Chapter 3

Rational Clocking

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

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

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

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

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

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

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

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

3.1 Clock Generation

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

3.1.1 Clock Division

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

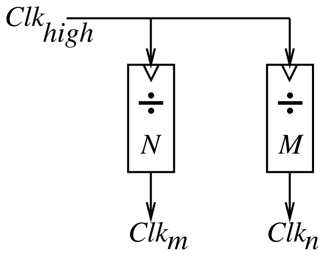


Figure 3-1: Generating rationally-related clocks through clock division.

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

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

$$N f_m = M f_n = f_{high} \tag{3.3}$$

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

3.1.2 Clock Multiplication

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

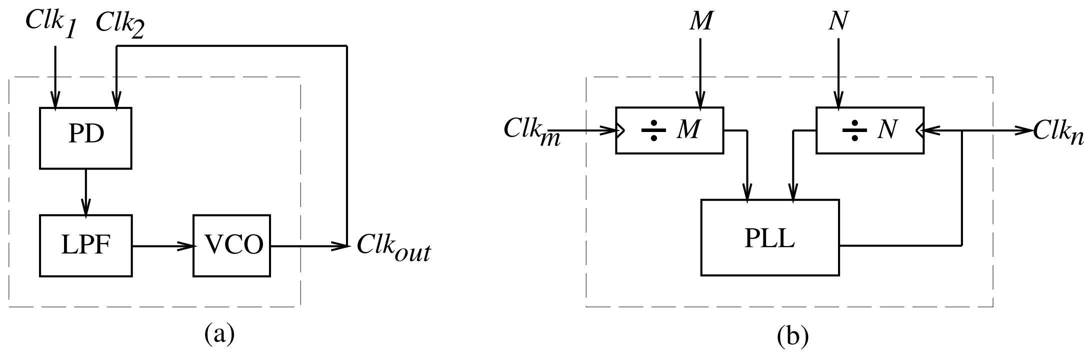


Figure 3-2: Clock multiplication. (a) A PLL. (b) A programmable clock multiplier.

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

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

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

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

3.1.3 Comparing Clock Division and Clock Multiplication

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

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

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

3.2 Table-Based Communication Control Hardware

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

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

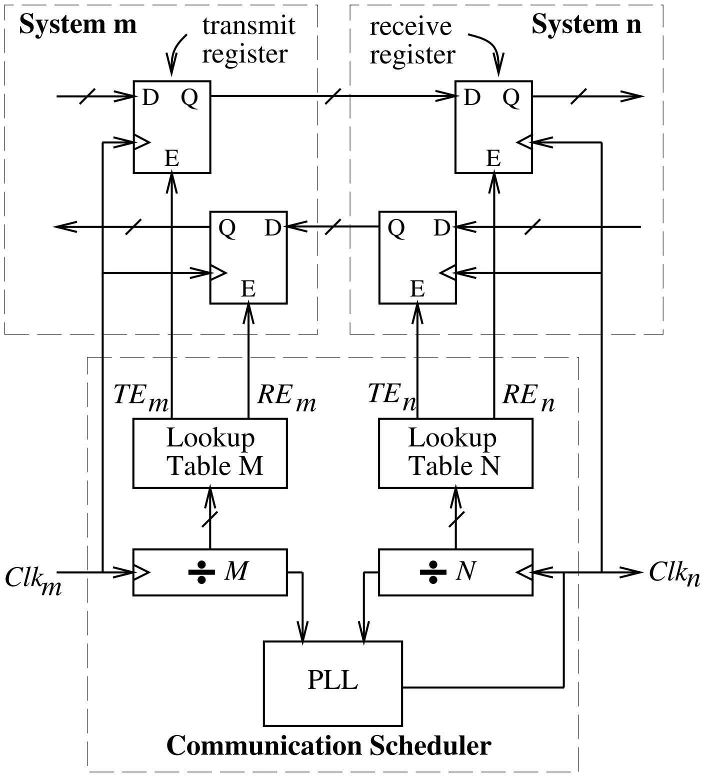


Figure 3-3: Interface hardware for rational clocking.

between systems.

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

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

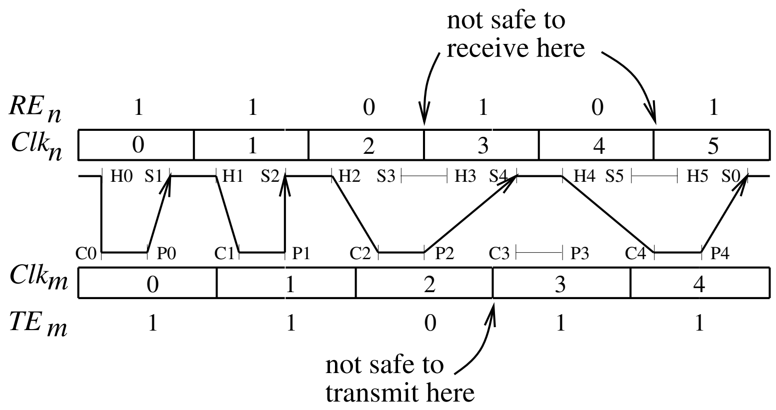


Figure 3-4: Transmitting from system m to system n where $M = 5$ and $N = 6$.

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

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

3.3 Communication Scheduling

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

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

1. Draw a schedule diagram indicating the relevant timing windows of the two systems.

2. Start with some propagation edge P_i (of some cycle m_i) and call this edge P_{begin}.

3. Move from P_i to the nearest setup edge S_j (of some cycle n_j) at or after it. Use modular arithmetic when "moving" from one edge to another. That is, if you go past the rightmost edge of the diagram, "wrap-around" to the leftmost edge. This reflects the periodic nature of the communication patterns.

4. Move from S_j to its corresponding hold edge H_j.

5. Move from H_j to the nearest contamination edge $C_{i'}$ at or after it.

6. Move from $C_{i'}$ to its corresponding propagation edge $P_{i'}$.

7. If $P_{i'}$ is *past* P_{begin} (i.e., you passed over P_{begin} in the process of moving from P_i to $C_{i'}$), then the scheduling is done.

8. If $P_{i'}$ is not past P_{begin}, then it is safe to transmit data from P_i to S_j. Draw an arrow from P_i to S_j (indicating that it is safe to transmit on cycle m_i and receive on cycle n_j) and a line from H_j to $C_{i'}$ (indicating it is safe to transmit on cycle $m_{i'}$ without contaminating the reception at clock edge n_j). Then, set lookup table entries $TE_m[(i-1) \bmod M]$ and $RE_n[(j-1) \bmod N]$ to 1 to enable the transfer.

9. If $P_{i'} = P_{begin}$, the scheduling is done. If not, make $P_{i'}$ the new P_i and go back to step 3.

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

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

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

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

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

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

3.4 Performance

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

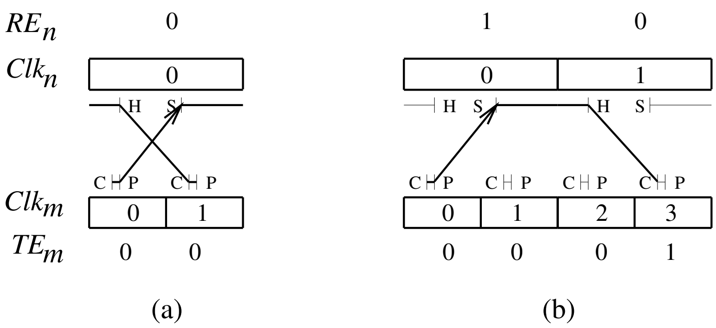


Figure 3-6: Equivalent ratios may not be equally efficient. (a) $M/N = 2/1$. (b) $M/N = 4/2$.

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

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

3.5 Variations and Improvements

3.5.1 Generic Lookup Tables

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

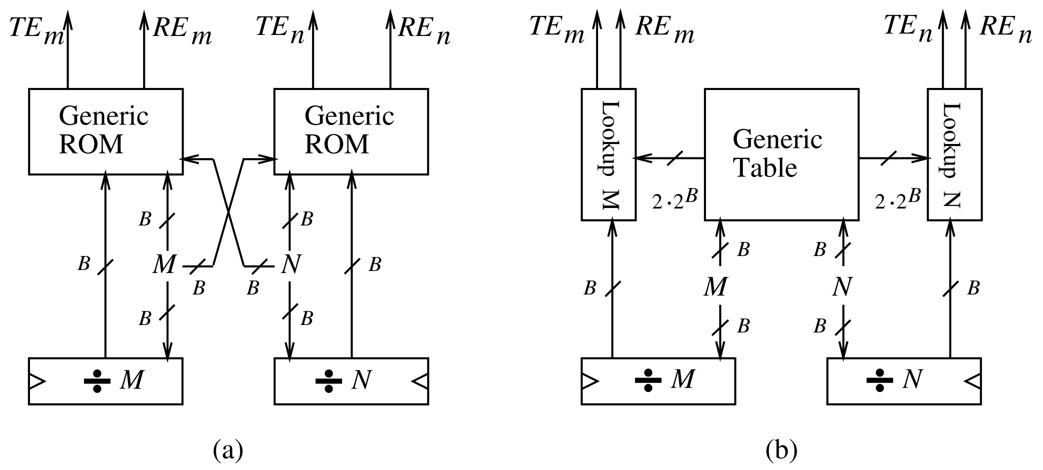


Figure 3-7: Generic lookup tables. (a) Using two ROMs. (b) Using a single table.

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

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

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

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

3.5.2 Reducing the Generic Table Size

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

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

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

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

3.5.3 Programmable Lookup Tables

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

3.5.4 Reaction Latency Compensation

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

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

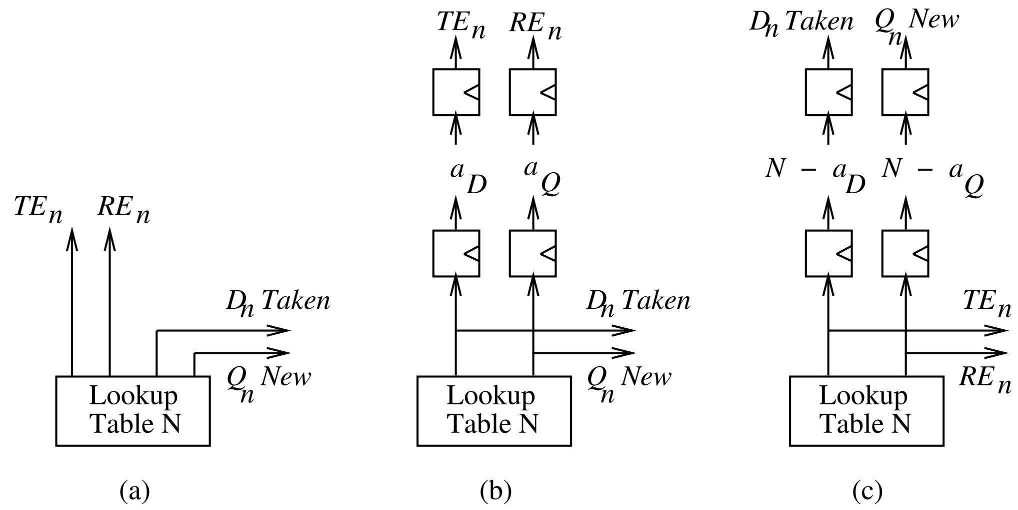


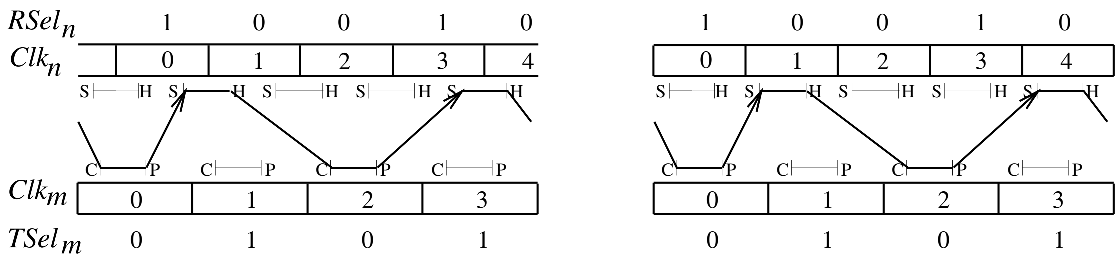
Figure 3-8: Interface hardware with reaction latency compensation.

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

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

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

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



(a) (b)

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

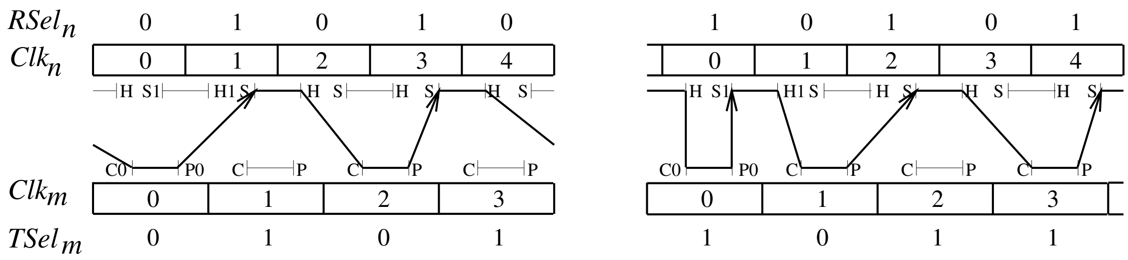
and a_D are implemented as delays of $N - a_Q$ and $N - a_D$ respectively. This circuit has the advantage that the lookup tables can be programmed without knowledge of how much reaction latency compensation is needed by the target application. Different amounts of delay needed for $Q\ New$ and $D\ Taken$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

3.5.5 Non-coincident Clocks

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

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

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



(a) (b)

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

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

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