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.

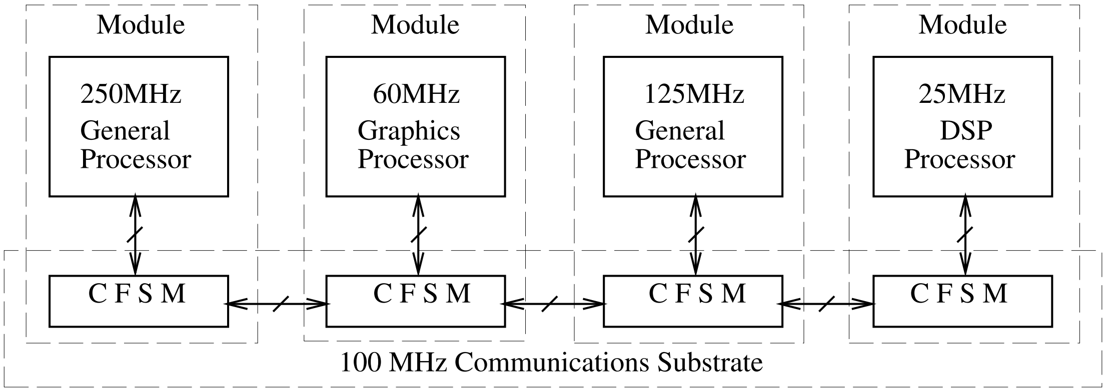

Figure 1-1: A simple NuMesh system with processors of different speeds.

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

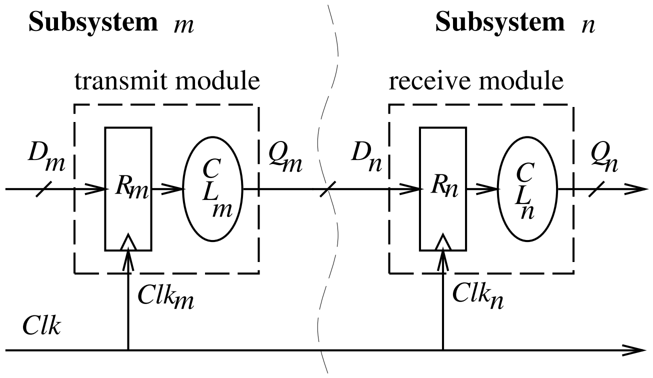

Figure 2-1: Communication in a synchronous system.

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

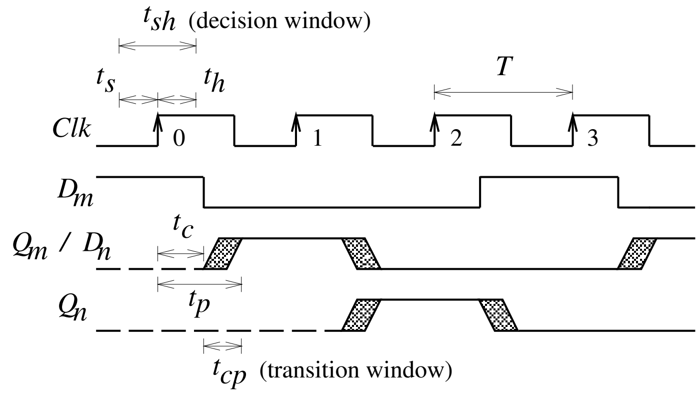

Figure 2-2: Synchronous data transfer and the timing parameters involved.

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.

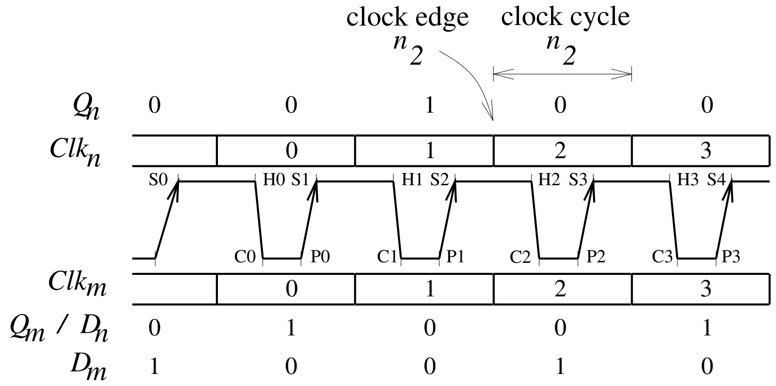

Figure 2-3: The schedule diagram of Figure 2-2.

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,

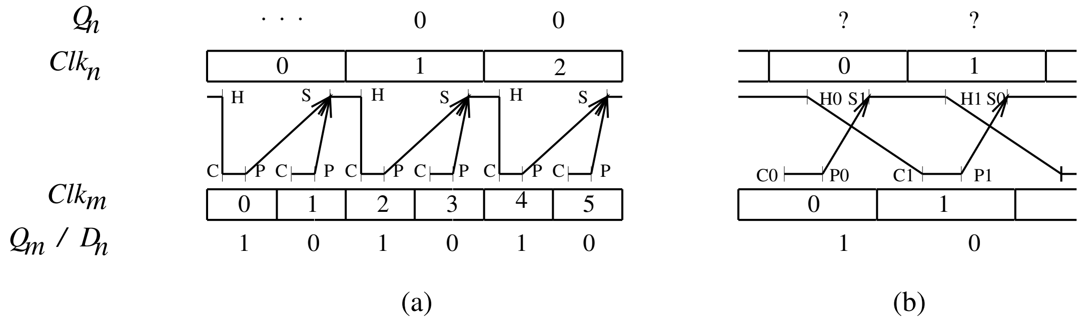

Figure 2-4: Problems with using different clocks. (a) Flow control. (b) Synchronization.

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

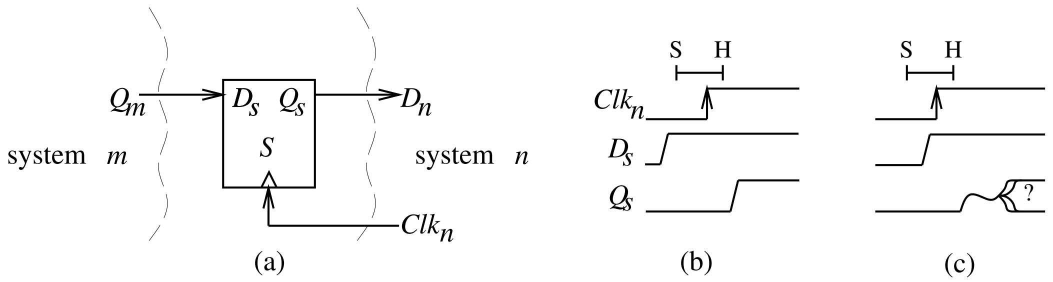

Figure 2-5: A synchronizer. (a) Implementation. (b) Normal operation. (c) Metastability.

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

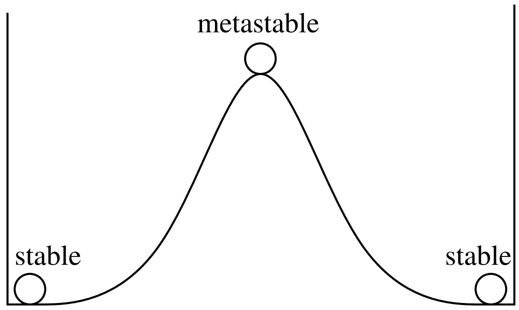

Figure 2-6: Ball and hill analogy for metastability.

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.