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

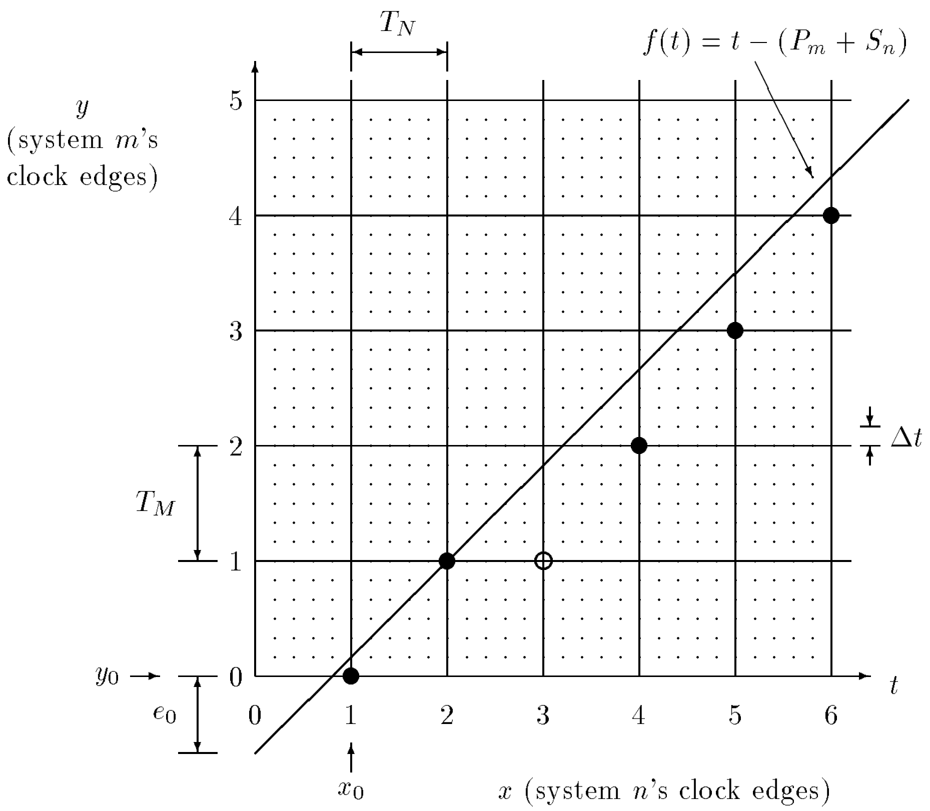

Figure 5-1: A schedule plot for the case where $M = 5$ and $N = 6$.

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. $x = 1$ $\quad\triangleright$ $x_0 = 1$
3. $f_0 = \lfloor -(P_m + S_n) \rfloor$ $\quad\triangleright$ get $f(t)$ at clock edge $x_0 - 1$
4. $y = f_0$ div N $\quad\triangleright$ 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. $\quad e = e_{old} + M$ ▷ compute error term for current x
10. $\quad$ **if** $(e \geq 0)$ **then** ▷ is new data available?
11. $\quad\quad TSel_m[(y-1) \bmod M] = R$ ▷ yes, transmit data
12. $\quad\quad TE_m[(y-1) \bmod M] = 1$
13. $\quad\quad RSel_n[(x-1) \bmod N] = R$
14. $\quad\quad RE_n[(x-1) \bmod N] = 1$
15. $\quad\quad R = \overline{R}$ ▷ use other register next time
16. $\quad\quad y = y + 1$ ▷ wait for next system m clock edge
17. $\quad\quad e = e - N$ ▷ make e relative to new y
18. $\quad x = x + 1$ ▷ get next x
19. $\quad e_{old} = e$ ▷ 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

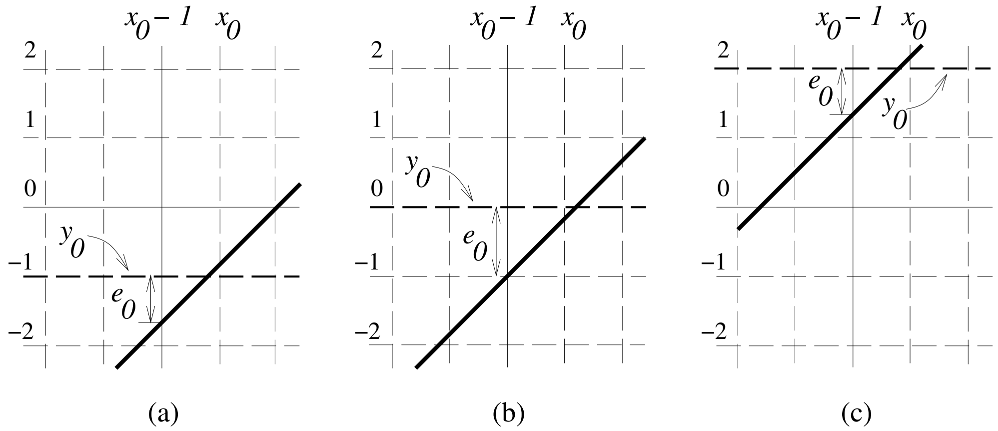

Figure 5-2: Three cases in computing y_0 and e_0. (a) $f_0 < 0$. (b) $(f_0 \bmod N) = 0$. (c) $f_0 \geq 0$

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

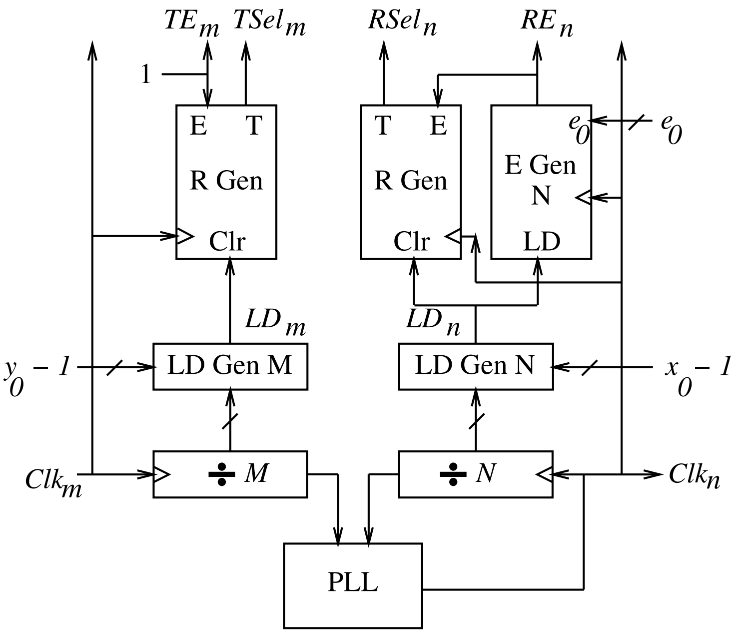

Figure 5-3: Basic run-time scheduler. (Only one direction shown.)

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.

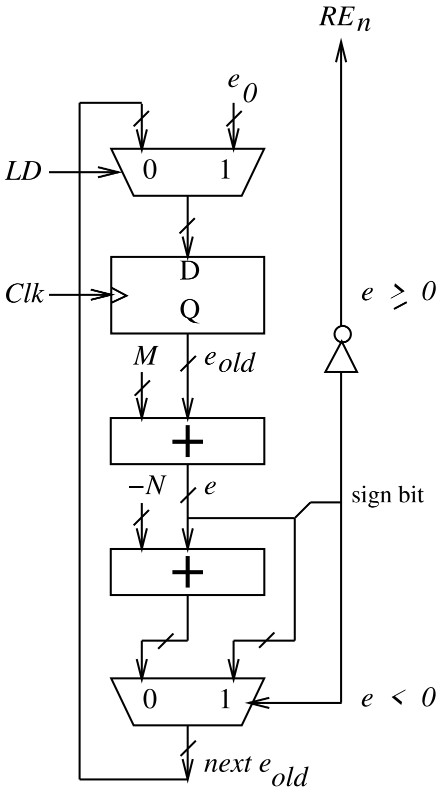

Figure 5-4: The *E Gen* module.

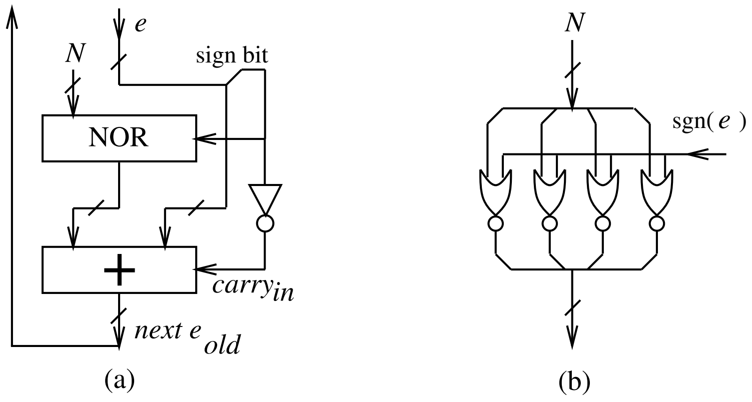

Figure 5-5: Using a NOR module instead of a mux in computing the next e_{old}.

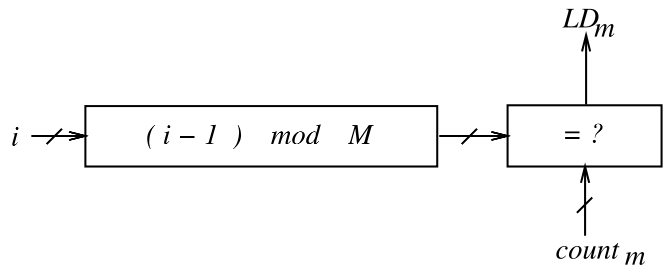

Figure 5-6: The *LD Gen* module.

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.

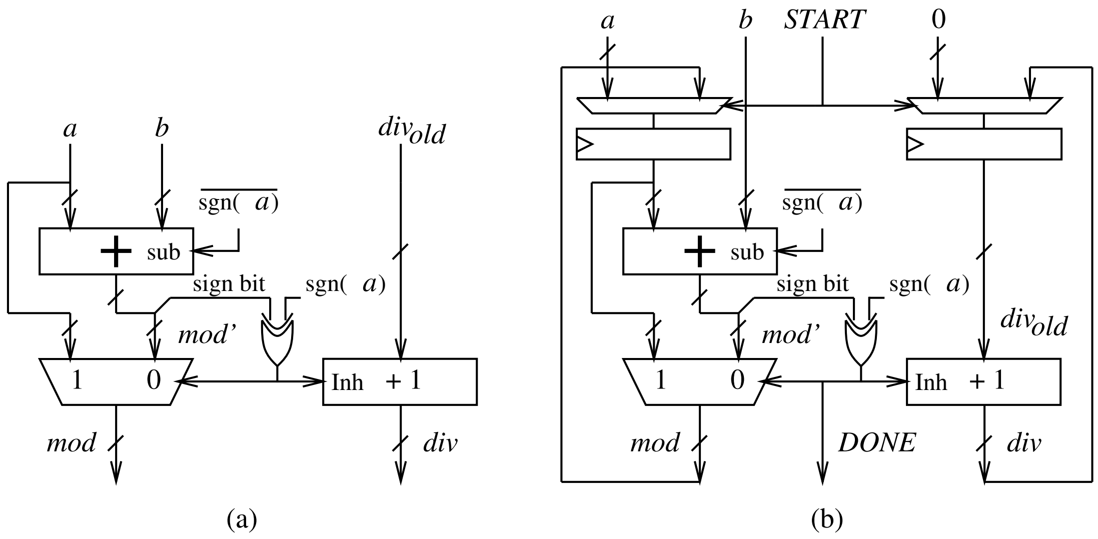

Figure 5-7: Computing mod and div. (a) The *MOD* module. (b) A sequential circuit.

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

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

and the *div* output would be

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

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.

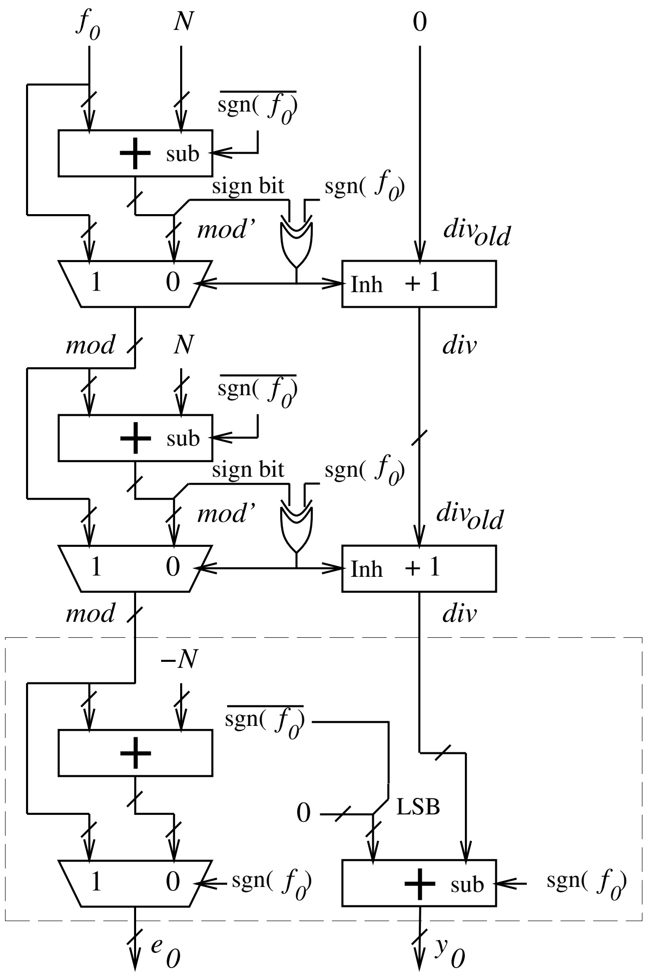

Figure 5-8: Combinational circuit for computing y_0 and e_0.

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

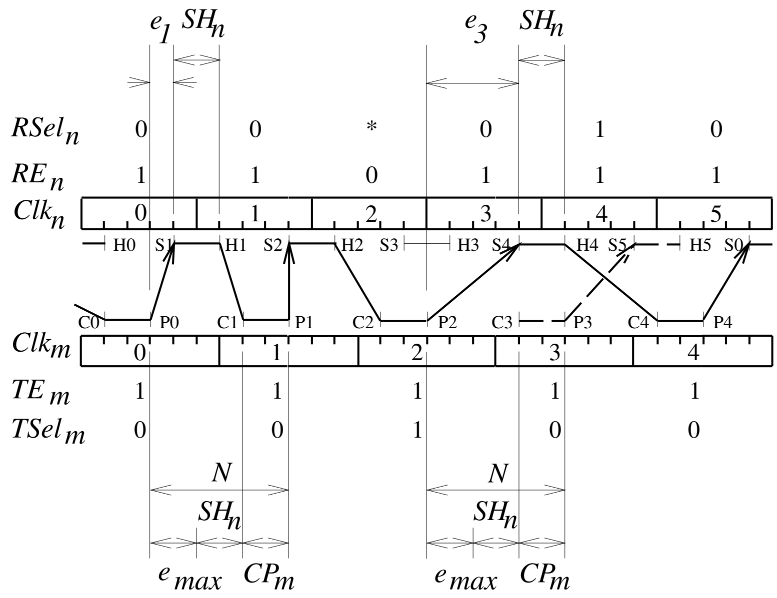

Figure 5-9: Checking for contamination using e.

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:

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

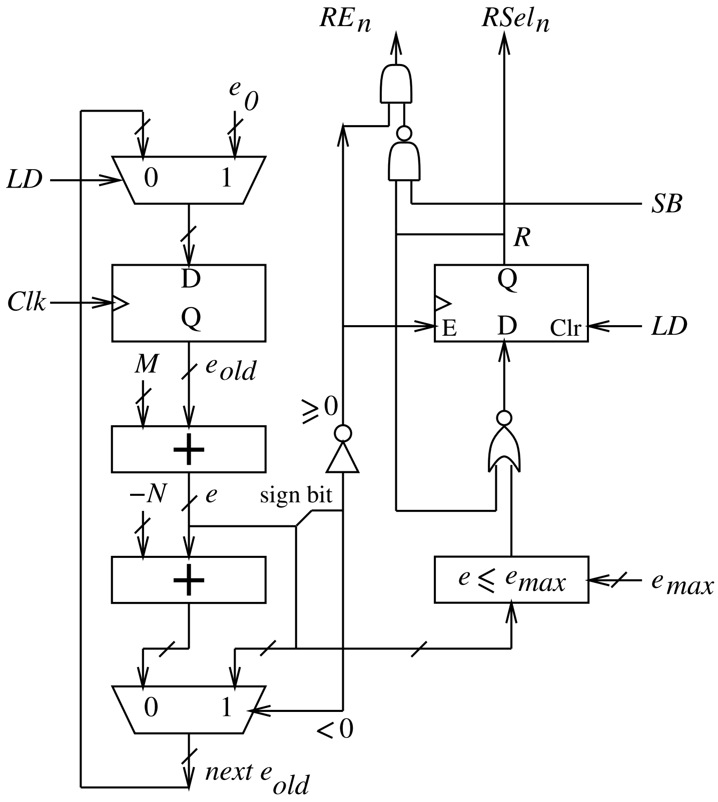

Figure 5-10: Scheduling hardware for the faster system.

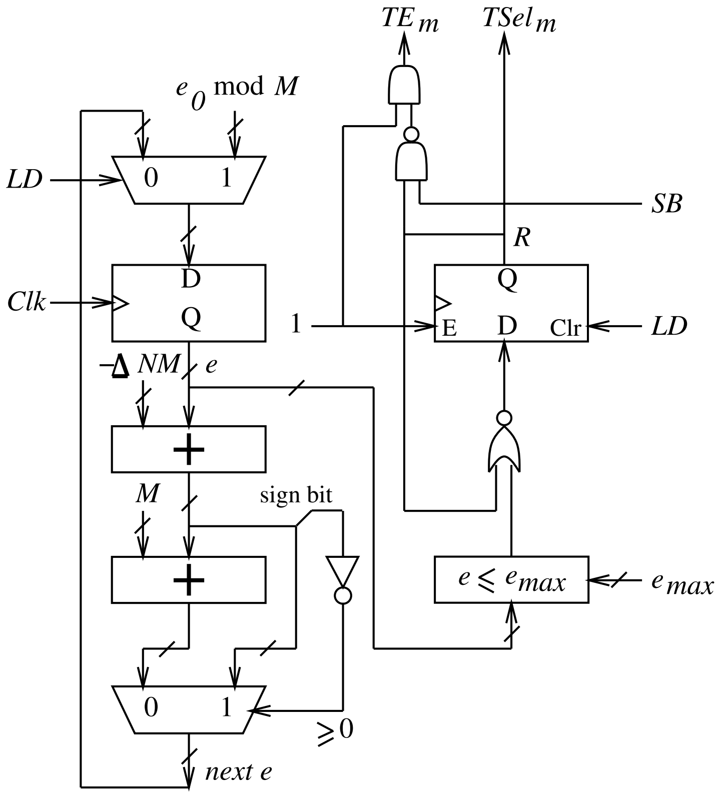

Figure 5-11: Scheduling hardware for the slower system.

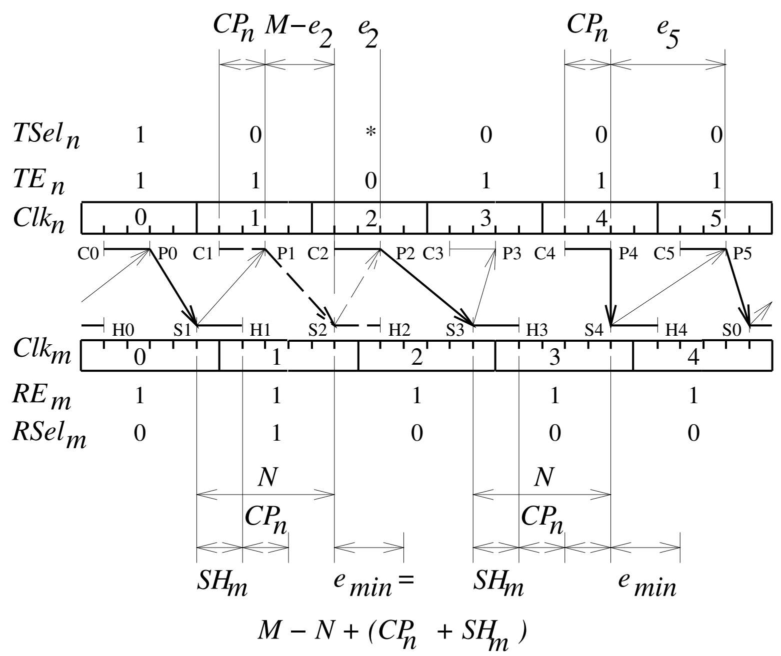

Figure 5-12: Checking for contamination in the slower receiver case.

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

$$M - e \leq N - (CP_n + SH_m) \tag{5.15}$$

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: $N - M > CP_n + SH_m$

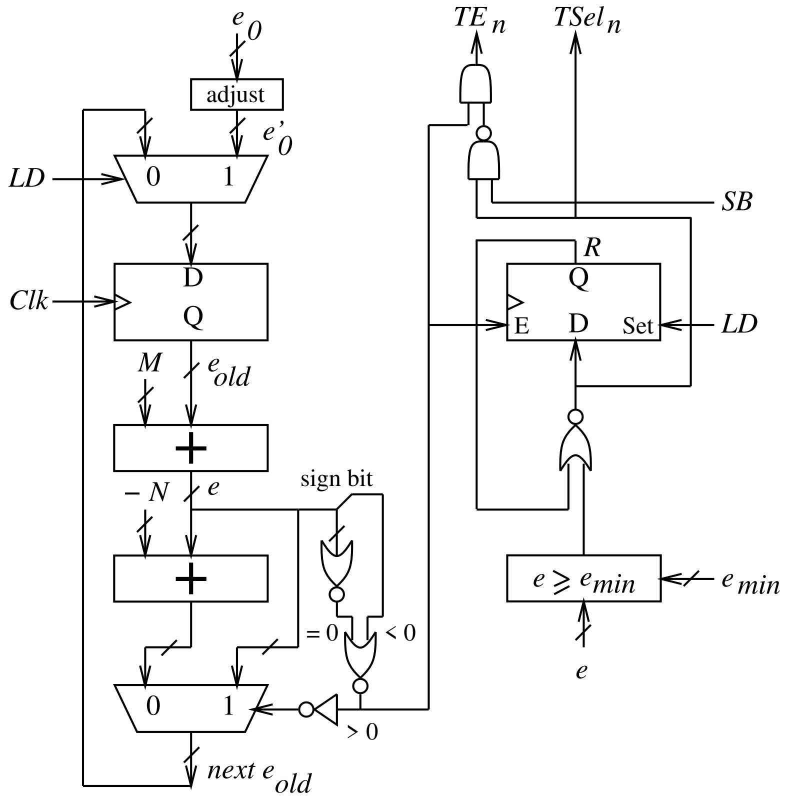

Figure 5-13: Faster system's hardware for the greedy slower receiver algorithm.

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 mod 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 mod M as input, where e'_0 is the output of the faster system's adjust module.

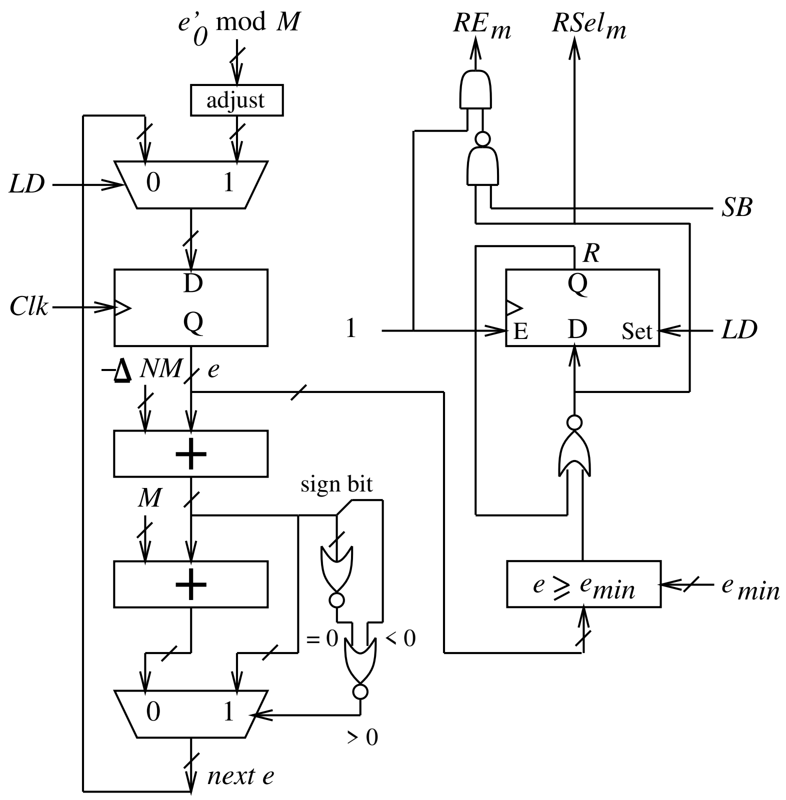

Figure 5-14: Slower system's hardware for the greedy slower receiver algorithm.

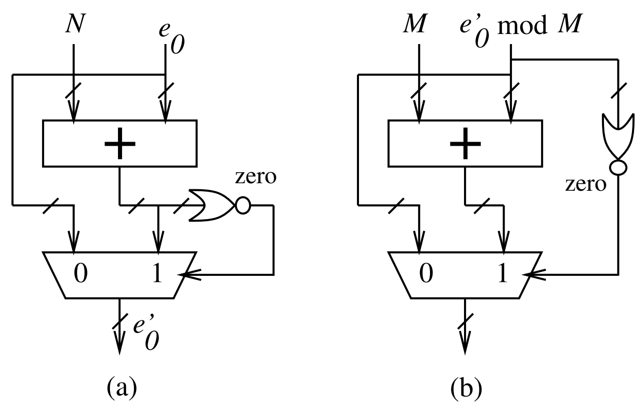

Figure 5-15: Adjusting initial errors. (a) For the faster system. (b) For the slower system.

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.