds to having no flow control at all — i.e., TE and RE are fixed at 1, and $TSel$ and $RSel$ are fixed at 0. This allows us to demonstrate clearly the need for flow control. Figure 6-14 shows the bit encoding actually used in the demonstration circuit.

Programming the ROM for use with the rational clocking chips requires some remapping of the lookup tables. First, we must account for the fact that the chip counters decrement instead of increment, as we have assumed so far in this paper and in the communication scheduling software. Then, we must fix the shift caused by the chip's bootup counter. Figure 6-15 shows the relationship between the cycle number ($cycle_m$), the counter value for the cycle $count_m$, and the ROM address (i) where the control signals for this cycle should be stored. From this table, we can determine what goes into some ROM address i

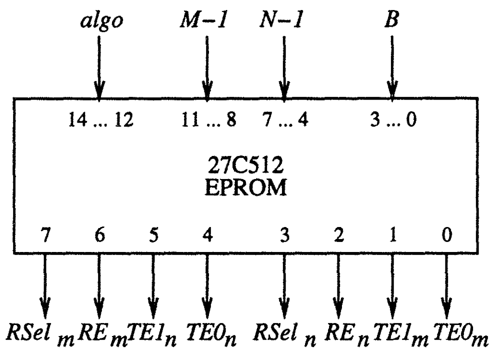


Figure 6-13: The ROM inputs and outputs.

Code	Algorithm
000	No Flow Control
001	Original Single-Buffered
010	Generalized Run-Time
011	Greedy Double-Buffered
100	Lazy Double-Buffered
other	undefined

Figure 6-14: Bit codes for different algorithms.

$cycle_m$	$count_m$	i
0	0	1
1	$M-1$	0
2	$M-2$	$M-1$
3	$M-3$	$M-2$
...	...	...

Figure 6-15: Mapping between cycle number and ROM address (for system m).

as follows:

$$ROM[i] \leftarrow Table[(M - i + 1) \bmod M] \tag{6.7}$$

where $Table[x]$ for some x represents lookup table entry x if we are using an incrementing counter.

6.3.5 Results and Recommendations

The demonstration circuit was tested using different frequency ratios and scheduling algorithms. It performed as expected, and provided a better feel for how rational clocking solves the flow control and synchronization problem. It was especially interesting to see how double-buffering allowed 100% throughput to be achieved where it could not be done with single-buffering. The waveforms produced by this circuit were very similar to those produced by the Verilog simulations shown in the next section.

In order to provide a more realistic demonstration of how rational clocking would actually be implemented in applications, the demonstration circuit should be modified in the future to use clock multiplication instead of clock division. Among other things, this would allow the user to test the robustness of the rational clocking circuitry in the presence of phase jitter and other possible problems involving the PLL.

6.4 Verilog Modelling and Simulation

The prototype chip and the demonstration circuit tested the rational clocking technique with a simple table-based implementation. To test the more complex run-time implementation, a model using the Verilog hardware description language [32] was constructed and simulated using Cadence Systems' Verilog tools. This model included a bidirectional double-buffered data channel, three versions of the run-time scheduling hardware, and circuitry for computing constants such as e_0 and y_0 at bootup. The complete source code is listed in Appendix D.

6.4.1 The Data Channel

The Verilog model contains two double-buffered data channels (one for each direction) implementing the original double-buffered configuration shown in Figure 4-8. The transmit registers in these channels, unlike those in the demonstration circuit, are controlled by TE and $TSel$, instead of $TE0$ and $TE1$. As with the demonstration circuit, counters with enable inputs are used as data sources, and an extra 1-bit register is used by the receiver for the valid bit.

This model does not use a delay element to increase propagation delay as done in the demonstration circuit, but instead uses Verilog features to simulate contamination and propagation delays in the transmit registers, and setup and hold times in the receive registers. These timing parameters can be specified by the user in units of Δt.

6.4.2 The Scheduling Hardware

The Verilog model implements three different versions of the scheduling hardware: the lazy and greedy basic run-time schedulers of section 5.2, and the generalized run-time scheduler of section 5.3. Conditional compilation directives allow the user to select between these versions by simply defining the appropriate names using the `'define` compiler directive.

6.4.3 The Bootup Computation Circuitry

In order to work properly, the scheduling hardware requires precomputed values of certain constants such as e_0 and y_0. In the Verilog model, we compute most of these constants at bootup time using the circuits described in section 5.2.4.

The model uses both combinational and sequential versions of the modulo circuit. The combinational version is used for computing e_0, y_0, and the value of $(i-1) \bmod M$ in the *LD Gen* module. The sequential version is used for computing ΔNM, and $e_0 \bmod M$.

Each of the three scheduling hardware implementations uses its own version of the bootup computation circuitry. Some versions use additional circuitry to perform necessary

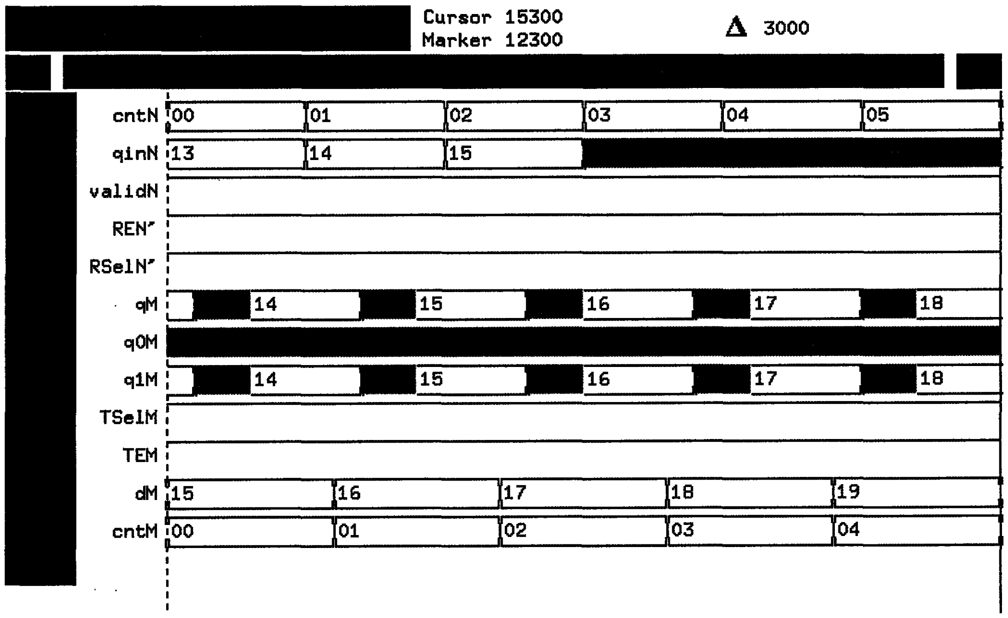


Figure 6-16: Slower transmitter case with no flow control.

adjustments to e_0 and y_0 as discussed in Chapter 5. The generalized run-time scheduler also uses extra circuitry for computing e_{max} and e_{min}.

6.4.4 Results and Recommendations

The three versions of the scheduling hardware were each simulated using several sets of timing parameters and frequency ratios, and the resulting schedules were compared with the schedules generated by the communication scheduling software. All the versions worked as expected, thus proving the correctness of the various run-time scheduling circuits.

Figures 6-16 to 6-19 show sample outputs from the Verilog simulator for the case where $M = 5$, and $N = 6$, and the timing parameters are $S = 1$, $H = 1$, $C = 1$, and $P = 3$, for both systems. In each of these figures, *cntM* and *cntN* indicate the index counters' outputs, *d* represents data at the input of the transmit register, *q*0 and *q*1 are the outputs of the two transmit registers, *q* is the output of the multiplexer, and *qin* is the output of the receive register. (Note each of these signal names are followed by *N* or *M* in the figure, depending on the system they belong to.) Shaded areas represent unknown values, and occur during the transition windows of the transmit registers and when the receive register suffers a decision window violation.

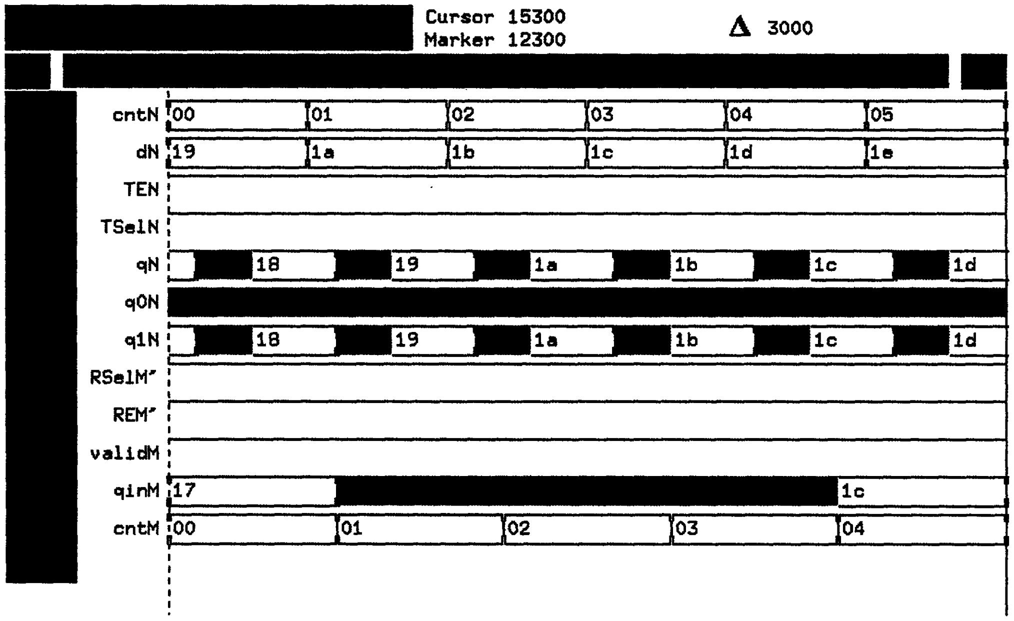


Figure 6-17: Slower receiver case with no flow control.

Figures 6-16 and 6-17 show the result of not using any flow control at all. It is clear from these figures that we not only suffer data loss as the faster system outruns the slower one, we also encounter a lot of decision window violations. Figures 6-18 and 6-19 show the result of using the generalized run-time scheduler with double-buffering. As shown, by emitting the right control signals at the right times, we successfully solve the flow control and synchronization problems. Furthermore, by using double-buffering, we are able to achieve 100% throughput where it would otherwise not have been possible.

There are still many ways to improve the Verilog model. It may be useful, for example, to modify the generalized run-time scheduler to work with timing parameters that are not integer multiples of Δt. This can be done by making the appropriate registers and multiplexers wider. Most efforts at improving this Verilog model, however, should be directed towards developing and simulating appropriate interfaces for real applications. It would particularly be useful, for example, to start integrating the Verilog model built for this thesis, with the Verilog model of the current NuMesh CFSM.

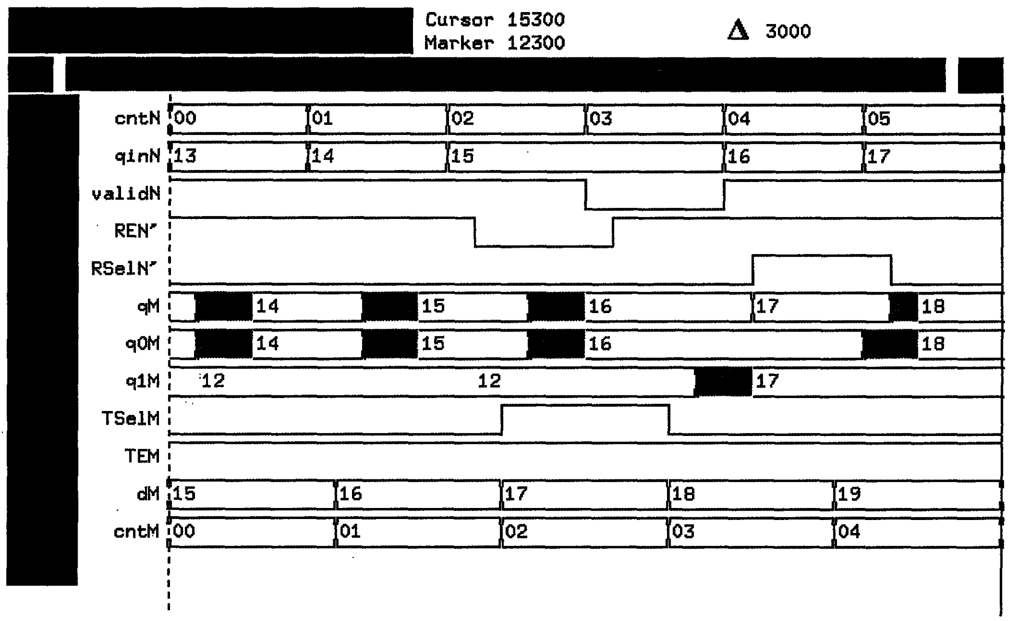


Figure 6-18: Slower transmitter case with generalized run-time scheduling and double-buffering.

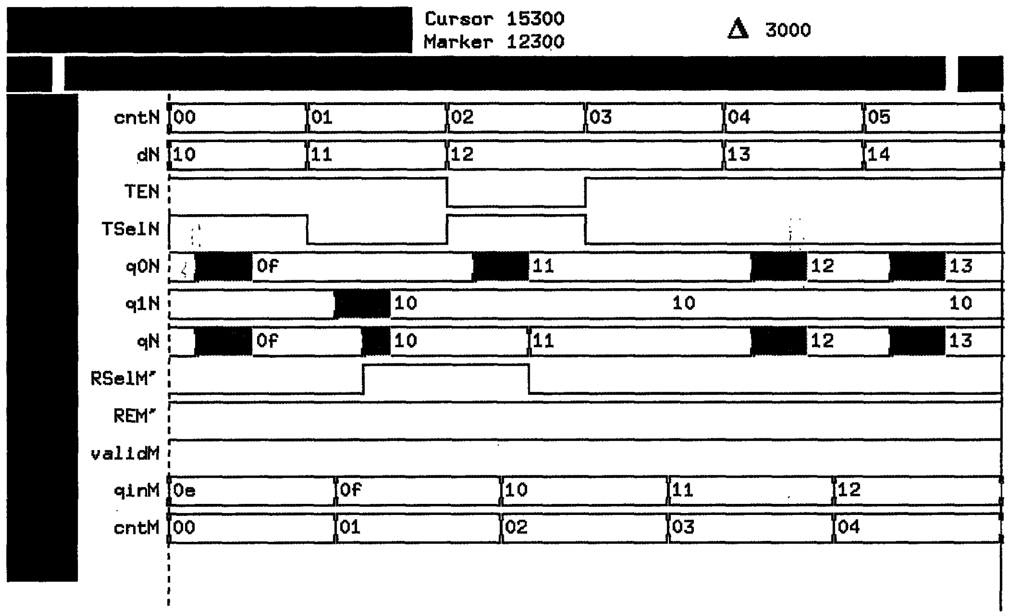


Figure 6-19: Slower receiver case with generalized run-time scheduling and double-buffering.

Chapter 7

Conclusion

Rational clocking addresses the increasingly common problem of providing efficient communication between two systems running at different clock speeds. Traditional solutions to this problem have generally lacked either efficiency or flexibility, and leave much to be desired. Rational clocking provides both efficiency and flexibility by requiring a rational relationship between the two clock frequencies. This rational frequency constraint makes it possible to achieve communication without handshaking and synchronizers for a wide variety of frequency pairs.

The most straightforward way to implement rational clocking is to use lookup tables that generate flow control signals according to a precomputed communication schedule. This schedule is determined given the frequency ratio and the two systems' timing parameters using a simple greedy algorithm which selectively enables and disables data transfers in such a way that timing violations are avoided. The table-based hardware can be made more versatile by using the algorithm on several different frequency ratios and then storing the resulting schedules into a single generic table.

Although the straightforward implementation of rational clocking provides high levels of throughput efficiency for different frequency ratios given typical timing constraints, it cannot maintain this high efficiency given the tighter timing constraints typical in high-performance systems. The double-buffering technique, which uses two transmit registers instead of one, solves this problem by providing a data channel where the possibility of contamination between successive data transfers is eliminated. This contamination-free data channel simplifies the scheduling requirements, and can be proven to guarantee 100% throughput for the slower system in most applications, regardless of frequency ratio and timing parameter values.

The table-based implementation has the disadvantage of trading-off space for flexibility. A generic table accommodating different frequency ratios and timing parameters, for example, requires as much as $O(N^7)$ bits of memory, where N is the maximum possible value for the frequency ratio numerator and denominator. This problem can be solved by avoiding precomputation altogether and taking advantage of the clocks' regularity to schedule communications at run-time using specialized rate multipliers. This run-time scheduling hardware only requires $O(\lg N)$ bits to handle all possible ratios and timing parameters. Two major variants of run-time scheduling have been presented. The basic algorithm works with contamination-free data channels, and is easy to implement. The general algorithm extends the basic algorithm to cases where contamination is possible, and makes it possible to design run-time scheduling hardware that can be used on single-buffered systems, but take advantage of double-buffering when available.

All these techniques were studied and tested using a combination of software, hardware, and simulations. Communication scheduling software was written for testing and using the different scheduling algorithms. A VLSI prototype chip employing programmable lookup tables was fabricated and used in a circuit that demonstrates double-buffering. The different run-time scheduling circuits were modelled and simulated in Verilog. These different tools successfully demonstrated the correctness and usefulness of rational clocking, and provide a base for future research in applying rational clocking to real systems.

7.1 Putting It All Together

The ideas and techniques presented in this thesis each have their own advantages and disadvantages, and can be used together when applying rational clocking on high-performance and high-flexibility systems. Figure 7-1 shows a single circuit that integrates these different techniques to achieve efficiency, flexibility, and cost-effectiveness.

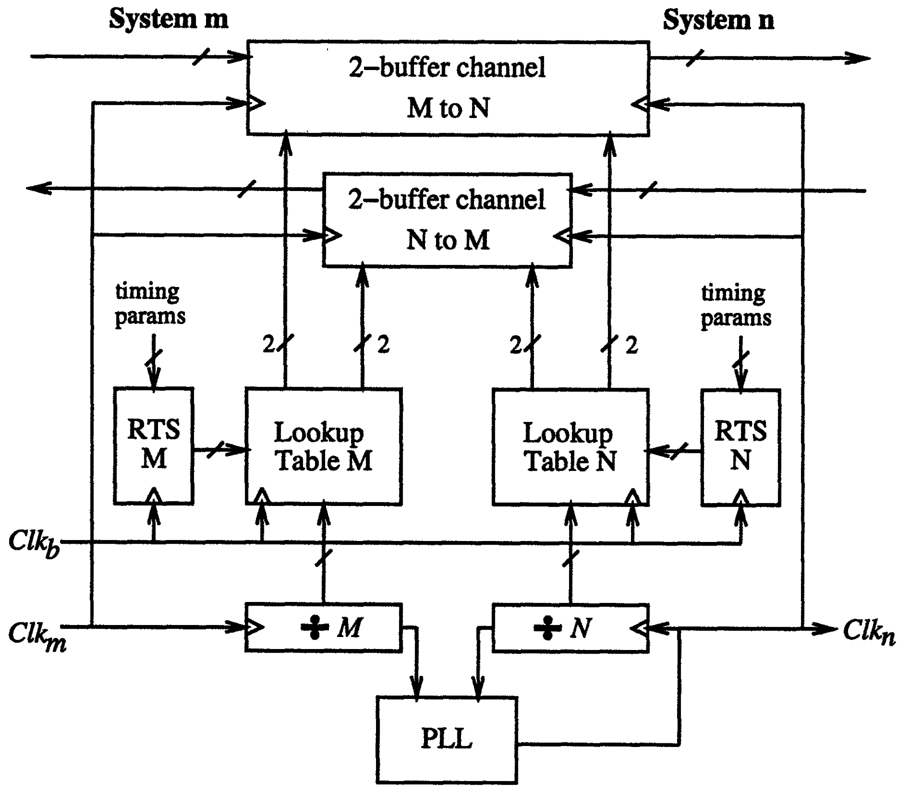


Figure 7-1: Putting it all together in a single circuit.

This circuit provides 100% throughput efficiency for all frequency ratios and timing parameters by using double-buffered data channels. The use of double-buffering also simplifies communication scheduling, and in particular, allows us to use the basic run-time scheduling algorithm, which is easier to implement than the general algorithm and less sensitive to variations in the timing parameter values. Run-time scheduling hardware allows maximum flexibility in choosing frequency ratios and timing parameters without taking-up a large amount of space. It can be used alone to generate control signals continuously, but in very high-performance systems where the computation delay through the run-time scheduler limits the maximum clock speed, is better used together with programmable tables as shown in the figure. These tables would not be unreasonably large or expensive — they need only around $4N$ bits each, where N is typically less than 16 — and thus can be made extremely fast. By loading these tables from the run-time scheduling hardware using a slow bootup clock, we can combine the flexibility of the run-time scheduler with the speed of the small lookup tables.

7.2 Future Research

There is still a significant amount of research, both theoretical and practical, that can be done on various aspects of the rational clocking technique.

On the theoretical side, the properties of the single-buffered algorithms — the original greedy algorithm of section 3.3 and the generalized algorithm of section 5.3 — can be studied further. In particular, it would be useful to determine the exact conditions that lead to sub-optimal throughput efficiency given a certain choice for P_{begin}, and to develop a way to choose the optimal P_{begin} without having to run through all the possibilities. It may be possible to solve this problem by unrolling the loop in the run-time scheduling algorithms and deriving an analytic expression of the control signals as a function of M, N, the timing parameters, and the current clock cycle number. Another good subject for future research is the application of double-buffering to out-of-phase communication. As discussed in section 4.4.2, this can be valuable in multiprocessor systems and should be studied further.

On the practical side, research can be directed towards applying rational clocking in real multiprocessor and uniprocessor systems. This would involve improving the scheduling software, performing more hardware experiments, and developing interfaces. It would also be useful to develop a standard "off-the-shelf" rational clocking chip that would allow system designers to easily incorporate rational clocking into their systems. This circuit would use run-time scheduling, either alone or with programmable lookup tables, to provide flexibility without requiring a large amount of space.

7.3 Concluding Remarks

In this thesis, we presented the rational clocking technique and improved it significantly, making its application in real systems not only more promising, but more feasible as well. More research should now be directed towards actually implementing rational clocking, and making it a standard part of existing and future high-performance computer systems.

Appendix A

Computing Timing Parameters

In computing the timing parameters of a system, it is not enough to consider only the parameters of the transmit and receive registers. The inputs, outputs, and clocks of these registers may suffer additional delays that can change their effective timing parameters. This appendix discusses general rules for handling these delays, and then applies these rules to the computation of the demonstration circuit's timing parameters.

A.1 Effective Timing Parameters

Figure A-1 shows three possible ways in which additional delays can affect a register's timing parameters. In general, delays can occur at a register's data output, data input, or clock input. As shown in the figure, we can model these delays with a buffer having contamination and propagation delays t_{cB} and t_{pB}. Taking the register and the buffer together as a single module, we can then use t_{cB}, t_{pB}, and the register's timing parameters, t_s, t_h, t_c, and t_p,

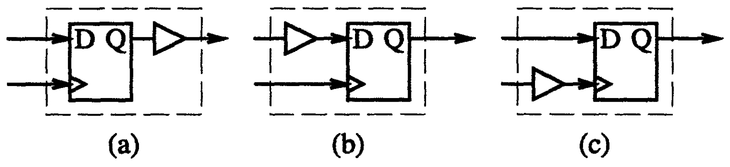


Figure A-1: Possible delay configurations. (a) Output. (b) Input. (c) Clock.

to compute the module's *effective* timing parameters, t'_s, t'_h, t'_c, and t'_p, in different ways depending on the delay configuration.

A.1.1 Delayed Output

The simplest case is that of a delay connected to the register's output, shown in Figure A-1(a), where the delays of the buffer simply add to the output delays of the register, giving the following effective timing parameters:

$$t'_s = t_s, \quad t'_h = t_h \tag{A.1}$$

$$t'_c = t_c + t_{cB}, \quad t'_p = t_p + t_{pB} \tag{A.2}$$

A.1.2 Delayed Input

The next case is that of a delayed input, shown in Figure A-1(b). Here, the delays of the buffer affect the input timing constraints of the module enclosed in the dashed box. The propagation delay through the buffer requires us to present data at the module's input t_{pB} before the setup time, while the contamination delay allows us to change the module's input t_{cB} before the hold time without violating the register's timing constraints. This leads to the following effective timing parameters:

$$t'_s = t_s + t_{pB}, \quad t'_h = t_h - t_{cB} \tag{A.3}$$

$$t'_c = t_c, \quad t'_p = t_p \tag{A.4}$$

A.1.3 Delayed Clock

In the last case of a delayed clock, shown in Figure A-1(c), both the input timing constraints and the output delays are affected. The propagation delay through the buffer requires us to hold the register's inputs stable t_{pB} after the register's hold time, while the contamination delay allows us to delay the input by an extra t_{cB} without violating the setup time. These

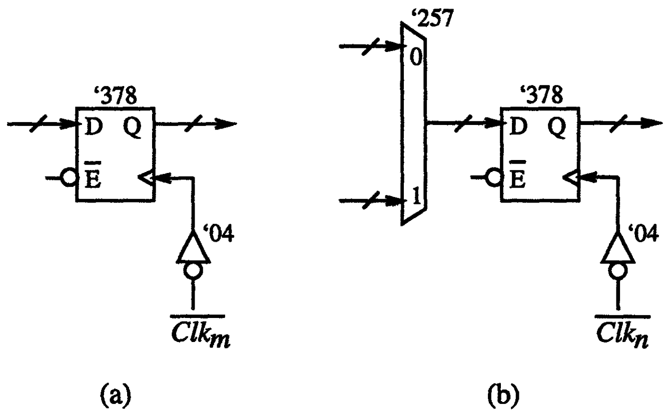


Figure A-2: The demonstration circuit's delays. (a) Transmit register. (b) Receive register.

delays also cause latching to be delayed, and thereby add to the output delays of the register. Thus, we have the following effective timing parameters:

$$t'_s = t_s - t_{cB}, \quad t'_h = t_h + t_{pB} \tag{A.5}$$

$$t'_c = t_c + t_{cB}, \quad t'_p = t_p + t_{pB} \tag{A.6}$$

A.1.4 Applications

Delays at the outputs usually represent combinational logic processing the register's output, as in the case of the transmit modules of Figure 2-1. Similarly, delays at the inputs represent logic processing the register's inputs, such as the receiver's multiplexer in the double-buffered hardware of Figure 4-8. Delays at the clock input can be due to clock skew, phase jitter, or both. In a typical system, these three delay configurations often occur together in a single module. In these cases, the effective timing parameters can be computed by successively applying the appropriate equations.

A.2 The Demonstration Circuit

The relevant circuitry for computing the effective timing parameters of the transmit and receive registers in the demonstration circuit are shown in Figure A-2. (These are taken

	74F378	74F04	74F257
Setup time of D input (ns)	4.0	N/A	N/A
Hold time of D input (ns)	0.0	N/A	N/A
Contamination delay (ns)	3.0	1.5	2.0
Propagation delay (ns)	9.5	6.0	7.0

Figure A-3: Worst-case timing parameters of circuit components.

from Figures 6-9 and 6-10.) The components in these circuits are all from the 74Fxx family, and have the worst-case timing parameters shown in Figure A-3 [33].

The transmit register has a delayed clock, so its effective contamination and propagation delays can be computed as follows using Equation A.6:

$$\begin{aligned} t'_c &= 3.0 \text{ ns } + 1.5 \text{ ns } = 4.5 \text{ ns} \\ t'_p &= 9.5 \text{ ns } + 6.0 \text{ ns } = 15.5 \text{ ns} \end{aligned} \tag{A.7}$$

The receive register has a delayed clock *and* a delayed input, so its effective parameters are computed using Equations A.6 and A.3 successively as follows:

$$\begin{aligned} t'_s &= (4.0 \text{ ns } - 1.5 \text{ ns }) + 7.0 \text{ ns } = 9.5 \text{ ns} \\ t'_h &= (0.0 \text{ ns } + 6.0 \text{ ns }) - 2.0 \text{ ns } = 4.0 \text{ ns} \end{aligned} \tag{A.8}$$

The other effective timing parameters can be computed in a similar manner, but only the ones computed here are relevant for determining communication schedules.

Appendix B

The Scheduling Software

This appendix presents the details of the scheduling software, starting with a discussion of the integer-based time representation scheme mentioned in Chapter 6, and some scheduling algorithm implementation details. A complete listing of the C source code and the LaTeXsupport files are then provided.

B.1 Representing Time with Integers

The original communication scheduler used for generating the data presented in [1] and [2], computed the throughput efficiency of the original algorithm in section 3.3 given M and N as integers and the timing parameters as floating-point numbers equal to the fraction of a clock cycle they represent (e.g., $P = 0.20$ would mean that the propagation edge occurs 20% of a clock cycle after the clock edge). This program was simple and worked well in most cases, but had an unfortunate limitation: it was prone to occasional floating-point rounding errors. These errors caused the program to prohibit data transfers unnecessarily, and resulted in reduced throughput efficiency. If we compare the throughput efficiency table shown in Figure 3-5 with the original one in [2], for example, we would see that even though the timing parameters used are the same, the table in [2] only claims 50% throughput for a transmit-receive ratio of 3/2, instead of the actual 100% throughput.

To avoid this problem, the software used in this thesis performs all computations using only integers. Time is expressed in integer units of dt, defined as:

$$dt = \Delta t/d \tag{B.1}$$

where $\Delta t = T_{MN}/MN$ as defined in Chapter 5, and d is some integer specified by the user. These units are useful because they allow the positions of the two systems' clocks to be represented by exact integers. That is, substituting the definitions of dt and Δt into the formula for the coincidence period, Equation 3.2, we get:

$$\begin{aligned} T_M &= N\Delta t = (N \cdot d) \cdot dt, \\ T_N &= M\Delta t = (M \cdot d) \cdot dt \end{aligned} \tag{B.2}$$

Furthermore, by making d large, we can express arbitrarily small units of time (as long as d does not get so large that the times we want to express do not fit in a 32-bit "long" integer).

Although the software performs all computations internally using units of dt, it allows the user to specify timing parameters in other units as well. As mentioned in section 6.1, the user has a choice of expressing timing parameters in units of dt, in units of Δt, as fractions of a clock cycle, or in absolute units. When expressing time in absolute units, the user has the further choice between using these units with clock division or clock multiplication. Converting timing parameters expressed in units of dt and Δt is straightforward, but converting parameters expressed as fractions or in absolute units requires additional analysis, and is discussed in the following sections.

B.1.1 Using Fractions

As discussed in section 3.5.1, it is useful to specify the timing parameters as fractions of a clock cycle if we presume that our target systems have timing parameters that scale with their minimum clock periods. If a timing parameter P of system m is equal to some fraction x of the system clock edge T_M, and we choose d such that $x = p/d$ for some integer p, then we can express P in units of dt as follows:

$$P = x \cdot T_M = \frac{p}{d} \cdot (N \cdot d) = p \cdot N \tag{B.3}$$

For example, if $M = 5$, $N = 6$, $d = 1000$, and $P = 0.20T_M$, then $p = 200$, and $P = 200 \cdot 6 = 1200$ time units. Note that this representation scheme limits us to using rational numbers for the timing parameters. This is not a problem, however, since we can make these parameters sufficiently close to their actual value by making d large enough.

B.1.2 Using Absolute Units with Clock Division

It may occasionally be desirable to use the same piece of hardware at different frequencies and frequency ratios. In this case, the timing parameters would not scale with the clock period but would be fixed at some value while the clock period changes.

If we are using clock division, then converting the timing parameters into units of dt is easy. Using the clock division circuit from Figure 3-1, we get (from Equations 3.3 and 6.2):

$$\Delta t = 1/f_{high} = T_{high} \tag{B.4}$$

where T_{high} is the period of Clk_{high} in seconds. Thus, if a timing parameter P, on *either* system, is equal to t_p seconds, we can express it in units of dt as follows:

$$P = \frac{t_p}{\Delta t} \cdot d = \frac{t_p}{T_{high}} \cdot d \tag{B.5}$$

where we choose d such that P is an integer.

When taking timing parameters in this mode, the program asks the user to enter t_p and T_{high} as integers. Although it is convenient to use standard time units such as picoseconds or nanoseconds in giving these values, any units that allow t_p and T_{high} to be expressed as integers can be used. This is because it is the *ratio* of t_p to T_{high} that is important, rather than their actual values. In fact, it is preferable to make t_p and T_{high} small integers whenever possible in order to avoid causing overflows during computation.

B.1.3 Using Absolute Units with Clock Multiplication

Using absolute units with clock multiplication is slightly more difficult. Here, the program assumes that Clk_n is being generated from Clk_m using the circuit in Figure 3-2, and asks the user to enter t_p and T_M as integers. From Equation B.3, we have

$$\Delta t = T_M / N \tag{B.6}$$

Thus, if a timing parameter P, on *either* system, is equal to t_p seconds, we can express it in units of dt as

$$P = \frac{t_p}{\Delta t} \cdot d = \frac{t_p \cdot N}{T_M} \cdot d \tag{B.7}$$

again choosing d to make sure that P is an integer. Note that the specific value of N is used here, and that this equation applies to timing parameters of *both* systems m and n. The asymmetry arises because we assume Clk_m to be the source clock and generate Clk_n from it. Thus, T_M is fixed, while T_N changes.

B.2 Scheduling Algorithms

Four scheduling algorithms were implemented in the program: the original greedy algorithm for single-buffering, the greedy and lazy versions of the run-time scheduling algorithm for double-buffering, and the generalized run-time scheduling algorithm. As can be seen from the source code in the next section, these algorithms were not implemented exactly as they have been described in the text of this thesis.

In implementing the original algorithm from section 3.3, for example, the program does *not* use "wrap-around" modulo arithmetic. That is, to check if it has passed P_{begin} on system m, it actually checks if it has passed $P_{begin+M}$. This makes the algorithm code simpler, and also makes it easier to correctly handle situations with abnormal timing parameters (i.e., timing parameters that are negative or are larger than a clock period).

Some minor modifications were also made to the three run-time scheduling algorithms. Instead of using units of Δt to represent timing parameters as done in Chapter 5, the program uses units of dt. Consequently, whenever M or N are used in the algorithms in Chapter 5 to represent the clock periods of the systems, they are replaced by $M \cdot d$, and $N \cdot d$ respectively. Instead of adding M to e_{old} in line 9 of the basic algorithm in section 5.1, for example, the program would add $M \cdot d$. Representing the timing parameters in this way makes it possible to correctly handle non-integer (in terms of Δt) timing parameters in the generalized algorithm.

B.3 Source Code

This section contains the complete C source code for the scheduling software.

```
/*
 * rcsched.c
 *
 * Scheduler for Rational Clocking, integer version
 *    by Luis F. G. Sarmenta
 *
 *    thesis version, 950525
 *
 * Notes: 1) Here, coincidence cycles are called "beat" cycles, as they
 *           are in Pratt and Ward's original paper.
 *
 *        2) All timing parameters are treated internally as offsets
 *           in the positive time direction from the clock edge they belong
 *           to.  Thus, setup times are internally negative.
 *           However, the user can still enter _positive_ setup times as
 *           these get converted automatically.
 *
 */

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>


/* standard min and max definitions */

#define min(a,b)   ((a) < (b) ? (a) : (b))
#define max(a,b)   ((a) > (b) ? (a) : (b))


/*
 * modulo and div operators modified to work correctly
 *    with negative numbers
 *
 * This code was developed on a Sun (mont-blanc.lcs.mit.edu) with gcc.
 *    Under this system type, the / and % operators work as follows:
 *        22 / 7 =  3;    22 % 7 =  1;    22 / -7 = -3;    22 % -7 =  1;
 *       -22 / 7 = -3;   -22 % 7 = -1;   -22 / -7 =  3;   -22 % -7 = -1;
 *
 *    The way it handles the mod of negative numbers allows us to
 *    define a correct mod operator as follows:
 */
```

```
long mod( long a, long b )
{
   long temp;

   temp = a % b;
   if ( temp < 0 )
      temp += b;
   return( temp );
}

long ndiv( long a, long b )                /*  ndiv(a,b)*b + mod(a,b) = a  */
{
   return( (a < 0) ? ((a-(b-1))/b) : (a/b) );
}

#define ndivup(a,b) ((ndiv((a)+((b)-1), (b))))        /* ndiv then round-up */

long gcd( long a, long b )
{
   if (!b)
      return( a );
   else
      return( gcd( b, a % b ) );
}


/*
 * Some Global Constants
 */


#define MAXMN   64               /* Max. value for M and N. */
#define ROM_B    4               /* bits for specifying M and N to ROM */
#define ROM_N   (1<<ROM_B)        /* equivalent max M and N for ROM */


long d  = 1000;                  /* smallest unit of time = (delta T) / d  */
long Tref;                       /* reference clock's period in picoseconds.
                                    This will be initialized properly later. */

/* Options Variables
 */

int PEEK = 0;                   /* Show intermediate results? */

int DRAWSCHEDS = 1;             /* Draw text schedule diagrams? */

#define DTUNITS 1
#define DLUNITS 2
#define FRUNITS 3
#define CDIVPS  4
#define CMULPS  5

int paramunits = DLUNITS;       /* Parameter Units */
```

```
#define PAIR 1
#define TABLE 2

int compmode = PAIR;            /* Compute Mode */

#define NONE 0
#define ORIG 1
#define GBUFF2 2
#define DBUFFG 3
#define DBUFFL 4
#define MAXALGO 4

int algorithm = 1;              /* Scheduling Algorithm */

#define TEXT  1
#define LDIAG 2
#define LPLOT 3
#define ROM   4

int outmode = LDIAG;            /* Output format */

int usewithchip = 0;            /* Use with prototype chip */

char outfilename[41] = "rcsched.log";       /* Log file name */

FILE *outfile;


/*
 * Primitive operations ('P' stands for primitive)
 *
 * These operations assume that M and N are integers representing
 * the normalized frequencies.
 *
 * Most of these operations work in units of dt = (delta t) / d.
 */

   /* The following macros are self-explanatory. */

#define Pbeatperiod(M,N)       ( (M)*(N)*d )
#define Pperiod(M,N)           ( (N)*d )
#define Pfreq(M,N)             ( 1 / Pperiod((M),(N)) )
#define Pdeltat(M,N)           d


   /* Pclockedge gets ith clock edge of system M  */

#define Pclockedge(M,N,i)      ( (i)*(N)*d )

   /* Pcycle returns the cycle which time t belongs to
    * Pupcycle returns the next cycle at or after time t
    * note that if t is exactly at some edge, then Pcycle and Pupcycle
    * return the same cycle.
    */
```

```
#define Pcycle(M,N,t)         (ndiv( (t), (d*(N)) ))
#define Pupcycle(M,N,t)       (ndivup( (t), (d*(N)) ))

   /* Pshcycle gets the current cycle if t is a timing param edge that is dt
    * units away from its clock edge.
    */

#define Pshcycle(M,N,t,dt)    (Pcycle((M),(N),(t)-(dt)))

   /* Pfracttodt converts from a timing param in the fractional form num/d
    * to its equivalent value in units of dt.
    * (Move this somewhere else?)
    */

#define Pfractodt(M,N,num)    ((num) * ( Pperiod((M),(N)) / d ) )
#define Pdelttodt(M,N,num)    ((num) * ( Pdeltat((M),(N)) ))

   /* The following macros convert from a timing param in units of
    * picoseconds to its equivalent in units of dt, using the reference
    * clock period, Tref.  Pmpstodt is used for clock multiplication,
    * where Tref is the clock period of M.  Pdpstodt is used for
    * clock division, where Tref is Thigh.
    *
    ******
    * Note that for now, these macros round _down_ to the nearest integer.
    * If you want to be conservative, you should really round _down_
    * if the value is contamination delay, but round _up_ otherwise.
    * Possibly fix this later.  For now, just pick d and Tref such that
    * the result is an integer anyway.
    ******
    */

#define Pmpstodt(M,N,num)     (((num) * (N) * d ) / Tref )
#define Pdpstodt(M,N,num)     (((num) * d) / Tref )

   /* Ptptodt converts the value tp to units of dt according to the
    * current units used for specifying tp.
    *
    * This allows the units of tp to change without changing the macros.
    * It also has the nice property that the appropriate scaling rule
    * for different M and N is naturally taken care of.
    *
    * Note that in these macros, parameters called 'dt' are in units
    * of dt, while parameters called 'tp' are in whatever units the scaling
    * rule specifies.
    */

long Ptptodt( long M, long N, long tp )
{
   switch (paramunits) {
      case DTUNITS : return( tp );
      case FRUNITS : return( Pfractodt( M, N, tp ) );
      case DLUNITS : return( Pdelttodt( M, N, tp ) );
      case CMULPS : return( Pmpstodt( M, N, tp ) );
      case CDIVPS : return( Pdpstodt( M, N, tp ) );
      default: {
               fprintf(stderr,"Invalid Param Units ...  using dt ...\n" );
               return( tp );
             }
   }
}


   /* Now we use Ptptodt in Pshedge, which gives timing edge i if
    * timing param is tp away from edge.
    */

#define Pshedge(M,N,i,tp)     ( Pclockedge((M),(N),(i)) + Ptptodt(M,N,tp) )

   /*
    * The following macros return the appropriate cycle of the _other_ clock.
    * (Note the switch in M and N.)
    */

#define PnxtOcycle(M,N,t)        Pupcycle( (N), (M), (t) )
#define PprvOcycle(M,N,t)        Pcycle( (N), (M), (t) )

#define PnxtOshcycle(M,N,t,dt)   Pupcycle( (N), (M), (t)-(dt) )
#define PprvOshcycle(M,N,t,dt)   Pshcycle( (N), (M), (t), (dt) )


/*
 * Data Structures
 *
 */

typedef int  sched[MAXMN];    /* Schedule table */

typedef struct systype {
   long f;             /* frequency = cycles per beat */
   long S,H,C,P;       /* timing params in units of dt = (delta t) / d */
   sched TE, Tsel;     /* transmit register control signals */
   sched TD;           /* input to transmit register */
   sched Tto;          /* cycle to transmit to (mod beat period) */
   sched RE, Rsel;     /* receive register control signals */
   sched RQ;           /* output of receive register */
   sched Rfrom;        /* cycle to receive from (mod beat period) */
   sched cnt;          /* counter value */
   sched checked;      /* Flag to tell if cycle has already been tried. */

   int  perf;          /* Performance as a transmitter. */

   struct systype *other;      /* pointer to the other system  */
} systype;


/* These macros are the same as the primitive macros but
 * work with systype structs instead.  Note the capitalization of the names.
 */
```

```
#define Mf(M)               (M->f)
#define Nf(M)               ((M->other)->f)

#define Beatperiod(M)       Pbeatperiod( Mf(M), Nf(M) )
#define Period(M)           Pperiod( Mf(M), Nf(M) )
#define Freq(M)             Pfreq( Mf(M), Nf(M) )
#define DeltaT(M)           PdeltaT( Mf(M), Nf(M) )

#define Clockedge(M,i)      Pclockedge( Mf(M), Nf(M), (i) )
#define Cycle(M,t)          Pcycle( Mf(M), Nf(M), (t) )
#define SHcycle(M,t,dt)     Pshcycle( Mf(M), Nf(M), (t), (dt) )

#define Fractodt(M,num)     Pfractodt( Mf(M), Nf(M), (num) )
#define Delttodt(M,num)     Pdelttodt( Mf(M), Nf(M), (num) )

#define Tptodt(M,tp)        Ptptodt( Mf(M), Nf(M), (tp) )
#define SHedge(M,i,tp)      Pshedge( Mf(M), Nf(M), (i), (tp) )

#define NxtOcycle(M,t)      PnxtOcycle( Mf(M), Nf(M), (t) )
#define PrvOcycle(M,t)      PprvOcycle( Mf(M), Nf(M), (t) )
#define NxtOshcycle(M,t,dt) PnxtOshcycle( Mf(M), Nf(M), (t), (dt) )
#define PrvOshcycle(M,t,dt) PprvOshcycle( Mf(M), Nf(M), (t), (dt) )


/* The following macros allow us to access the different timing params
 * by name.  ('TP' stands for timing parameter.)  Valid values for S are
 * S, H, C, and P.  Note that these operations won't work as desired
 * if implemented as functions, and not macros.
 */

#define TP(M,S)                   Tptodt( M, (M->S) )
#define TPedge(M,i,S)             SHedge( M, (i), (M->S) )
#define TPcycle(M,t,S)            SHcycle( M, (t), TP(M,S) )

   /* These macros find the appropriate cycle corresponding to time t if
    * t is a timing parameter edge S.
    */

#define _NxtOTPcycle(M,t,S)       NxtOshcycle( M, (t), TP((M->other),S) )
#define _PrvOTPcycle(M,t,S)       PrvOshcycle( M, (t), TP((M->other),S) )

   /* NxtOTPcycle returns j when asked to find the next Sn edge j of the
    * other system at or after the Sm edge i of the original system (M).
    * PrvOTPcycle is similar but returns the edge at or _before_.
    */

#define NxtOTPcycle(M,i,Sm,Sn)    _NxtOTPcycle( M, TPedge(M,i,Sm), Sn )
#define PrvOTPcycle(M,i,Sm,Sn)    _PrvOTPcycle( M, TPedge(M,i,Sm), Sn )



/*
 * Global Variables
 *
 */

systype M, N;
long fM, fN;
long Sm, Hm, Cm, Pm;
long Sn, Hn, Cn, Pn;
int quit = 0;

unsigned char ROMimage[65536];



/*
 * Basic Output Routines
 *
 */

void change_outfile( char *filename )
{
      fclose( outfile );
      outfile = fopen( filename, "wt" );
}


void show( char *fmt, ... )
{
   va_list args;

   va_start( args, fmt );
   vprintf( fmt, args );
   if ( (outmode == TEXT) || (outmode == ROM) ) {
      vfprintf( outfile, fmt, args );
   }
   va_end( args );
}

void fshow( char *fmt, ... )
{
   va_list args;

   va_start( args, fmt );
   vfprintf( outfile, fmt, args );
   va_end( args );
}

void show2( char *fmt, ... )
{
   va_list args;

   va_start( args, fmt );
   vprintf( fmt, args );
   vfprintf( outfile, fmt, args );
   va_end( args );
}
```

```
/*
 * Initialization Routines
 *
 */

void ClearROM()
{
   long i;

   for ( i=0; i < 65536; i++ )
      ROMimage[i] = 0xFF;
   for ( i=0; i < (1 << 12); i++ )
      ROMimage[i] = 0x55;
}

/*
 * ClearSched clears the schedule tables used for
 *   transmitting from M to N.
 */

void ClearSched( systype *M, systype *N )
{
   int i;

   for (i=0; i < MAXMN; i++) {
      M->TE[i] = 0;    M->Tsel[i] = -1;   M->TD[i] = -1;
      N->RE[i] = 0;    N->Rsel[i] = -1;   N->RQ[i] = -1;
      M->checked[i] = 0;
      N->checked[i] = 0;
   }
}

/*
 * initcnt fills the cnt array with the proper counter values
 */

void initcnt( systype *M )
{
   int i;

   for (i=0; i < MAXMN; i++) {
      M->cnt[i] = i % (M->f);
   }
}

void initMN( systype *M, long fM, long Sm, long Hm, long Cm, long Pm,
             systype *N, long fN, long Sn, long Hn, long Cn, long Pn )
{
   int i;

   M->f = fM;
   M->other = N;
   M->S = -Sm;
   M->H = Hm;
   M->C = Cm;
   M->P = Pm;

   N->f = fN;
   N->other = M;

   N->S = -Sn;
   N->H = Hn;
   N->C = Cn;
   N->P = Pn;

   ClearSched( M, N );
   initcnt( M );
   ClearSched( N, M );
   initcnt( N );
}



/*
 * Scheduler Algorithms
 *
 */

/* No Flow Control.  Always use register 0. */

int UnctlTransmit( systype *M, systype *N )
{
   int i;

   for (i = 0; i < M->f; i++) {
         M->TE[i] = 1;
         M->Tsel[i] = 0;
   }
   for (i = 0; i < N->f; i++) {
         N->RE[i] = 1;
         N->Rsel[i] = 0;
   }
   return( 0 );
}

/*
 * Original single-buffer greedy algorithm.
 *
 * This is adapted from Pratt and Ward's original algorithm, described
 * in section 3.3 of my thesis.
 *
 */

int _OrigTransmit( systype *M, systype *N, int begin )
{
   long i, j;
   long i1, j1, nexti;
```

```
int count = 0;
long Pi, Sj, Hj, Ci;
long tcur, tbegin;


      /* STEP 2: Begin at Pbegin.
       */

i = begin;

Pi = TPedge(M,i,P);
Ci = TPedge(M,i,C);
tbegin = Pi + Beatperiod(M);    /* Note that we're _not_ using modulo math */
tcur = Pi;
   if (PEEK) {
      show( "\nBEGIN: i = %ld, Ci = %ld, Pi = %ld \n", i, Ci, Pi );
   }

do {
      /* STEP 3: Pi -> Sj.  Go to next Sj.
       */

   j = NxtOTPcycle(M,i,P,S);
   Sj = TPedge(N,j,S);
   tcur = Sj;
      if (PEEK) {
         Hj = TPedge(N,j,H);
         show( "j = %ld, Sj = %ld, Hj = %ld \n", j, Sj, Hj );
      }

      /* STEP 4: Go to Hj.
       */

   Hj = TPedge(N,j,H);
   tcur = Hj;

      /* STEP 5: Hj -> Ci'.  Go to next Ci.
       */

   nexti = NxtOTPcycle(N,j,H,C);
   Ci = TPedge(M,nexti,C);
   tcur = Ci;
      if (PEEK) {
         Pi = TPedge(M,nexti,P);
         show( "i' = %ld, Ci' = %ld, Pi' = %ld \n", nexti, Ci, Pi );
      }

      /* STEP 6: Go to next Pi and check.
       */

   Pi = TPedge(M,nexti,P);
   tcur = Pi;

      /* STEP 8: If Pbegin is not passed, mark the appropriate
       * schedule entries.
```

```
         */
      if ( tcur <= tbegin )  {
         i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

         M->TE[i1] = 1;
         M->Tsel[i1] = 0;
         M->TD[i1] = count % 0x10;
         M->Tto[mod(i,M->f)] = mod(j,N->f);

         N->RE[j1] = 1;
         N->Rsel[j1] = 0;
         N->RQ[mod(j,N->f)] = count % 0x10;
         N->Rfrom[mod(j,N->f)] = mod(i,N->f);

         count++;

          if (PEEK) {
             show( "TE[%d] and RE[%d] set. \n",
                     mod(i-1,M->f), mod(j-1,N->f) );
          }
      }

         /* STEPS 7 and 9: If you pass or hit Pbegin, then quit, else
          * make i = nexti and repeat the process.
          */

      i = nexti;

   } while (tcur < tbegin);

   return( count );
}

/* _Transmit schedules a transmission from M -> N and returns
 * the number of successful transmissions in a coincidence cycle.
 */

int _Transmit( systype *M, systype *N )
{
   int count, begin;
   int bestcount, bestbegin;
   int best;
   int found;

   best = min( M->f, N->f );
   bestcount = 0;
   bestbegin = 0;
   begin     = 0;

   while ( (bestcount != best) && (begin < M->f) ) {
      ClearSched( M, N );
      count = _OrigTransmit( M, N, begin );
      if (count > bestcount) {
         bestcount = count;
         bestbegin = begin;
      }
      begin++;
   }

   /* get lowest bestbegin. (if bestcount=best, then bestbegin
    * will also be the lowest bestbegin.)
    */

   if (bestcount != best) {
      ClearSched( M, N );
      count = _OrigTransmit( M, N, bestbegin );
   }
   return count;
}


/*
 * Lazy Double-Buffered Algorithm
 *
 */

int _DBuffLTransmit( systype *M, systype *N )
{
   long x, y;
   long eold, e;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
   int  c, count = 0;
   int  slowtx;

   slowtx = M->f <= N->f;

   if (slowtx) {
      m = Period(N);
      n = Period(M);
   } else {
      m = Period(M);
      n = Period(N);
   }

   P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
   S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

   x = 1;
   if ( slowtx )
      f0 = -( P - S );  /* Remember that S here is -S of S in the paper. */
   else
      f0 = -( H - C );
   y = ndiv( f0, n ) + 1;
   eold = mod( f0, n ) - n;
   R = 0;
      if (PEEK) {
         show( "f0: %ld, e0: %ld, x0: %ld, y0: %ld\n", f0, eold, x, y );
      }
   for (c=0; c < max(M->f,N->f); c++) {
      e = eold + m;
         if (PEEK) {
            show( "eold: %ld, e: %ld, x: %ld, y: %ld\n", eold, e, x, y );
         }
      if ( e >= 0 ) {
         if (slowtx) {
            i = y; j = x;
         } else {
            i = x; j = y;
         }

         i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

         M->TE[i1] = 1;
         M->Tsel[i1] = R;
         M->TD[i1] = count % 0x10;
         M->Tto[mod(i,M->f)] = mod(j+2,N->f);

         N->RE[j1] = 1;
         N->Rsel[j1] = R;
         N->RQ[mod(j+2,N->f)] = count % 0x10;
         N->Rfrom[mod(j+2,N->f)] = mod(i,M->f);

         R = (~R) & 1;
         y = y + 1;
         e = e - n;

         count++;

      }
      x = x + 1;
      eold = e;
   }

   return( count );
}


/*
 * Lazy Double-Buffered Algorithm
 *
 */

int _DBuffGTransmit( systype *M, systype *N )
{
   long x, y;
   long eold, e;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
```

```
int  c, count = 0;
int  slowtx;

slowtx = M->f <= N->f;

if (slowtx) {
   m = Period(N);
   n = Period(M);
} else {
   m = Period(M);
   n = Period(N);
}

P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

x = 1;
if ( !slowtx )
   x = x - 1;            /* For greedy algo, x = x_0 - 1 = 0 */
if ( slowtx )
   f0 = -( P - S );
else
   f0 = -( S - P );      /* The greedy algo transformation */
y = ndiv( f0, n ) + 1;
eold = mod( f0, n ) - n;
if (!slowtx) {           /* Adjust e0 and y0 for greedy algo */
   if ( mod(f0,n) == 0 )
      y = y - 1;
   if ( eold == -n )
      eold = 0;        /*******/
}
R = 0;
   if (PEEK) {
      show( "f0: %ld, e0: %ld, x0: %ld, y0: %ld\n", f0, eold, x, y );
   }
for (c=0; c < max(M->f,N->f); c++) {
   e = eold + m;
      if (PEEK) {
         show( "eold: %ld, e: %ld, x: %ld, y: %ld\n", eold, e, x, y );
      }
   if (( slowtx && (e >= 0) ) || ( !slowtx && (e > 0) )) {
      if (slowtx) {
         i = y; j = x;
      } else {
         i = x; j = y;
      }

      i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

      M->TE[i1] = 1;
      M->Tsel[i1] = R;
      M->TD[i1] = count % 0x10;
      M->Tto[mod(i,M->f)] = mod(j,N->f);

      N->RE[j1] = 1;
```

```
         N->Rsel[j1] = R;
         N->RQ[mod(j,N->f)] = count % 0x10;
         N->Rfrom[mod(j,N->f)] = mod(i,M->f);

         R = (~R) & 1;
         y = y + 1;
         e = e - n;

         count++;

      }
      x = x + 1;
      eold = e;
   }

   return( count );
}


/*
 * Generalized Algorithm
 *
 */

int _GBuff2Transmit( systype *M, systype *N )
{
   long x, y;
   long eold, e, emax, emin;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
   int  c, count = 0;
   int  slowtx;

   slowtx = M->f <= N->f;
/*                        Uncomment this to test slower system's algo.
   if (slowtx)
      _slowGBuff2Transmit( M, N );
   else {
*/
   if (slowtx) {
      m = Period(N);
      n = Period(M);
   } else {
      m = Period(M);
      n = Period(N);
   }

   P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
   S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

   emax = n - ( (P-C) + (H-S) );
   emin = m - n + ( (P-C) + (H-S) );
   x = 1;
   if ( !slowtx )
      x = x - 1;             /* For greedy algo, x = x_0 - 1 = 0 */
   if ( slowtx )
      f0 = -( P - S );
   else
      f0 = -( S - P );       /* The greedy algo transformation */
   y = ndiv( f0, n ) + 1;
   eold = mod( f0, n ) - n;
   if (!slowtx) {            /* Adjust e0 and y0 for greedy algo */
      if ( mod(f0,n) == 0 )
         y = y - 1;
      if ( eold == -n )
         eold = 0;    /*********/
   }

   R = ( slowtx ? 0 : 1 );

   for (c=0; c < max(M->f,N->f); c++) {
      e = eold + m;
      if (( slowtx && (e >= 0) ) || ( !slowtx && (e > 0) )) {
         if (slowtx) {
            i = y; j = x;
         } else {
            i = x; j = y;
         }

         i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

         if ( !slowtx )
            if ( R == 0 )
               R = ( (e < emin) ? 1 : 0 );
            else
               R = 0;

         M->TE[i1] = 1;
         M->Tsel[i1] = R;
         M->TD[i1] = count % 0x10;
         M->Tto[mod(i,M->f)] = mod(j,N->f);

         N->RE[j1] = 1;
         N->Rsel[j1] = R;
         N->RQ[mod(j,N->f)] = count % 0x10;
         N->Rfrom[mod(j,N->f)] = mod(i,M->f);

/* Adjust count only to measure single-buffered performance.
 * Note that as a consequence of this, TD would not be incremented
 * whenever TSel is 1.
 */
         if ( R == 0 )
            count++;

         if ( slowtx )
            if ( R == 0 )
            R = ( (e > emax) ? 1 : 0 );
         else
            R = 0;

      y = y + 1;
      e = e - n;
   }

   x = x + 1;
   eold = e;
}
/* }                       Uncomment this to test slower system's algo.
*/
return( count );
}

/*
 * Slower system's algorithm
 *
 */

int _slowGBuff2Transmit( systype *M, systype *N )
{
   long x, y;
   long eold, e, emax, emin;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
   int  c, count = 0;
   int  slowtx;
   long dMN;

/* assume M < N */

      m = Period(N);
      n = Period(M);
      dMN = n % m;

   P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
   S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

   emax = n - ( (P-C) + (H-S) );
   emin = m - n + ( (P-C) + (H-S) );

   f0 = -( P - S );
   y = ndiv( f0, n ) + 1;

   eold = mod( f0, n ) - n;
   e = mod(eold,m);

   R = 0;

   for (c=0; c < M->f; c++) {
        i = y; j = x;
        i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

        M->TE[i1] = 1;
        M->Tsel[i1] = R;
        M->TD[i1] = count % 0x10;
        M->Tto[mod(i,M->f)] = mod(j,N->f);

        if ( R == 0 )
           count++;

        if ( slowtx )
           if ( R == 0 )
              R = ( (e > emax) ? 1 : 0 );
           else
              R = 0;

        e = e - dMN;
        if ( e < 0 )
           e = e + m;

        y = y + 1;
   }

   return( count );
}


/*
 * Higher-level Scheduling Routines
 *
 */

int Transmit( systype *M, systype *N )
{
   ClearSched( M, N );
   switch( algorithm ) {
      case NONE : return( UnctlTransmit( M, N ) );
      case ORIG : return( _Transmit( M, N ) );
      case DBUFFL : return( _DBuffLTransmit( M, N ) );
      case DBUFFG : return( _DBuffGTransmit( M, N ) );
      case GBUFF2 : return( _GBuff2Transmit( M, N ) );
      default : return( 0 );
   }
}

int MakeSched( systype *M, systype *N )
{
   int a, b, c, opt;

   opt = min( M->f, N->f );

      if (PEEK) {
         show("M -> N\n");
      }
   a = Transmit( M, N );
      if (PEEK) {
         show("\nN -> M\n");
      }
   b = Transmit( N, M );

   M->perf = ((double) a / (double) opt) * 100;
   N->perf = ((double) b / (double) opt) * 100;

   c   = min( M->perf, N->perf );

   return( c );
}



/*
 * Display Routines
 *
 */

void PlainSched( systype *M, int *schedptr )
{
   int i;

   for (i=0; i < M->f; i++)
      if (schedptr[i] == (-1))
         show("x");
      else
         show("%X",schedptr[i]);
   show("\n");
}


void DrawSched( systype *M, int *schedptr )
{
   int i, j;
   long Mw, Nw, MNgcd;

   Mw = Nf(M);
   Nw = Mf(M);
   MNgcd = gcd( Mw, Nw );
   Mw /= MNgcd;
   Nw /= MNgcd;

   if ((Mw == 1) || (Nw == 1))
      Mw = 2*Mw;

   show("|");
   for( i = 0; i < M->f; i++ ) {
      for( j = 0; j < Mw-1; j++ ) {
         if (j == (Mw-1)/2 )
            if (schedptr[i] == (-1))
               show("x");
            else
```

```
                show("%X", schedptr[i] );
            else
               show(" ");
         }
         show("|");
      }
      show("\n");
}

#define PrintSched(M,sptr)   ( (DRAWSCHEDS) ? DrawSched(M,sptr) : \
                                              PlainSched(M,sptr)   )


void ShowTimes( systype *M, systype *N )
{
   show("\n");
   show( "Tref = %ld\n", Tref );
   show("TM:  TM = %ld, tS = %ld,  tH = %ld,  tC = %ld,  tP = %ld\n",
            Period(M), TP(M,S), TP(M,H), TP(M,C), TP(M,P));
   show("TN:  TN = %ld, tS = %ld,  tH = %ld,  tC = %ld,  tP = %ld\n",
            Period(N), TP(N,S), TP(N,H), TP(N,C), TP(N,P));
   show("\n");
}

void ShowSched( systype *M, systype *N )
{
   int i;

   show("\n");
   show("M:  f = %ld, S = %ld,  H = %ld,  C = %ld,  P = %ld\n",
            M->f, M->S, M->H, M->C, M->P);
   show("N:  f = %ld, S = %ld,  H = %ld,  C = %ld,  P = %ld\n",
            N->f, N->S, N->H, N->C, N->P);
   show("\n");

   show("M -> N\n");
   show("---------\n");
   show("N.RE   : ");
   PrintSched( N, N->RE );
   show("M.TE   : ");
   PrintSched( M, M->TE );
   show("---------\n");
   show("N.Rsel : ");
   PrintSched( N, N->Rsel );
   show("M.Tsel : ");
   PrintSched( M, M->Tsel );
   show("---------\n");
   show("N.RQ   : ");
   PrintSched( N, N->RQ );
   show("M.TD   : ");
   PrintSched( M, M->TD );

   show("\n");

   show("N -> M\n");
   show("--------\n");
   show("N.TE   : ");
   PrintSched( N, N->TE );
   show("M.RE   : ");
   PrintSched( M, M->RE );
   show("--------\n");
   show("N.Tsel : ");
   PrintSched( N, N->Tsel );
   show("M.Rsel : ");
   PrintSched( M, M->Rsel );
   show("--------\n");
   show("N.TD   : ");
   PrintSched( N, N->TD );
   show("M.RQ   : ");
   PrintSched( M, M->RQ );
   show("--------\n");

   show("Performance:  M->N = %d%%, N->M = %d%% \n", M->perf, N->perf );
}

void ROMsched( systype *M, systype *N )
{
   long baseaddr, offs;
   long Mi, Nj;
   int  REM, RselM, TE0M, TE1M;
   int  REN, RselN, TE0N, TE1N;

#define nneg( x ) ( ((x) < 0) ? 0 : x )
#define bit0( x ) ( (x) & 1 )


   baseaddr = ((Mf(M)-1) << (ROM_B*2))  + ((Mf(N)-1) << ROM_B);

   for (offs=0; offs < ROM_N; offs++) {
      Mi = (usewithchip ? (Mf(M) - offs + 1) : offs);
      Nj = (usewithchip ? (Mf(N) - offs + 1) : offs);
      Mi = mod( Mi, Mf(M) );
      Nj = mod( Nj, Mf(N) );

      TE0M = bit0( M->TE[Mi] & ~(nneg(M->Tsel[Mi])) );
      TE1M = bit0( M->TE[Mi] & nneg(M->Tsel[Mi]) );
      REM  = bit0( M->RE[Mi] );
      RselM = bit0( nneg( M->Rsel[Mi] ) );
      TE0N = bit0( N->TE[Nj] & ~(nneg(N->Tsel[Nj])) );
      TE1N = bit0( N->TE[Nj] & nneg(N->Tsel[Nj]) );
      REN  = bit0( N->RE[Nj] );
      RselN = bit0( nneg( N->Rsel[Nj] ) );

      ROMimage[baseaddr+offs] = TE0M + (TE1M << 1)
                              + (REN << 2) + (RselN << 3)
                              + (TE0N << 4) + (TE1N << 5)
                              + (REM << 6) + (RselM << 7);
   }
}

void IntelMCS( systype *M, systype *N )
{
   long baseaddr, offs;
   long chksum = 0;

   baseaddr = ((Mf(M)-1) << (ROM_B*2))  + ((Mf(N)-1) << ROM_B);

   show( ":%02X%04X00", ROM_N, baseaddr );
   chksum = ROM_N + (baseaddr >> 8) + (baseaddr & 0xFF);
   for (offs=0; offs < ROM_N; offs++) {
       show( "%02X", ROMimage[baseaddr+offs] );
       chksum += ROMimage[baseaddr+offs];
   }
   chksum = (-chksum) & 0xFF;
   show( "%02X\n", chksum );
}

void LaTeXsched( systype *M, int *schedptr, int ypos )
{
   int i;
   char s[10];

#define entry(x,y,s) fshow("\\put(%d,%d){\\makebox(%d,2){%s}}\n", x, y, \
                           M->other->f, s )

   for (i=0; i < M->f; i++) {

      if (schedptr[i] == (-1))
         strcpy( s, "*" );
      else
         sprintf( s, "%d", schedptr[i] );

      entry( i*(M->other->f), ypos, s );
   }
}

void LaTeXvals( systype *M, systype *N )
{

#define label(y,s) fshow( "\\put(-1,%d){\\makebox(0,0)[r]{$%s$}}\n", y, s )
#define Lsched( y, sys, S ) label( (y+1), #S ); LaTeXsched( sys, sys->S, y );

   Lsched( 13, N, RE );
   Lsched( 15, N, Rsel );
   Lsched( 17, N, RQ );

   Lsched( -4, M, TE );
   Lsched( -6, M, Tsel );
   Lsched( -8, M, TD );

}
```

```
void LaTeXparams( systype *M, systype *N )
{
   double Mf = M->f;
   double Nf = N->f;

#define set( x, v )   fshow( "\\renewcommand{\\" #x "}{%g}\n", v )
#define gdttodelt( x ) ((double)(x) / (double) d )

   set( M, Mf );
   set( Mo, Mf+1 );
   set( nM, -Mf );

   set( N, Nf );
   set( No, Nf+1 );
   set( nN, -Nf );

   set( MN, Mf*Nf );

   set( CM, gdttodelt(TP(M,C)) );
   set( PM, gdttodelt(TP(M,P)) );
   set( CPM, gdttodelt( TP(M,P) - TP(M,C) ) );
   set( SN, gdttodelt( -TP(N,S) ) );
   set( NSN, Mf+gdttodelt( (TP(N,S)) ) );
   set( HN, gdttodelt( TP(N,H) ) );


}

void LaTeXdiag( systype *M, systype *N )
{
/* Note: This will fclose the current outfile. */
   change_outfile( "sched.tex" );

   fshow( "\\documentstyle{article}\n"
          "\\begin{document}\n"
          "\\setlength{\\unitlength}{.1in}\n"
          "\\newcounter{cM}\n"
          "\\newcounter{cN}\n"
          "\\input{setup}\n"         );
   LaTeXparams( M, N );
   fshow( "\\begin{picture}(%d,31)(-11,-9)\n"
          "\\input{clocks}\n",
          (M->f)*(N->f)+11        );
   LaTeXvals( M, N );
   fshow( "\\end{picture}\n"
          "\n"        );
   LaTeXparams( N, M );
   fshow( "\\begin{picture}(%d,27)(-11,-7)\n"
          "\\input{clocks}\n",
          (M->f)*(N->f)+11        );
   LaTeXvals( N, M );
   fshow( "\\end{picture}\n"
          "\\end{document}\n"        );
   fclose( outfile );
```

```
/* Use "at" here to resume writing log file. */
   outfile = fopen( outfilename, "at" );
}

void singlesched( systype *M, systype *N )
{
   MakeSched( M, N );
   ShowSched( M, N );
   if ( outmode == LDIAG )
      LaTeXdiag( M, N );
   else if ( outmode == ROM ) {
           ROMsched( M, N );
           IntelMCS( M, N );
        }
}

void tablesched( systype *M, systype *N )
{
   int oldM, oldN;
   int tx, rx, maxfreq;
   int LaTeX = ((outmode == LDIAG) || (outmode == LPLOT));

   oldM = M->f; oldN = N->f;
   maxfreq = max( oldM, oldN );

   if ( LaTeX ) {
      fshow( "{\\scriptsize\n"  );
      fshow( "\\begin{tabular}{l|" );
      for (rx=1;rx<=maxfreq;rx++)
         fshow( "r" );
      fshow( "}\nTransmit & \\multicolumn{%d}{c}{Receive Frequency} \\\\ \n",
             maxfreq );
   } else
      show("Transmit\tReceive Frequency\n");

   show2( "Freq." );
   for (rx=1;rx<=maxfreq;rx++) {
      if ( LaTeX )
          fshow( " & " );
      show( "\t");
      show2( "%d", rx );
   }
   if ( LaTeX )
      fshow( " \\\\ \\hline" );
   show2("\n" );
   for (tx = 1; tx <= maxfreq; tx++) {
      show2( "%d", tx );
      if ( LaTeX )
         fshow( " & " );
      show( "\t" );
      for (rx = 1; rx <= maxfreq; rx++) {
         M->f = tx; N->f =rx;
         MakeSched( M, N );
         show2("%d", M->perf );
         if (rx < maxfreq) {
            if ( LaTeX )
               fshow( " & " );
            show("\t");
         } else {
            if ( LaTeX )
               fshow( " \\\\\n" );
            show("\n");
         }
      }
   }
   if ( LaTeX )
      fshow( "\\end{tabular}\n}" );
   show( "\n" );
   M->f = oldM; N->f = oldN;
}

void ROMtable( systype *M, systype *N, long seg )
{
   int oldM, oldN;
   int tx, rx, maxfreq;
   long chksum;

   oldM = M->f; oldN = N->f;
   maxfreq = max( oldM, oldN );

   chksum = 2 + 2 + (seg >> 8) + (seg & 0xFF);
   chksum = (-chksum) & 0xFF;
   show( ":02000002%04X%02X\n", seg, chksum );

   for (tx = 1; tx <= maxfreq; tx++) {
      for (rx = 1; rx <= maxfreq; rx++) {
         M->f = tx; N->f =rx;
         MakeSched( M, N );
         ROMsched( M, N );
         IntelMCS( M, N );
      }
   }

   show( ":00000001FF\n" );
}

void bigROMtable( systype *M, systype *N )
{
   int algo;
   int oldalgo = algorithm;

   for (algo = 0; algo <= MAXALGO; algo++ ) {
      algorithm = algo;
      ROMtable( M, N, (algo << (3*ROM_B - 4)) );
   }
   algorithm = oldalgo;
}
```

```
/*
 * User Interface (Menu) Data Structures and Routines
 *
 * This is a very simple menu system.  I'm tempted to use C++
 * here, but I'll keep it simple to make it portable.
 *
 */

#define MAXNAME  50
#define MAXITEMS 9

typedef struct procitem {
   char name[MAXNAME];      /* name of the item */
   char value[MAXNAME];     /* string to be printed beside the name */
   void (*proc)(char *);    /* procedure to execute when the item is chosen */
} procitem;

typedef struct procmenu {
   char name[MAXNAME];       /* name of menu */
   int  numitems;            /* number of items in menu */
   procitem *item[MAXITEMS]; /* array of pointers to items */
} procmenu;

typedef char valitem[MAXNAME]; /* valitem is just a string with the name */

typedef struct valmenu {
   char name[MAXNAME];      /* name of menu */
   int  *variable;          /* pointer to the variable to be changed */
   int  numitems;           /* number of items in menu */
   valitem item[MAXITEMS];  /* array of valitems (array of strings)  */
} valmenu;


/*
 * General ValMenu Routine
 *
 */

void do_valmenu( valmenu *M )
{
   char s[10];
   int i, choice;
   int quit = 0;

   printf( "\n%s\n\n", M->name );
   for ( i=0; i < (M->numitems); i++ )
      printf( "%d) %s\n", i+1, M->item[i] );
   do {
      choice = *(M->variable);
      printf( "\nEnter your choice (1-%d) [%d]: ", M->numitems+1,
              choice );
      gets( s );
      sscanf( s, "%d", &choice );
   } while ((choice < 1) || (choice > (M->numitems)));
   *(M->variable) = choice;
```

```
}


/*
 * ValMenu definitions
 *
 */

valmenu PUnits = { "Units for Timing Parameters", &paramunits, 5,
                   { "dt",
                     "delta t",
                     "1/d of clock period",
                     "Absolute units (clk div)",
                     "Absolute units (clk mul)",
                     "", "", "", ""
                   }
                 };

valmenu CompMode = { "Compute Mode", &compmode, 2,
                   { "Pair",
                     "Table",
                     "",
                     "", "", "", "", "", ""
                   }
                 };

valmenu OutMode = { "Output Format", &outmode, 4,
                    { "Text",
                      "LaTeX schedule diagram",
                      "LaTeX schedule plot",
                      "ROM",
                      "", "", "", "", ""
                    }
                  };

valmenu Algo = { "Scheduling Algorithm", &algorithm, 4,
                 { "Single Buffered",
                   "Gen. Online (2 buffers)",
                   "Double Buffered (Greedy)",
                   "Double Buffered (Lazy)",
                   "", "", "", "", ""
                 }
               };


/*
 * General ProcMenu Routine
 *
 */

void do_procmenu( procmenu *M )
{
   char s[10];
   int i, choice;
   int quit = 0;
   do {
      printf( "\n>>> %s <<<\n\n", M->name );
      for ( i=0; i < (M->numitems); i++ ) {
         printf( "%d) %s", i+1, M->item[i]->name );
         if ( M->item[i]->value[0] != 0 )
            printf( "[%s]", M->item[i]->value );
         printf( "\n" );
      }
      do {
         printf( "\nEnter your choice (1-%d, <cr> to go back) : ",
                 M->numitems );
         gets( s );
         if ( (s[0] == 0) )
            choice = (M->numitems+1);
         else
            sscanf( s, "%d", &choice );
      } while ((choice < 1) || (choice > (M->numitems+1)));
      printf("\n\n");
      if (choice != M->numitems+1)
         (*(M->item[choice-1]->proc))( M->item[choice-1]->value );
   } while (choice != M->numitems+1);
}


/*
 * Input Routines
 *
 * These routines as the user for input.  Pressing CR keeps the variable's
 * old value.
 *
 */

char YN[2] = "ny";

void getYN( char *s, int *x )
{
   char temp[20];

   temp[0] = 0;
   printf( "%s (y/n)? [%c] ", s, YN[*x] );
   gets( temp );

   if ( temp[0] == 'y' )
      *x = 1;
   else if ( temp [0] == 'n' )
           *x = 0;
}

int askYN( char *s, int defval )
{
   int x = defval;

   getYN( s, &x );
   return( x );
```

```
}

void getstring( char *msg, char *s )
{
   char temp[40];

   printf( "%s [%s]: ", msg, s );
   gets( temp );
   if ( temp[0] != 0 )
      strcpy( s, temp );
}

void getlong( long *x )
{
   char temp[20];

   gets( temp );
   sscanf( temp, "%ld", x );
}

#define getTP(x)   printf(#x " [%ld] : ", x);  \
                  getlong(&x);


/*
 * Menu item procedures
 *
 */

void do_Params( char *s )
{
   getTP(d);

   printf( "\nM, N [%ld, %ld] : ", fM, fN );
   gets( s );
   sscanf( s, "%ld, %ld", &fM, &fN );

   printf( "\nEnter the following timing parameters in units of %s:\n\n",
           PUnits.item[paramunits-1] );

   if ( (paramunits == CMULPS) || (paramunits == CDIVPS)) {
      getTP(Tref);
   }

   getTP(Sm); getTP(Hm); getTP(Cm); getTP(Pm);
   getTP(Sn); getTP(Hn); getTP(Cn); getTP(Pn);

/* Note: this call uses the _global_ variables M and N. */
   initMN( &M, fM, Sm, Hm, Cm, Pm, &N, fN, Sn, Hn, Cn, Pn );

      if (PEEK) {
         ShowTimes( &M, &N );
      }
   s[0] = 0;
}
```

```
void converttodt( systype *M, systype *N, int oldunits )
{
   paramunits = oldunits;

   Sm = -TP(M,S);   Hm = TP(M,H);   Cm = TP(M,C);   Pm = TP(M,P);
   Sn = -TP(N,S);   Hn = TP(N,H);   Cn = TP(N,C);   Pn = TP(N,P);

   paramunits = DTUNITS;

   initMN( M, fM, Sm, Hm, Cm, Pm, N, fN, Sn, Hn, Cn, Pn );

      if (PEEK) {
         ShowTimes( M, N );
      }
}


void do_PUnits( char *s )
{
   int oldunits = paramunits;

   do_valmenu( &PUnits );
   if ((paramunits != oldunits)) {
      if (paramunits == DTUNITS) {
         if ( askYN( "Convert to dt units?", 0 ) ) {
            printf( "\nConverting to dt units ... " );
            converttodt( &M, &N, oldunits );
            printf( "\n\n" );
         }
      }
   }
   s[0] = 0;
}

void do_CompMode( char *s )
{
   do_valmenu( &CompMode );
   s[0] = 0;
}

void do_OutMode( char *s )
{

   do_valmenu( &OutMode );
   s[0] = 0;
}

void do_Algo( char *s )
{
   do_valmenu( &Algo );
   s[0] = 0;
}

void do_Output( char *s )
```

```
{
   if (compmode == TABLE) {
      if ( outmode == ROM )
         bigROMtable( &M, &N );
      else
         tablesched( &M, &N );
   }
   else
      singlesched( &M, &N );
   s[0] = 0;
   fflush( outfile );
}

void do_OutOpts( char *s )
{
   char temp[41];
   strcpy( temp, outfilename );

   getstring( "Enter Log File Name", outfilename );
   if (strcmp(outfilename, temp)) {
      change_outfile( outfilename );
   }

   getYN( "Show intermediate results", &PEEK );
   getYN( "Draw text schedule diagram", &DRAWSCHEDS );

   getYN( "Use with prototype chip", &usewithchip );
   s[0] = 0;
}

/*
 * ProcItem definitions
 *
 */

procitem I_PUnits = { "Parameter Units", "", do_PUnits };
procitem I_Algo = { "Scheduling Algorithm", "", do_Algo };
procitem I_CompMode = { "Compute Mode", "", do_CompMode };
procitem I_OutMode = { "Output Format", "", do_OutMode };
procitem I_OutOpts = { "Output Options", "", do_OutOpts };

/* Hmm ... this is messy but I need to define this before defining
 * do_Options.
 */

procmenu OptionsMenu = { "Options Menu", 5,
                         { &I_PUnits, &I_Algo, &I_CompMode,
                           &I_OutMode, &I_OutOpts, NULL,
                           NULL, NULL, NULL
                         }
                       };

void do_Options( char *s )
{
   do_procmenu( &OptionsMenu );
   s[0] = 0;
}



/*
 * ProcMenu definitions
 *
 */

procitem I_Params  = { "Enter Parameters", "", do_Params };
procitem I_Output  = { "Generate Output", "", do_Output };
procitem I_Options = { "Options", "", do_Options };


procmenu MainMenu = { "Main Menu", 3,
                      { &I_Params, &I_Output, &I_Options,
                        NULL, NULL, NULL, NULL, NULL, NULL
                      }
                    };



/*
 * Main Program Routines
 *
 */

void initdisplay()
{
   ClearROM();
   outfile = fopen( outfilename, "wt" );
}

void shutdown()
{
   fclose( outfile );
}


main()
{
   initdisplay();
   do {
      do_procmenu( &MainMenu );
      getYN( "Do you really want to quit", &quit );
   } while (!quit);
   shutdown();
}
```

B.4 LaTeX Support Files

The *sched.tex* file produced by the scheduling software requires two support files: *setup.tex* and *clocks.tex*.

```
% setup.tex
%
\newcommand{\M}{5}%
\newcommand{\Mo}{6}%
\newcommand{\nM}{-5}%
%
\newcommand{\N}{6}%
\newcommand{\No}{7}%
\newcommand{\nN}{-6}%
%
\newcommand{\MN}{30}%
%
\newcommand{\CM}{1}%
\newcommand{\PM}{3}%
\newcommand{\CPM}{2}%
\newcommand{\SN}{1}%
\newcommand{\NSN}{4}  %  >>>> Don't forget this!  Set to (M-SN) %
\newcommand{\HN}{1}%
```

```
% clocks.tex
%
\setcounter{cM}{0}
\setcounter{cN}{0}
%
% Draw M cycles
\multiput(0,-1)(0,3){2}{\line(1,0){\MN}}
\multiput(0,-1)(\N,0){\Mo}{\line(0,1){3}}
\multiput(0,1.5)(1,0){\MN}{\line(0,1){.5}}
\multiput(0,-1)(\N,0){\M}{\makebox(\N,2.5){\arabic{cM}}\addtocounter{cM}{1}}
%
% Draw C edges
\multiput(\CM,3)(\N,0){\M}{\line(0,1){1}}
%
% Draw P edges
\multiput(\PM,3)(\N,0){\M}{\line(0,1){1}}
%
% Draw C-P lines
\multiput(\CM,3.5)(\N,0){\M}{\line(1,0){\CPM}}
%
% Draw C's
\multiput(\CM,3)(\N,0){\M}{\makebox(0,0)[r]{\tiny C\,}}
% Draw P's
\multiput(\PM,3)(\N,0){\M}{\makebox(0,0)[l]{\tiny \,P}}
%
%
% Draw N cycles
```

```
\multiput(0,9)(0,3){2}{\line(1,0){\MN}}
\multiput(0,9)(\M,0){\No}{\line(0,1){3}}
\multiput(0,9)(1,0){\MN}{\line(0,1){.5}}
\multiput(0,9.5)(\M,0){\N}{\makebox(\M,2.5){\arabic{cN}}\addtocounter{cN}{1}}
%
% Draw S edges
\multiput(\NSN,7)(\M,0){\N}{\line(0,1){1}}
%
% Draw H edges
\multiput(\HN,7)(\M,0){\N}{\line(0,1){1}}
%
% Draw S
\multiput(\NSN,8)(\M,0){\N}{\makebox(0,0)[r]{\tiny S\,}}
%
% Draw S-Clk lines
\multiput(\NSN,7.5)(\M,0){\N}{\line(1,0){\SN}}
%
% Draw H
\multiput(\HN,8)(\M,0){\N}{\makebox(0,0)[l]{\tiny \,H}}
%
% Draw H-Clk lines
\multiput(\HN,7.5)(\M,0){\N}{\line(-1,0){\HN}}
```

Appendix C

The Prototype Chip

Figure C-1 shows the layout and pin assignments of the prototype rational clocking chip. Details of its features and its intended use are provided in Chapter 6.

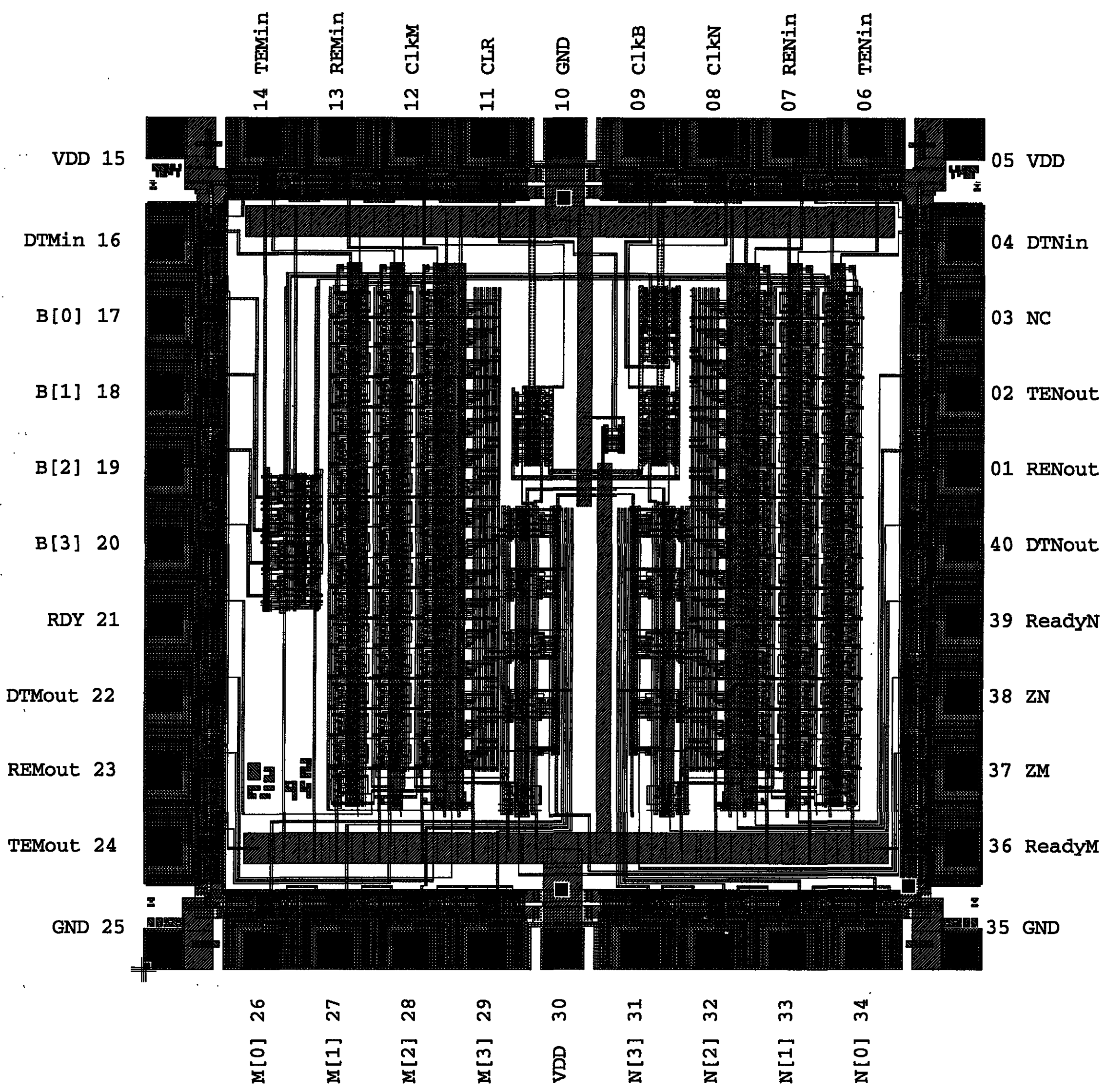


Figure C-1: Layout and pinout of the prototype rational clocking chip.

Appendix D

Verilog Source Code

This appendix contains the complete Verilog source code of the run-time scheduling hardware model. The code is divided into five sections of descending hierarchical order. The top-level modules are presented first, followed by the high-level communication scheduling hardware modules. The components that make up the scheduling hardware are presented next, followed by the circuits that compute the constants used by these components. The listing ends with the basic components, such as registers and multiplexers, which are used by all the other higher-level modules.

D.1 Top-Level Modules

```
/* params.vh
 *
 * This file contains the timing parameters in units of Delta t.
 * It is included in a number of modules.  Modify this file to modify
 * the timing parameters.
 *
 */

   `define M 5'd5
   `define N 5'd6

   `define CM 5'd1
   `define PM 5'd3
   `define SM 5'd1
   `define HM 5'd1

   `define CN 5'd1
   `define PN 5'd3
   `define SN 5'd1
   `define HN 5'd1


/* driver.v
 *
 * Driver / Test Module
 *
 * Luis F. G. Sarmenta 950514
 *
 */

/* Define the algorithm/hardware to use */

`define GENB


/* Define which wave forms to view */

`define VIEW_M_TO_N


module driver;

   parameter width = 5;

   reg clkhi, clr;
   wire clkM, clkN;
   wire [width-1:0] cntM, cntN, dM, dN, qM, qN, qinM, qinN;

   wire TEM,TSelM,REN,RSelN,TEN,TSelN,REM,RSelM;

   wire validN, validM;
   wire dlyREM, dlyREN, dlyRSelM, dlyRSelN;

`include "params.vh"

   parameter SB = 1'b0;

   parameter M = `M;
   parameter N = `N;

   parameter CM = `CM;
   parameter PM = `PM;
   parameter SM = `SM;
   parameter HM = `HM;

   parameter CN = `CN;
   parameter PN = `PN;
   parameter SN = `SN;
   parameter HN = `HN;


/*
```

```
 * Clock Generation (Clock Division) Counters
 *
 */

   fdiv #(width) genclkM( clkM, , clkhi, N, 1'b0, clr );
   fdiv #(width) genclkN( clkN, , clkhi, M, 1'b0, clr );

/*
 * Scheduling Hardware
 *
 */

`ifdef NOCNTL                           /* No flow control */
   /* Index Counters (the other hardware options have built-in
      index counters) */
   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   assign TEM = 1'b1;
   assign TSelM = 1'b1;
   assign REN = 1'b1;
   assign RSelN = 1'b1;
   assign TEN = 1'b1;
   assign TSelN = 1'b1;
   assign REM = 1'b1;
   assign RSelM = 1'b1;
`endif

`ifdef DBLAZY                           /* Double-buffered Lazy */
   dbstfrl #(width) stfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );

   dbftsrl #(width) ftsr( , , REM, RSelM, , , TEN, TSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );
`endif

`ifdef DBGREEDY                         /* Double-buffered Greedy */
   dbstfrg #(width) stfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );

   dbftsrg #(width) ftsr( , , REM, RSelM, , , TEN, TSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );
`endif

`ifdef GENB                             /* Generalized Scheduling */
   gbstfr #(width) stfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                          clkhi, clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN,
                          SB );

   gbftsr #(width) ftsr( , , REM, RSelM, , , TEN, TSelN,
                          clkhi, clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN,
                          SB );
`endif


/*
 * Data Channels
 *
 */

        /* M to N */

   counten #(width) dataM( , dM, clkM, TEM, clr );

   /* Note: Here we delay RSel by the receiver's C delay to prevent RSel
    * from causing a hold time violation.  */

   assign #(CN*100) dlyRSelN = RSelN;
   assign #(CN*100) dlyREN = REN;
   dbout   #(width,CM*100,PM*100) outM( qM, clkM, dM, TEM, TSelM, dlyRSelN );

   rxregN   #(width) inN( qinN, clkN, qM, dlyREN );
   regen   vN( validN, clkN, REN, 1'b1 );

        /* N to M */

   counten #(width) dataN( , dN, clkN, TEN, clr );

   assign #(CM*100) dlyRSelM = RSelM;
   assign #(CM*100) dlyREM = REM;
   dbout   #(width,CN*100,PN*100) outN( qN, clkN, dN, TEN, TSelN, dlyRSelM );

   rxregM   #(width) inM( qinM, clkM, qN, dlyREM );
   regen   vM( validM, clkM, REM, 1'b1 );


/*
 * Clkhi and Reset Generators
 *
 */

        /* clock period = 100 */

   initial
      forever
      begin
         clkhi = 1;
         #50;
         clkhi = 0;
         #50;
      end

        /* reset on start */

   initial
      begin
         clr = 0 ;
         #150;
         clr = 1 ;
         #100;
         clr = 0 ;
      end

/*
 * Display Routines
 *
 */

/*  Primitive text output for non-graphics terminals

         always @( cntM )
            $strobe( "cM=%d, %b%b %b%b", cntM[width-1:0],
                     TEM, TSelM, REM, RSelM );

         always @( cntN )
            $strobe( "\t\tcN=%d, %b%b %b%b", cntN[width-1:0],
                     REN, RSelN, TEN, TSelN,  );
*/

/* Graphical Output */

   initial
      begin
         $gr_waves(
`ifdef VIEW_M_TO_N
                  "cntN", cntN[width-1:0],
                  "qinN", qinN[width-1:0],
                  "validN", validN,
                  "REN'", dlyREN,
                  "RSelN'", dlyRSelN,

                  "qM", qM[width-1:0],
                  "q0M", outM.q0[width-1:0],
                  "q1M", outM.q1[width-1:0],
                  "TSelM", TSelM,
                  "TEM", TEM,
                  "dM", dM[width-1:0],
                  "cntM", cntM[width-1:0]
`endif
`ifdef VIEW_N_TO_M
                  "cntN", cntN[width-1:0],
                  "dN", dN[width-1:0],
                  "TEN", TEN,
                  "TSelN", TSelN,
                  "q0N", outN.q0[width-1:0],
                  "q1N", outN.q1[width-1:0],
                  "qN", qN[width-1:0],

                  "RSelM'", dlyRSelM,
                  "REM'", dlyREM,
                  "validM", validM,
                  "qinM", qinM[width-1:0],
                  "cntM", cntM[width-1:0]
`endif
                 );

      end

endmodule
```

D.2 Schedulers and Data Channels

```
/* dbstfrl.v
 *
 * Controller for double-buffering, slower transmitter case (lazy algo)
 *  ( assumes M < N )
 *
 */

module dbstfrl( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, TEM, TSelM, zN, REN, RSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   wire [width-1:0] f0, e0, x0, y0;
   wire [width-1:0] divst, modst;

   assign x0 = 1;
   assign f0 = -(PM+SN);

   commoddiv #(width) compe0y0st( divst, modst, , f0, N );
   forme0    #(width) compe0st( e0, modst, N, f0[width-1] );
   formy0    #(width) compy0st( y0, divst, f0[width-1] );

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdbstl #(width) schM( TEM, TSelM, clkM, cntM, M, N, e0, y0 );
   schdbfrl #(width) schN( REN, RSelN, clkN, cntN, M, N, e0, x0 );

endmodule
```

```
/* dbftsrl.v
 *
 * Controller for double-buffering, slower receiver case (lazy algo)
 *  ( assumes M < N )
 *
 */

module dbftsrl( zM, cntM, REM, RSelM, zN, cntN, TEN, TSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, REM, RSelM, zN, TEN, TSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   wire [width-1:0] f0, e0, x0, y0;
   wire [width-1:0] divst, modst;

   assign x0 = 1;
   assign f0 = CN-HM;

   commoddiv #(width) compe0y0st( divst, modst, , f0, N );
   forme0    #(width) compe0st( e0, modst, N, f0[width-1] );
   formy0    #(width) compy0st( y0, divst, f0[width-1] );

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdbsrl #(width) schM( REM, RSelM, clkM, cntM, M, N, e0, y0 );
   schdbftl #(width) schN( TEN, TSelN, clkN, cntN, M, N, e0, x0 );

endmodule


/* dbstfrg.v
 *
 * Controller for double-buffering, slower transmitter case (greedy algo)
 *  ( assumes M < N )
 *
 */

module dbstfrg( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, TEM, TSelM, zN, REN, RSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   dbstfrl #(width) schMN( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                           clkM, clkN, clr,
                           M, N, CM, PM, SM, HM, CN, PN, SN, HN );
endmodule


/* dbftsrg.v
 *
 * Controller for double-buffering, slower receiver case (greedy algo)
 *  ( assumes M < N )
 *
 */

module dbftsrg( zM, cntM, REM, RSelM, zN, cntN, TEN, TSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, REM, RSelM, zN, TEN, TSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   wire [width-1:0] f0, e0, adje0, x0, y0, adjy0;
   wire [width-1:0] divsr, modsr;

   assign x0 = 1;
   assign f0 = SM+PN;

   commoddiv #(width) compe0y0sr( divsr, modsr, , f0, N );
   forme0    #(width) compe0sr( e0, modsr, N, f0[width-1] );
   adjuste0  #(width) adjuste0( adje0, e0, N );

   formy0    #(width) compy0sr( y0, divsr, f0[width-1] );
   assign    adjy0 = (adje0 == 0) ? (y0-1) : y0;

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdbsrg #(width) schM( REM, RSelM, clkM, cntM, M, N, adje0, adjy0 );
   schdbftg #(width) schN( TEN, TSelN, clkN, cntN, M, N, adje0, x0 );

endmodule


/* gbstfr.v
 *
 * Controller for generalized scheduling, slower transmitter case
 *  ( assumes M < N )
 *
 */
```

```
module gbstfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                 clkS, clkM, clkN, clr,
                 M, N, CM, PM, SM, HM, CN, PN, SN, HN,
                 SB );

   parameter width = 5;

   output zM, TEM, TSelM, zN, REN, RSelN;
   output  [width-1:0] cntM, cntN;
   input  clkS, clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;
   input SB;

   wire [width-1:0] f0, e0, x0, y0, DMN, negDMN, e0M, emax;
   wire [width-1:0] divst, modst;
   wire doneeOM, doneDMN;

   seqmod #(width) cDMN( DMN, doneDMN, clkS, N, M, clr );
   neg    #(width) cnDMN( negDMN, DMN );

   assign x0 = 1;
   assign f0 = -(PM+SN);

   commoddiv #(width) compe0y0st( divst, modst, , f0, N );
   forme0    #(width) compe0st( e0, modst, N, f0[width-1] );
   formy0    #(width) compy0st( y0, divst, f0[width-1] );

   seqmod  #(width) ce0Mst( e0M, donee0M, clkS, e0, M, clr );

   assign emax = N - ((PM-CM) + (SN+HN));

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdgst #(width) schM( TEM, TSelM, clkM, cntM,
                          M, negDMN, e0M, y0, emax, SB );
   schdgfr #(width) schN( REN, RSelN, clkN, cntN,
                          M, N, e0, x0, emax, SB );
endmodule


/* gbftsr.v
 *
 * Controller for generalized scheduling, slower receiver case (greedy algo)
 *  ( assumes M < N )
 *
 */

module gbftsr( zM, cntM, REM, RSelM, zN, cntN, TEN, TSelN,
              clkS, clkM, clkN, clr,
              M, N, CM, PM, SM, HM, CN, PN, SN, HN,
              SB );

   parameter width = 5;
   output zM, REM, RSelM, zN, TEN, TSelN;
   output  [width-1:0] cntM, cntN;
   input  clkS, clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;
   input SB;

   wire [width-1:0] f0, e0, adje0, x0, y0, adjy0;
   wire [width-1:0] DMN, negDMN, e0M, adje0M, emin;
   wire [width-1:0] divsr, modsr;
   wire donee0M, doneDMN;

   seqmod #(width) cDMN( DMN, doneDMN, clkS, N, M, clr );
   neg    #(width) cnDMN( negDMN, DMN );

   assign x0 = 1;
   assign f0 = SM+PN;

   commoddiv #(width) compe0y0sr( divsr, modsr, , f0, N );
   forme0    #(width) compe0sr( e0, modsr, N, f0[width-1] );
   adjuste0  #(width) adjuste0( adje0, e0, N );

   formy0    #(width) compy0sr( y0, divsr, f0[width-1] );
   assign    adjy0 = (adje0 == 0) ? (y0-1) : y0;

   seqmod  #(width) ce0Msr( e0M, donee0M, clkS, adje0, M, clr );
   adjuste0M #(width) adjuste0M( adje0M, e0M, M );

   assign emin = M + ((PN-CN) + (SM+HM)) - N;


   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdgsr #(width) schM( REM, RSelM, clkM, cntM,
                          M, negDMN, adje0M, adjy0, emin, SB );
   schdgft #(width) schN( TEN, TSelN, clkN, cntN,
                          M, N, adje0, x0, emin, SB );
endmodule


/* dbout.v
 *
 * Double-bufferd output hardware
 *
 * This module contains two transmit registers a selecting demux.
 * It takes the transmitter's C and P delays as parameters
 * and passes them to txreg.
 *
 */

module dbout( q, clk, d, TE, TSel, RSel );

   parameter width = 1;
   parameter C = 0;
   parameter P = 0;
   output [width-1:0] q;
   input  clk;
   input [width-1:0] d;
   input TE, TSel, RSel;

   wire en0, en1;
   wire [width-1:0] q0, q1;
   wire dlyRSel;

   txreg #(width,C,P) r0( q0, clk, d, en0 );
   txreg #(width,C,P) r1( q1, clk, d, en1 );
   demux1to2 seldemux( en0, en1, TE, TSel );

   mux1of2 #(width) qmux( q, q0, q1, RSel );

endmodule


/* rxregM.v
 *
 * Receive Register with Enable and specifiable S and H times
 * <<< For System M >>>
 *
 * Note: S and H times are specified by modifying params.vh
 *
 */

module rxregM( q, clk, d, en );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en;

   reg [width-1:0] q;
   wire [width-1:0] q1;
   reg  flag;

'include "params.vh"

   specify
      specparam S = ('SM*100), H = ('HM*100);

      $setuphold( posedge clk &&& en, d, S, H, flag );
   endspecify

   always @(posedge clk)
   begin
      if (en)
         q = d;
   end
   always @(flag) q = q + 1'bx;

endmodule


/* rxregN.v
 *
 * Receive Register with Enable and specifiable S and H times
 * <<< For System N >>>
 *
 * Note: S and H times are specified by modifying params.vh
 *
 */

module rxregN( q, clk, d, en );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en;

   reg [width-1:0] q;
   wire [width-1:0] q1;
   reg  flag;

`include "params.vh"

   specify
      specparam S = (`SN*100), H = (`HN*100);

      $setuphold( posedge clk &&& en, d, S, H, flag );
   endspecify

   always @(posedge clk)
   begin
      if (en)
         q = d;
   end

   always @(flag) q = q + 1'bx;

endmodule
```

D.3 Scheduler Components

```
/* schdbfrg.v
 *
 * Scheduling hardware for double-buffering, faster receiver, greedy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbfrg( RE, RSel, clk, cnt, M, N, e0, x0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0;

   schdbfrl #(width) sch( RE, RSel, clk, cnt, M, N, e0, x0 );

endmodule


/* schdbfrl.v
 *
 * Scheduling hardware for double-buffering, faster receiver, lazy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbfrl( RE, RSel, clk, cnt, M, N, e0, x0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0;

   wire ld;
   wire [width-1:0] x0_1;

   assign x0_1 = x0 - 1;

   ldgen #(width) ldgenN( ld, cnt, x0_1, N );
   egendb #(width) egenN( RE, , clk, ld, M, N, e0, 1'b0 );
   rgendb rgenN( RSel, clk, RE, ld );

endmodule


/* schdbftg.v
 *
 * Scheduling hardware for double-buffering, faster transmitter, greedy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbftg( TE, TSel, clk, cnt, M, N, adje0, x0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, adje0, x0;

   wire ld;
   wire [width-1:0] x0_2;

   assign x0_2 = x0 - 2;

   ldgen #(width) ldgenN( ld, cnt, x0_2, N );
   egendbg #(width) egenN( TE, , clk, ld, M, N, adje0, 1'b0 );
   rgendb rgenN( TSel, clk, TE, ld );

endmodule


/* schdbftl.v
 *
 * Scheduling hardware for double-buffering, faster transmitter, lazy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbftl( TE, TSel, clk, cnt, M, N, e0, x0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0;

   schdbfrl #(width) sch( TE, TSel, clk, cnt, M, N, e0, x0 );

endmodule


/* schdbsrg.v
 *
 * Scheduling hardware for double-buffering, slower receiver, greedy algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbsrg( RE, RSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   schdbstl #(width) sch( RE, RSel, clk, cnt, M, N, e0, y0 );

endmodule


/* schdbsrl.v
```

```
 *
 * Scheduling hardware for double-buffering, slower receiver, lazy  algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbsrl( RE, RSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   schdbstl #(width) sch( RE, RSel, clk, cnt, M, N, e0, y0 );

endmodule


/* schdbstg.v
 *
 * Scheduling hardware for double-buffering, slower transmitter, greedy algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbstg( TE, TSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   schdbstl #(width) sch ( TE, TSel, clk, cnt, M, N, e0, y0 );

endmodule


/* schdbstl.v
 *
 * Scheduling hardware for double-buffering, slower transmitter, lazy algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbstl( TE, TSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   wire TE;
   wire ld;
   wire [width-1:0] y0_1;

   assign TE = 1'b1;
   assign y0_1 = y0 - 1;

   ldgen #(width) ldgenM( ld, cnt, y0_1, M );
   rgendb rgenM( TSel, clk, TE, ld );

endmodule


/* schdgfr.v
 *
 * Scheduling hardware for generalized scheduling, faster receiver
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdgfr( RE, RSel, clk, cnt, M, N, e0, x0, emax, SB );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0, emax;
   input  SB;

   wire ld;
   wire [width-1:0] x0_1;
   wire preRE;
   wire [width-1:0] e;

   assign x0_1 = x0 - 1;

   ldgen #(width) ldgenN( ld, cnt, x0_1, N );

   egendb #(width) egenN( preRE, e, clk, ld, M, N, e0, 1'b0 );
   rgengbst #(width) rgenN( RSel, clk, preRE, ld, e, emax );

   nand nand1( noregs, SB, RSel );
   and  and1( RE, preRE, noregs );

endmodule


/* schdgft.v
 *
 * Scheduling hardware for generalized scheduling, faster transmitter
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdgft( TE, TSel, clk, cnt, M, N, adje0, x0, emin, SB );
```

```
   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, adje0, x0, emin;
   input  SB;

   wire ld;
   wire [width-1:0] x0_2;
   wire preTE;
   wire [width-1:0] e;

   assign x0_2 = x0 - 2;

   ldgen #(width) ldgenN( ld, cnt, x0_2, N );

   egendbg  #(width) egenN( preTE, e, clk, ld, M, N, adje0, 1'b0 );
   rgengbsr #(width) rgenN( TSel, clk, preTE, ld, e, emin );

   nand nand1( noregs, SB, TSel );
   and  and1( TE, preTE, noregs );

endmodule


/* schdgsr.v
 *
 * Scheduling hardware for generalized scheduling, slower receiver
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdgsr( RE, RSel, clk, cnt, M, negDMN, adje0M, y0, emin, SB );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, negDMN, cnt, adje0M, y0, emin;
   input  SB;

   wire preRE;
   wire ld;
   wire [width-1:0] y0_1;
   wire [width-1:0] negM, e;

   assign preRE = 1'b1;
   assign y0_1 = y0 - 1;

   assign negM = -M;

   ldgen #(width) ldgenN( ld, cnt, y0_1, M );

   egendbg #(width) egenM( , e, clk, ld, negDMN, negM, adje0M, 1'b1 );
   rgengbsr #(width) rgenM( RSel, clk, preRE, ld, e, emin );
   nand nand1( noregs, SB, RSel );
   and  and1( RE, preRE, noregs );

endmodule


/* schdgst.v
 *
 * Scheduling hardware for generalized scheduling, slower transmitter
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdgst( TE, TSel, clk, cnt, M, negDMN, e0M, y0, emax, SB );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, negDMN, cnt, e0M, y0, emax;
   input  SB;

   wire preTE;
   wire ld;
   wire [width-1:0] y0_1;
   wire [width-1:0] negM, e;

   assign preTE = 1'b1;
   assign y0_1 = y0 - 1;

   assign negM = -M;

   ldgen #(width) ldgenN( ld, cnt, y0_1, M );

   egendb #(width) egenN( , e, clk, ld, negDMN, negM, e0M, 1'b1 );
   rgengbst #(width) rgenN( TSel, clk, preTE, ld, e, emax );

   nand nand1( noregs, SB, TSel );
   and  and1( TE, preTE, noregs );

endmodule


/* egendb.v
 *
 * The EGen module, slower transmitter case
 *
 * This module works as the EGen module for double-buffering when gb is 0,
 * and works as the modified EGen module in the slower system's
 * generalized schedling hardware for the slower transmitter case.
 *
 */

module egendb( E, eout, clk, ld, M, N, e0, gb );
```

```
   parameter width = 1;

   output E;
   output [width-1:0] eout;
   input clk;
   input ld;
   input [width-1:0] M, N, e0;
   input gb;

   wire E;
   wire [width-1:0] eout, negN, eold, e, eN, nxteold;
   wire sgne;
   wire muxsel;

   assign negN = -N;

   muxreg #(width) eoldreg( eold, clk, nxteold, e0, ld );
   adder  #(width) eoplusM( e, eold, M, 1'b0 );
   adder #(width) esubN( eN, e, negN, 1'b0 );

   assign sgne = e[width-1];
   assign E = ~sgne;

   xor x1( muxsel, sgne, gb );
   mux1of2 #(width) neomux( nxteold, eN, e, muxsel );

   mux1of2 #(width) eoutmux( eout, e, eold, gb );

endmodule


/* egendbg.v
 *
 * The EGen module for double-buffering greedy slower receiver algo
 *
 */

module egendbg( E, eout, clk, ld, M, N, e0, gb );

   parameter width = 1;

   output E;
   output [width-1:0] eout;
   input clk;
   input ld;
   input [width-1:0] M, N, e0;
   input gb;

   wire E, notE;
   wire [width-1:0] eout, negN, eold, e, eN, nxteold;
   wire sgne, z;
   wire muxsel;

   assign negN = -N;
   muxreg #(width) eoldreg( eold, clk, nxteold, e0, ld );
   adder  #(width) eoplusM( e, eold, M, 1'b0 );
   adder #(width)  esubN( eN, e, negN, 1'b0 );
   zero   #(width) ez( z, e );

   assign sgne = e[width-1];

   nor    gtz( E, z, sgne );
   assign notE = ~E;

   xor x1( muxsel, notE, gb );
   mux1of2 #(width) neomux( nxteold, eN, e, muxsel );

   mux1of2 #(width) eoutmux( eout, e, eold, gb );

endmodule


/* ldgen.v
 *
 * LD Gen module
 *   (checks if count == (i-1) mod M and generates LD signal)
 *
 */

module ldgen( ld, count, i, M );

   parameter width = 1;

   output ld;
   input  [width-1:0] count, i, M;

   wire   [width-1:0] i_1, a1, iMo;

   assign a1 = 1;

   addsub #(width) sub1( i_1, i, a1, 1'b1 );
   commod #(width) domod( iMo, i_1, M );

   assign ld = ( count == iMo );

endmodule


/* rgendb.v
 *
 * The RGen module for double-buffering
 *
 */

module rgendb( r, clk, en, ld );

   output r;
   input clk;
   input en, ld;

   tflopenld rtflop( r, clk, 1'b0, en, ld );

endmodule


/* rgengbsr.v
 *
 * The RGen module for generalized run-time scheduling, slower receiver case
 *
 */

module rgengbsr( r, clk, en, ld, e, emin );

   parameter width = 1;

   output r;
   input clk;
   input en, ld;
   input [width-1:0] e, emin;

   wire ge, q;

   leq #(width) egeqemin( ge, emin, e );
   nor nr1( r, ge, q );
   regencs rreg( q, clk, r, en, 1'b0, ld );

endmodule


/* rgengbst.v
 *
 * The RGen module for generalized scheduling, slower transmitter case
 *
 */

module rgengbst( r, clk, en, ld, e, emax );

   parameter width = 1;

   output r;
   input clk;
   input en, ld;
   input [width-1:0] e, emax;

   wire le, d;

   leq #(width) eleqemax( le, e, emax );
   nor nr1( d, le, r );
   regencs rreg( r, clk, d, en, ld, 1'b0 );

endmodule
```

D.4 Bootup Computation Circuitry

```
/* commod.v
 *
 * combinational mod module
 *
 * This module computes:
 *       mod = (a mod b) for  -3b < a < 3b
 *
 */

module commod( modout, a, b );

   parameter width = 1;

   output [width-1:0] modout;
   input  [width-1:0] a, b;

   wire [width-1:0] mod, modprime;

   commoddiv #(width) cmd( , mod, modprime, a, b );
   mux1of2 #(width) outmux( modout, mod, modprime, a[width-1] );

endmodule


/* commoddiv.v
 *
 * combinational mod / div module -- cascaded twice
 *
 * (see moddiv.v for more details)
 *
 */

module commoddiv( div, mod, modprime, a, b );

   parameter width = 1;

   output [width-1:0] div, mod, modprime;
   input  [width-1:0] a, b;

   wire [width-1:0] nxtmod, nxtdiv, div0;

   assign div0 = 0;

   moddiv #(width) m1( nxtdiv, nxtmod, , , a, b, div0 );
   moddiv #(width) m2( div, mod, modprime, , nxtmod, b, nxtdiv );

endmodule


/* moddiv.v
 *
 * mod / div module
 *
 * This module computes:
 *     if a >= 0
 *        mod = (a mod b)
 *        div = (a div b)
 *     if a < 0
 *        mod = (a mod b) - b
 *        mod' = (a mod b)
 *        div = -(a div b)         if (a mod b != 0)
 *            = -(a div b) - 1     if (a mod b == 0)
 *
 * This module can be cascaded by connecting mod to a and div to divin.
 * sgnchg would then indicate when the operation is done.
 *
 */

module moddiv( div, mod, modprime, sgnchg, a, b, divin );

   parameter width = 1;

   output [width-1:0] div, mod, modprime;
   output sgnchg;
   input  [width-1:0] a, b, divin;

   wire [width-1:0] div, mod, modprime;
   wire sgna, nsgna, sgnchg;

   assign sgna = a[width-1];
   assign nsgna = ~sgna;

   addsub #(width) add1( modprime, a, b, nsgna );
   xor x1( sgnchg, modprime[width-1], sgna );
   mux1of2 #(width) modmux( mod, modprime, a, sgnchg );
   incinh  #(width) divinc( div, divin, sgnchg );

endmodule


/* seqmod.v
 *
 * sequential mod module
 *
 * This module computes:
 *       mod = (a mod b)
 *
 */

module seqmod( modout, done, clk, a, b, start );

   parameter width = 1;

   output [width-1:0] modout;
   output done;
   input clk;
   input  [width-1:0] a, b;
   input start;
   wire [width-1:0] mod, modprime, nxtmod;

   seqmoddiv #(width) smd( , mod, modprime, done, clk, a, b, start );
   mux1of2 #(width) outmux( modout, mod, modprime, a[width-1] );

endmodule


/* seqmoddiv.v
 *
 * sequential mod / div module
 *
 * (see moddiv.v)
 *
 */

module seqmoddiv( div, mod, modprime, done, clk, a, b, start );

   parameter width = 1;

   output [width-1:0] div, mod, modprime;
   output done;
   input clk;
   input  [width-1:0] a, b;
   input start;

   wire [width-1:0] div, mod, nxtmod, nxtdiv;

   regencs #(width) curdiv( div, clk, nxtdiv, 1'b1, start, 1'b0 );
   muxreg #(width) curmod( mod, clk, nxtmod, a, start );
   moddiv #(width) compute( nxtdiv, nxtmod, modprime, done, mod, b, div );

endmodule


/* adjuste0.v
 *
 * Adjust module for e0 for greedy slower receiver algo
 *
 */

module adjuste0( newe0, e0, N );

   parameter width = 1;

   output [width-1:0] newe0;
   input  [width-1:0] e0, N;

   wire   [width-1:0] eplusN;
   wire   z;

   adder #(width) ein( eplusN, e0, N, 1'b0 );
   zero  #(width) zcheck( z, eplusN );
   mux1of2 #(width) emux( newe0, e0, eplusN, z );
```

```
endmodule


/* adjuste0M.v
 *
 * Adjust module for e0 mod M for generalized slower receiver algo
 *
 */

module adjuste0M( newe0M, e0M, M );

   parameter width = 1;

   output [width-1:0] newe0M;
   input  [width-1:0] e0M, M;

   wire   [width-1:0] eplusM;
   wire   z;

   adder #(width) ein( eplusM, e0M, M, 1'b0 );
   zero  #(width) zcheck( z, e0M );
   mux1of2 #(width) emux( newe0M, e0M, eplusM, z );

endmodule


/* forme0.v
 *
 * Module to form e0 = (f0 mod N) - N
 *    from the output of the mod/div module
 *
 */

module forme0( e0, modin, N, sgnf0 );

   parameter width = 1;

   output [width-1:0] e0;
   input  [width-1:0] modin, N;
   input  sgnf0;

   wire [width-1:0] e0, eN;

   addsub #(width) add1( eN, modin, N, 1'b1 );
   mux1of2 #(width) outmux( e0, eN, modin, sgnf0 );

endmodule


/* formy0.v
 *
 * Module to form y0
 *    from the output of the mod/div module
 *
 */

module formy0( y0, divin, sgnf0 );

   parameter width = 1;

   output [width-1:0] y0;
   input  [width-1:0] divin;
   input  sgnf0;

   wire nsgnf0;
   wire [width-1:0] zor1;

   assign nsgnf0 = ~sgnf0;

   assign zor1 = 0 | nsgnf0;

   addsub #(width) add1( y0, zor1, divin, sgnf0 );

endmodule
```

D.5 Basic Components

```
/* adder.v
 *
 * variable width adder with cin
 *
 */

module adder( sum, a, b, cin );

   parameter width = 1;

   output [width-1:0] sum;
   input  [width-1:0] a, b;
   input  cin;

   assign sum = a + b + cin;

endmodule


/* addsub.v
 *
 * variable width adder/subtractor with carry-in
 *
 */

module addsub( sum, a, b, sub );

   parameter width = 1;
   output [width-1:0] sum;
   input  [width-1:0] a, b;
   input  sub;

   assign sum = sub ? (a-b) : (a+b);

endmodule


/* counten.v
 *
 * countup timer with enable, Asynchronous clear
 *
 */

module counten( z, q, clk, en, clr );

   parameter width = 1;

   output z;
   output [width-1:0] q;
   input clk;
   input en, clr;

   reg [width-1:0] q;

   assign z = (q == 0);

   always @(clr)
      if (clr)
         q = 0;

   always @(posedge clk)
      if (clr)
         q = 0;
      else
         if (en)
               q = q + 1;

endmodule


/* countup.v
 *
 * countup timer with terminal count (tc), Asynchronous clear and set (to tc)
 *
 */

module countup( z, q, clk, tc, clr, set );

   parameter width = 1;

   output z;
   output [width-1:0] q;
   input clk;
   input [width-1:0] tc;
   input clr, set;

   reg [width-1:0] q;

   assign z = (q == 0);

   always @(clr or set)
      if (clr)
         q = 0;
      else
         if (set)
            q = tc;

   always @(posedge clk)
      if (clr)
         q = 0;
      else
         if (set)
            q = tc;
         else
            q = (q == tc) ? 0 : (q + 1);

endmodule


/* demux1to2.v
 *
 * 1-to-2 demux
 *
 */

module demux1to2( q0, q1, d, sel );

   parameter width = 1;

   output [width-1:0] q0, q1;
   input [width-1:0] d;
   input sel;

   assign q0 = sel ? 0 : d;
   assign q1 = sel ? d : 0;

endmodule


/* fdiv.v
 *
 * Frequency divider, divides by M
 *
 * (Note: fdiv currently doesn't work for M = 1)
 *
 */

module fdiv( clkout, q, clkin, M, clr, set );
```

```
   parameter width = 1;

   output clkout;
   output [width-1:0] q;
   input clkin;
   input [width-1:0] M;
   input clr, set;

   wire z;
   wire [width-1:0] tc;

   wire  clkout;

   assign tc = M-1;

   countup #(width) cnt ( z, q, clkin, tc, clr, set );

   assign clkout = z;

endmodule


/* incinh.v
 *
 * Combinational increment with Inhibit
 *
 */

module incinh( res, a, inh );

   parameter width = 1;

   output [width-1:0] res;
   input  [width-1:0] a;
   input  inh;

   assign res = inh ? a : (a+1);

endmodule


/* leq.v
 *
 * Less-than-or-equal-to comparator
 *   ( checks if a <= b )
 *
 * Note that this version assumes Verilog treats the numbers as
 * unsigned numbers and does its own signed version of leq.
 *
 * Luis F. G. Sarmenta 950409.2128
 *
 */

module leq( res, a, b );
```

```
   parameter width = 1;

   output res;
   input  [width-1:0] a, b;

   assign res = ( a[width-1]
                 ? ( b[width-1]
                      ? ( a >= b )
                      : 1
                   )
                 : ( !b[width-1]
                      ? ( a <= b )
                      : 0
                   )
               );
endmodule


/* mux1of2.v
 *
 * 1-of-2 mux
 *
 */

module mux1of2( q, d0, d1, sel );

   parameter width = 1;
   output [width-1:0] q;
   input [width-1:0] d0, d1;
   input sel;

   assign q = sel ? d1 : d0;

endmodule


/* muxreg.v
 *
 * 1-of-2 mux with storage
 *
 */

module muxreg( q, clk, d0, d1, sel );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d0, d1;
   input sel;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      case (sel)
         0 : q = d0;
         1 : q = d1;
      endcase
   end

endmodule


/* neg.v
 *
 * Arithmetic Negator
 *
 */

module neg( nega, a );

   parameter width = 1;

   output [width-1:0] nega;
   input  [width-1:0] a;

   assign nega = -a;

endmodule


/* regen.v
 *
 * Register with Enable
 *
 */

module regen( q, clk, d, en );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      if (en)
         q = d;
   end

endmodule


/* regencs.v
```

```
 *
 * Register with Enable and Synchronous Clear and Set
 * (Clr overrides Set)
 *
 */

module regencs( q, clk, d, en, clr, set );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en, clr, set;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      if (en)
         if (clr)
            q = 0;
         else
            if (set)
               q = 1;
            else
               q = d;
   end

endmodule


/* tflopenld.v
 *
 * a T flip-flop with Enable and Load
 *  ( ld overrides en )
 *
 */

module tflopenld( q, clk, d, en, ld );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en, ld;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      case (ld)
            0 : q = en ? ~q : q;
            1 : q = d;
      endcase
   end


endmodule


/* zero.v
 *
 * Zero check
 *
 */

module zero( z, a );

   parameter width = 1;

   output z;
   input  [width-1:0] a;

   assign z = ( a == 0 );

endmodule
```

Bibliography

[1] G. A. Pratt and S. A. Ward. Rationally clocked communication. Unpublished Manuscript, MIT Computer Architecture Group, October 1992.

[2] MIT Computer Architecture Group. Rationally clocked communication. In *Progress Report 29*, pages 101–107. MIT Laboratory for Computer Science, June 1992.

[3] S. A. Ward et al. The NuMesh: A modular, scalable communication substrate. In *Proceedings of the International Conference on Supercomputing*, 1993.

[4] B. Burgess et al. The PowerPC 603 microprocessor. *Communications of the ACM*, June 1994.

[5] IEEE standard for a simple 32-bit backplane bus: NuBus. ANSI/IEEE Std 1196-1987, IEEE, 1988.

[6] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, April 1973.

[7] J. F. Wakerly. A designer's guide to synchronizers and metastability. Technical report, Center for Reliable Computing, Stanford University, February 1988.

[8] G. R. Couranz and D. F. Wann. Theoretical and experimental behavior of synchronizers operating in the metastable region. *IEEE Transactions on Computers*, June 1975.

[9] M. Pěchouček. Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, February 1976.

[10] C. A. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, Mass., 1980.

[11] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, October 1984.

[12] M. Afghahi and C. Svensson. Performance of synchronous and asynchronous schemes for vlsi systems. *IEEE Transactions on Computers*, July 1992.

[13] W. K. Stewart and S. A. Ward. A solution to a special case of the synchronization problem. *IEEE Transactions on Computers*, January 1988.

[14] R. C. Swiston. An adaptive periodic synchronizer circuit. Bachelor's thesis, MIT, May 1988.

[15] K. Suzuki et al. A 500MHz 32b 0.4μm CMOS RISC Processor LSI. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 214–215, 1994.

[16] J. Schutz. A 3.3v 0.6μm BiCMOS superscalar microprocessor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 202–203, 1994.

[17] E. Rashid et al. A CMOS RISC CPU with on-chip parallel cache. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 210–211, 1994.

[18] D. C. Flemming. Data transfer apparatus. European Patent Application 81102943.8, Publication number 004294, 1982.

[19] F. Gardner. *Phaselock Techniques.* John Wiley & Sons, New York, 2nd edition, 1979.

[20] D. K. Jeong et al. Design of PLL-based clock generation circuits. *IEEE Journal of Solid-State Circuits*, April 1987.

[21] I. A. Young et al. A PLL clock generator with 5–110 MHz of lock range for microprocessors. *IEEE Journal of Solid-State Circuits*, November 1992.

[22] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A System Perspective.* Addison-Wesley Publishing Company, Reading, Mass., 2nd edition, 1993.

[23] D. G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, October 1990.

[24] L. R. Dennison, W. J. Dally, and D. Xanthopoulos. Low-latency plesiochronous data retiming. In *Proceedings of the 16th Conference on Advanced Research in VLSI.* IEEE Computer Society, March 1995.

[25] R. D. Rettberg and L. A. Glasser. U. S. Patent 4,700,347, October 1987.

[26] David F. Rogers. *Procedural Elements for Computer Graphics.* McGraw-Hill Book Company, New York, 1985. pages 34–42.

[27] R. M. M. Oberman. A flexible rate multiplier circuit with uniform pulse distribution outputs. *IEEE Transactions on Computers*, August 1972.

[28] R. M. M. Oberman. *Counting and Counters.* John Wiley & Sons, New York, 1981.

[29] Leslie Lamport. LaTeX *: A Document Preparation System.* Addison-Wesley Publishing Company, Reading, Mass., 1986.

[30] Data I/O Corporation, Redmond, Washington. *UniSite User Manual*, 1992. page 7-25.

[31] M. Morris Mano. *Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1984. pages 356–357.

[32] D. E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, Boston, 1991.

[33] Signetics Company, Sunnyvale, California. *Signetics FAST Logic Data Handbook*, 1989.