Rational Clocking

Luis F. G. Sarmenta Gill A. Pratt Stephen A. Ward

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Communication between independently-clocked digital subsystems typically involves a finite probability of synchronization failure whose minimization introduces delays and consequent performance costs. This paper explores a technique that eliminates both the inherent unreliability of such communication and the performance overhead it implies. Our approach maintains a known phase relationship between clocks whose frequencies are related by a rational factor, and exploits the predictability of their relative phases to algorithmically time communications without run-time arbitration contests.

1 Introduction

The designer of a digital system is often confronted with the need to establish communication between synchronous subsystems operating at different clock speeds. This problem occurs in heterogenous multiprocessor systems, and in uniprocessor systems where the processor and bus are clocked independently.

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 [1, 2, 3, 4]. Synchronous solutions are more efficient and reliable, but have traditionally lacked flexibility because they require the clock frequencies to be related by integer factors [6, 7].

In this paper, we present an efficient *and* flexible solution, called *rational clocking*, that allows 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, while assuring a periodic relationship between the two clocks that can be used to determine a schedule for transferring data between the two systems.

*A U.S. patent application has been filed on behalf of MIT covering portions of this work.

2 Basic Implementation

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

$$\frac{f_m}{f_n} = \frac{M}{N} \tag{1}$$

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:

$$T_{MN} = MT_M = NT_N \tag{2}$$

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).

Given the frequency ratio M/N, we can 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.

2.1 Hardware Implementation

Figure 1 shows a simple circuit implementing this idea. In this circuit, a phase-locked loop (PLL) and two frequency-dividing counters are used to obtain Clk_n from Clk_m while ensuring the desired phase and frequency relationships. (The characteristics of the PLL may limit the operational range of this circuit. However, this problem is orthogonal to the present discussion, and will not be discussed in this paper.) The values in the frequency-dividing counters indicate the current clock cycle (within the coincidence cycle) of their respective systems, and are used to index two lookup tables. Each lookup table produces two control signals per clock cycle number: TE (Transmit Enable), which controls the transmitting register, and RE (Receive Enable), which controls the receiving

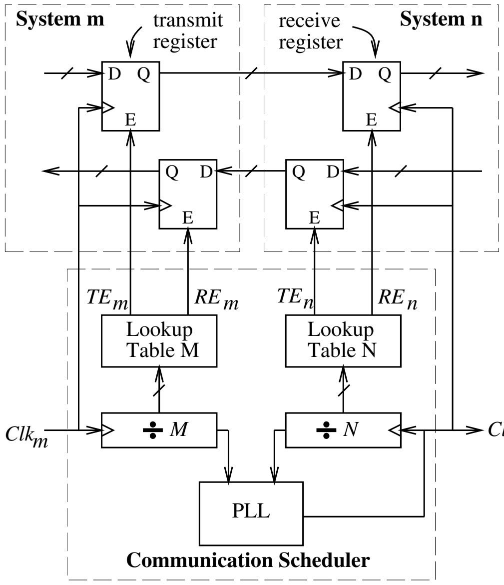

Figure 1: Interface hardware for rational clocking.

register. By programming the lookup tables appropriately, we ensure that the transmitter's output never violates the receiver's timing constraints, and effectively synchronize the transmitter's output to the receiver's clock.

This circuit can be used in the place of synchronizers for exchanging handshake signals in existing systems that rely on handshaking. Since timing violations are now impossible, synchronization delays are no longer necessary, and handshaking latency can be reduced significantly. Efficiency can further be improved by using this circuit for transferring actual data, and modifying each system to use its RE and TE signals instead of handshake signals for flow control. This eliminates the need to exchange signals for every data transfer, and makes it possible to achieve very high throughput for continuous data transfers.

2.2 Communication Scheduling

The appropriate lookup table entries can be computed using knowledge of the clock frequencies and the timing parameters of each system. These parameters are used to draw a *schedule diagram* such as that in Figure 2, which shows system m transmitting data to system n where $M = 5$ and $N = 6$.

In a schedule diagram, the clock cycles of each system are denoted by numbered boxes bounded by the active edges of the system's clock. A clock cycle's number corresponds to the counter value used to index the lookup tables during that cycle. 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

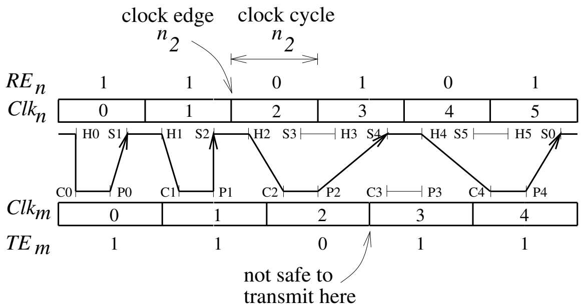

Figure 2: A schedule diagram.

name and cycle number. Clock cycle n_2 and its active edge are indicated in Figure 2 as examples.

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 call these lines the setup, hold, contamination, and propagation *edges* of the systems. Each of these edges is associated with some clock cycle and numbered accordingly as shown in Figure 2.

The setup and hold edges of the receiver (system n) define a *decision window* during which data at the receive register's input must be held stable as it is sampled by the receiver. The contamination and propagation edges of the transmitter (system m) define a *transition window* during which the output of the transmit register can change as a result of the transmitter latching data into it. 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 its inputs. (We assume that clock skew and phase jitter are accounted for by shifting and widening the windows appropriately.)

The following algorithm describes one way to schedule safe communications between two systems:

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 (i.e., coinciding in time with) 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}, 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. Then, set lookup table entries $TE_m[(i-1) \bmod M]$ and $RE_n[(j-1) \bmod N]$ to 1 to enable the transfer. Finally, draw a line from H_j to $C_{i'}$, indicating that it is safe to transmit on cycle $m_{i'}$ without contaminating the reception at clock edge n_j.

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 2 shows the result of using this algorithm with $P_{begin} = P_0$. As shown, sampling must be 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 must be 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 must be 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 2, 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* — 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. Since no short-cuts are possible, such an algorithm cannot produce a schedule that allows more data transfers per coincidence cycle than the greedy algorithm.

Trans.	Receive Frequency							
Freq.	1	2	3	4	5	6	7	8
1	100	100	100	100	100	100	100	100
2	100	100	100	100	100	100	100	100
3	100	100	100	100	100	100	100	100
4	100	100	100	100	100	100	100	100
5	100	100	100	100	100	80	100	100
6	100	100	100	100	80	100	83	100
7	100	100	100	100	100	83	100	85
8	100	100	100	100	100	100	85	100

Figure 3: Throughput efficiency table (in %).

2.3 Pipeline Compensation

For simplicity, we have assumed that new data is made available at the transmit register's input in the same cycle that TE is enabled. In actual systems where pipelining may not allow a system to react to TE within the same cycle, we can produce an appropriately *time-advanced* copy of TE by simply *rotating* the lookup tables accordingly.

2.4 Lookup Table Variations

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 lookup tables that are useful regardless of the actual frequencies being used, presuming that the timing parameters scale with the minimum clock period. Such *generic* lookup tables would take M and N as inputs and produce the signals needed for the frequency ratio M/N. They can be used together with presettable frequency-dividing counters to make a generic rational clocking circuit that would support different frequency ratios as needed in applications.

Another way to support different frequency ratios is with *programmable* lookup tables. We can have RAM-based tables on-chip, and then have these loaded *serially* at bootup time from an off-chip ROM containing the generic lookup table. This results in a circuit which is both faster and more space-efficient than one that uses the generic table directly. Programmable tables also give us the option of using software to compute the appropriate lookup table values at bootup.

2.5 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 2, the throughput efficiency was 4/5, or 80%. Figure 3 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%.

Trans. Freq.	Receive Frequency 1	2	3	4	5	6	7	8
1	100	100	100	100	100	100	100	100
2	0	100	100	100	100	100	100	100
3	100	100	100	66	66	100	100	100
4	100	50	66	100	75	100	100	100
5	100	100	66	75	100	60	80	80
6	100	100	33	100	60	100	66	66
7	100	100	100	75	60	66	100	71
8	100	100	100	50	60	66	71	100

Figure 4: Effect of doubling the timing parameters.

3 Double-Buffering

As Figure 3 shows, rational clocking provides good throughput efficiency for a variety of frequency ratios given typical timing constraints. As the timing windows are made wider, however, throughput not only deteriorates, it also begins to vary widely with the particular frequency ratio being used. This can be seen in Figure 4, where the timing parameters have been doubled (i.e., $S = 40\%$, $H = 20\%$, $C = 30\%$, $P = 40\%$). This can undesirably limit the usefulness and flexibility of rational clocking in high-performance systems.

This problem occurs because it is possible for one transmission to *contaminate* a previous transmission. In Figure 2, for example, system m cannot transmit on clock edge m_3 because doing so contaminates the transmission on clock edge m_2. We can avoid this problem easily by using 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 for new data, it then reads from the second register as well. Though simple, this *double-buffering* technique is highly effective and, in almost all practical cases, guarantees 100% throughput efficiency regardless of the specific timing parameters and frequency ratios used.

3.1 Why Double-Buffering Works

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. 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. This means that the transmitter can transmit on every cycle and the receiver can always receive the transmission without losing data — that is, we 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.

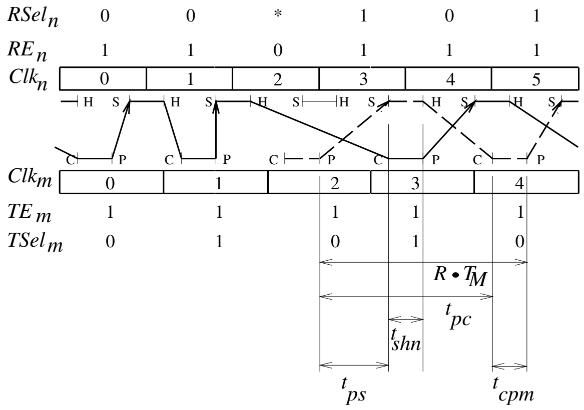

Figure 5: Computing R in the slower transmitter case.

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 simulate an unlimited supply of registers and eliminate the possibility of contamination.

Figure 5 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, we have

$$t_{pc} = R \cdot T_M - t_{cpm} \qquad (3)$$

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 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, we must have

$$t_{pc} \geq t_{ps} + t_{shn} \qquad (4)$$

Substituting the definition of t_{pc} from Equation 3, and rearranging terms, we get:

$$R \cdot T_M \geq t_{ps} + t_{shn} + t_{cpm} \qquad (5)$$

We can simplify this by noting that $t_{ps} < T_N < T_M$. Thus, Inequality 5 will be satisfied if

$$R \cdot T_M \geq T_M + t_{shn} + t_{cpm} \qquad (6)$$

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

$$R \geq \frac{t_{shn} + t_{cpm}}{T_M} + 1 \qquad (7)$$

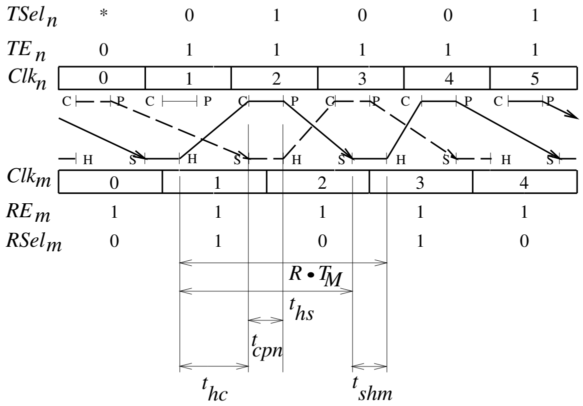

Figure 6: Computing R in the slower receiver case.

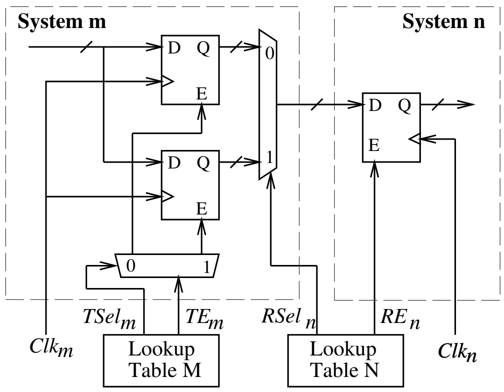

Figure 7: Double-buffered hardware.

This inequality shows that R depends directly on the widths of the timing windows. In the *ideal* case,

$$t_{shn} = t_{cpm} = 0 \tag{8}$$

and *one* register would be sufficient. In the *worst* case, we know that these windows cannot be wider than the clock periods of their respective systems. Thus,

$$t_{shn} \leq T_N < T_M, \quad t_{cpm} \leq T_M \tag{9}$$

and we would need *three* registers 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. In these cases, we have

$$t_{shn} + t_{cpm} \leq T_M \tag{10}$$

and can guarantee 100% throughput efficiency with only *two* transmit registers.

A similar analysis can be done for the case where the receiver is slower (i.e., system m is receiving). Figure 6 shows the time intervals involved. 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 7, and the conclusion that two transmit registers are enough to guarantee 100% throughput efficiency in the slower receiver case as well.

3.2 Hardware Implementation

Figure 7 shows the hardware used for double-buffering. There are now two transmit registers in each direction. (Only one direction is shown in the figure.) Data to be transmitted is fed into the inputs of both 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.

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, 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 when implementing this circuit, we must make sure that $RSel$ does not cause a setup or hold time violation in the receive register that would not have otherwise happened in a single-buffered configuration. Fortunately, since $RSel$ is synchronous with the receiver's clock, this restriction is not hard to meet. Also note that the delay through the multiplexer should be accounted for when computing the receiver's setup and hold times.

3.3 Communication Scheduling

The elimination of contamination simplifies communication scheduling by allowing us to schedule each transaction on the slower system *independently* from the transaction immediately following it.

For the slower transmitter case, this means we can simply draw arrows from each propagation edge to the next setup edge as long as we set $TSel$ and $RSel$ so that the two registers are used alternatingly. For the slower receiver case, we can use a similar algorithm that uses the hold and contamination edges instead. That is, we draw a line from some edge H_i 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}. Note that this is 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}.

Figures 5 and 6 show examples of how the slower transmitter algorithm and the lazy slower receiver algorithm respectively would be used. They also expose a potential problem in implementing these algorithms — if M is odd, we cannot alternate perfectly between the two registers. 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 examples shown, 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 2 and making lookup tables based on the resulting equivalent ratio instead.

4 Run-Time Scheduling

A lookup table for a particular ratio M/N, needs $O(\max(M,N))$ bits of memory. A generic table for numerators and denominators ranging from 1 to N needs $O(N^3)$ bits. This cubic growth rate constitutes a significant disadvantage of the table-based implementation in terms of cost, speed, and flexibility.

Fortunately, since the difference between the two clock frequencies is constant, the interacting timing windows will move relative to each other at a constant rate. This makes it possible to keep track of their positions *on-the-fly* using counters. This section shows how this is done for double-buffered communication.

4.1 The Algorithm

The algorithm used for run-time scheduling is a graphical algorithm similar to *Bresenham's algorithm* for drawing lines on raster displays [8]. It works with a *schedule plot* such as that in Figure 8, which shows the algorithm being used for scheduling receptions on system n in the double-buffered slower transmitter case shown in Figure 5.

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 function $f(t)$, equal to the time that system m's transmit register must clock-in new data for system n to properly sample the data by clocking the receive register at time t, is plotted on the grid. As shown,

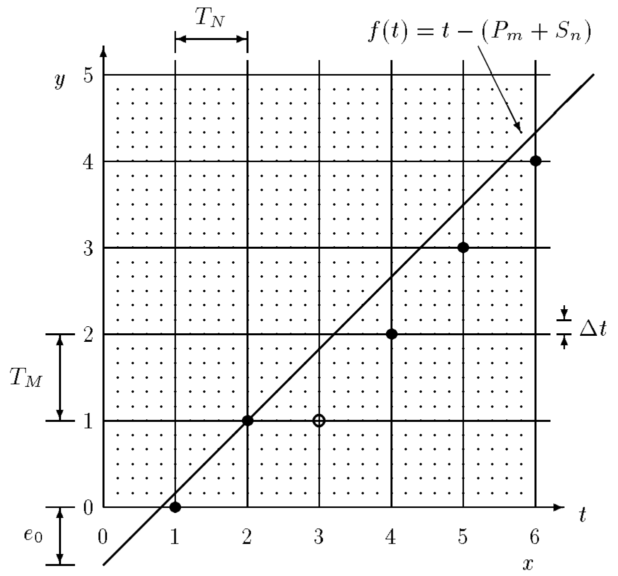

Figure 8: A schedule plot.

$$f(t) = t - (P_m + S_n) \tag{11}$$

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 8 has both P_m and S_n equal to 2, and thus has $f(t) = t - 4$.

The schedule plot allows us to schedule receptions on system n with a simple graphical algorithm: for each vertical line x, we plot a dot on the nearest horizontal line 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 identifying the earliest x at which $f(t)$ is at or above the horizontal line y, this algorithm identifies the nearest system n setup edge S_x at or after each system m propagation edge P_y, and is equivalent to the greedy slower transmitter algorithm from section 3.3. It can be implemented as follows:

1. Initialize all entries of $TSel_m$, TE_m, $RSel_n$, and RE_n to 0.
2. $x = x_0 = 1$
3. $f_0 = \lfloor -(P_m + S_n) \rfloor$

4. $y_0 = f_0 \text{ div } N$
5. **if** $(f_0 \geq 0)$ **or** $(f_0 \bmod N = 0)$ **then**
   $y_0 = y_0 + 1$
6. $y = y_0$
7. $e_{old} = e_0 = (f_0 \bmod N) - N$
8. $R = 0$
9. **repeat** N **times**
10. $\quad e = e_{old} + M$
11. $\quad$ **if** $(e \geq 0)$ **then**
12. $\quad\quad TSel_m[(y-1) \bmod M] = R$
13. $\quad\quad TE_m[(y-1) \bmod M] = 1$
14. $\quad\quad RSel_n[(x-1) \bmod N] = R$
15. $\quad\quad RE_n[(x-1) \bmod N] = 1$
16. $\quad\quad R = \overline{R}$
17. $\quad\quad y = y + 1$
18. $\quad\quad e = e - N$
19. $\quad x = x + 1$
20. $\quad e_{old} = e$

Here, the variable x indicates the system n clock edge (vertical line) currently being considered. It is initialized to 1 in line 2, and incremented at the end of each loop iteration.The variable y indicates the next system m clock edge (horizontal line) from which data has *not* yet been received. It is initialized in lines 4 to 6 so that it points to the first horizontal line *above* $f(t=0)$, and incremented after a system n clock edge that can receive the data has been found.

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 n_x. If e is negative, then system n would be too early and reception must be postponed by at least $|e|\Delta t$ seconds to give data enough time to be available and stable. If e is nonnegative, then data is already stable and can be received on edge n_x.

The initial error, e_0, is computed in line 7 relative to y_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 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.

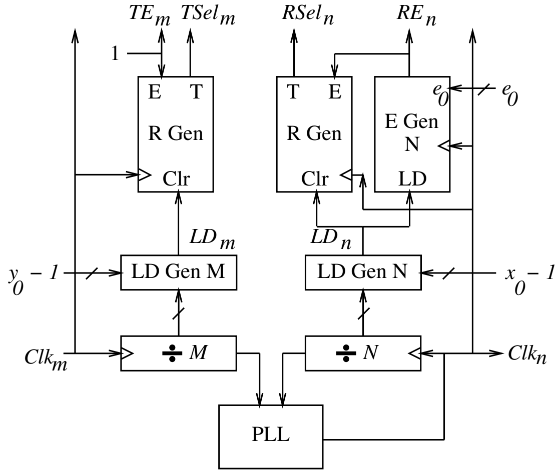

Figure 9: Run-time scheduler for double-buffering. (Only one direction shown.)

The same 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:

$$P_m \leftarrow H_m, \quad S_n \leftarrow -C_n \tag{12}$$

where H_m and C_n are the hold and contamination times expressed in units of Δt. These transformations allow the algorithm to determine the nearest contamination edge C_x at or after each hold edge H_y. The assignments in lines 12 to 15 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. In this way, the lazy algorithm can be implemented by simply feeding the appropriate values for P_m and S_n.

4.2 Hardware Implementation

Figure 9 shows the run-time scheduling hardware. This circuit uses three modules: *R Gen*, *E Gen*, and *LD Gen*.

The *R Gen* module stores the value of R, and is simply a toggle (T) flip-flop with synchronous enable (E) and clear (Clr) inputs.

The *E Gen* module, shown in Figure 10, generates the RE_n and TE_n signals for the faster system (in the slower system, these signals are simply tied to logical 1), and is a direct implementation of the loop in lines 9 to 20 of the algorithm. 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.

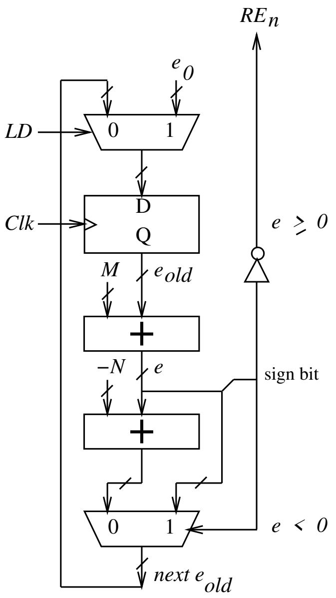

Figure 10: The *E Gen* module.

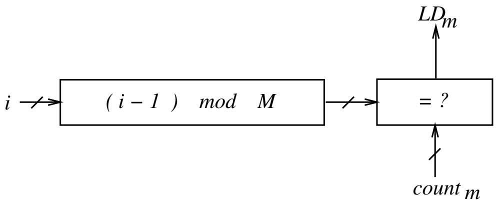

Figure 11: The *LD Gen* module (of system m).

The *LD Gen* module, shown in Figure 11, is used for periodically initializing the *R Gen* and *E Gen* modules at appropriate times. It takes the counter value and a constant i as inputs, and produces a load signal that causes the connected components to initialize at the ith clock edge of every coincidence cycle.

As shown in Figure 9, we take into account the initial values of y and x, as well as the one cycle subtractions performed in lines 12 to 15 of the algorithm, by setting the *LD Gen* module of systems m and n to initialize on clock cycles $y_0 - 1$ and $x_0 - 1$ respectively. The *LD Gen* module can also be used for pipeline compensation by adjusting x_0 and y_0 to reflect the necessary time advance.

The speed of the run-time scheduling hardware 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, it is probably best to use a fast RAM-based implementation loaded at bootup by run-time scheduling hardware clocked at a slower rate.

5 Summary

The rational clocking technique described here affords many of the advantages of single-clock synchronous systems while providing the flexibility to choose clock frequencies nearly independently. Two implementation approaches have been presented. The first, based on a precompiled scheduling table, may be interesting in applications involving small M and N, or where it is possible to use programmable tables that can be initialized by software on bootup. The second run-time implementation is remarkably space-efficient, requiring only $O(\lg N)$ bits of memory instead of the $O(N^3)$ required by the table-based implementation, and taking the timing parameters as inputs as well at no extra cost. Used alone or with programmable lookup tables, this technique makes it possible to design a versatile "off-the-shelf" rational clocking chip, or to integrate the rational clocking hardware directly into target applications. Other variations of the rational clocking technique exist and are discussed in [9].

References

[1] T. J. Chaney and C. E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, April 1973.

[2] W. Fleischhammer and O. Dörtok. The Anomalous Behavior of Flip-Flops in Synchronizer Circuits. *IEEE Transactions on Computers*, March 1979.

[3] M. Pěchouček. Anomalous Response Times of Input Synchronizers. *IEEE Transactions on Computers*, Feb. 1976.

[4] G. R. Couranz and D. F. Wann. Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region. *IEEE Transactions on Computers*, June 1975.

[5] W. K. Stewart and S. A. Ward. A Solution to a Special Case of the Synchronization Problem. *IEEE Transactions on Computers*, Jan. 1988.

[6] M. Afghahi and C. Svensson. Peformance of Synchronous and Asynchronous Schemes for VLSI Systems. *IEEE Transactions on Computers*, July 1992.

[7] F. Anceau. A Synchronous Approach for Clocking VLSI Systems. *IEEE Journal of Solid-State Circuits*, Feb. 1982.

[8] D. F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill Book Company, 1985.

[9] Luis F. G. Sarmenta. *Synchronous Communication Techniques for Rationally Clocked Systems*. Master's Thesis, Department of Electrical Engineering and Computer Science, MIT. May 1995.