Chapter 4

Double-Buffering

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

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

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

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

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

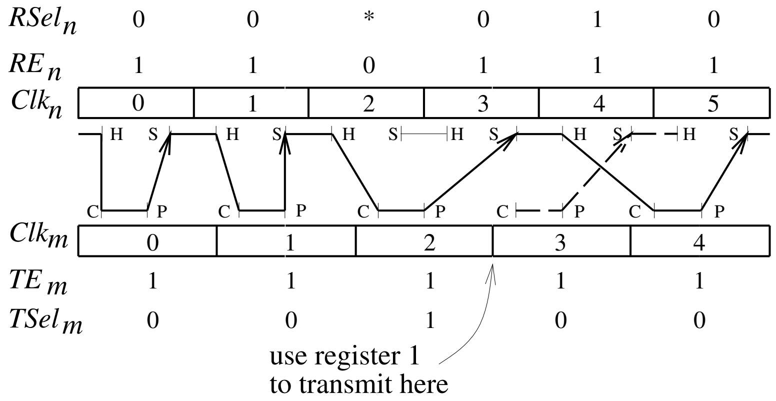

Figure 4-2: Getting 100% throughput efficiency using a second transmit register.

ratios and timing parameters.

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

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

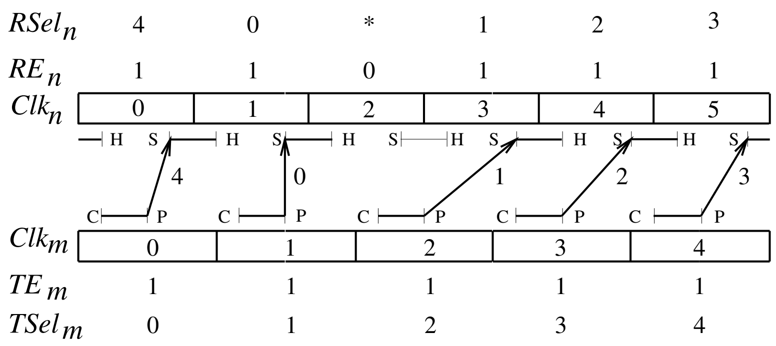

Figure 4-3: Avoiding contamination by using a new register for every transmission.

4.1 Why Double-Buffering Works

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

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

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

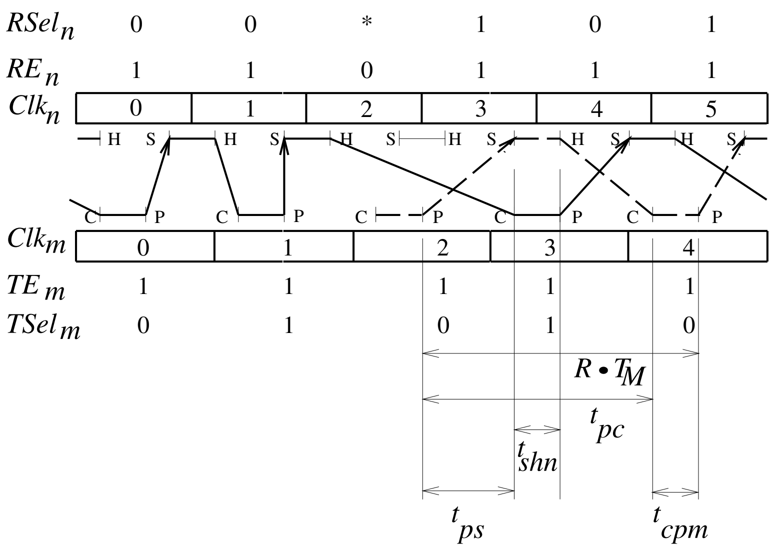

Figure 4-4: Determining R in the slower transmitter case.

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

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

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

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

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

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

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

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

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

and Inequality 4.3 will be satisfied if

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

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

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

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

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

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

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

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

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

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

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

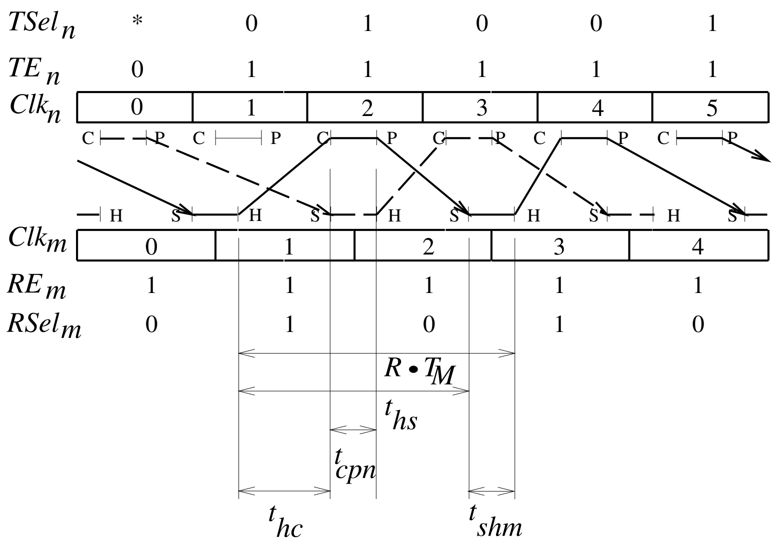

Figure 4-5: Determining R in the slower receiver case.

4.2 Communication Scheduling

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

4.2.1 Slower Transmitter Case

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

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

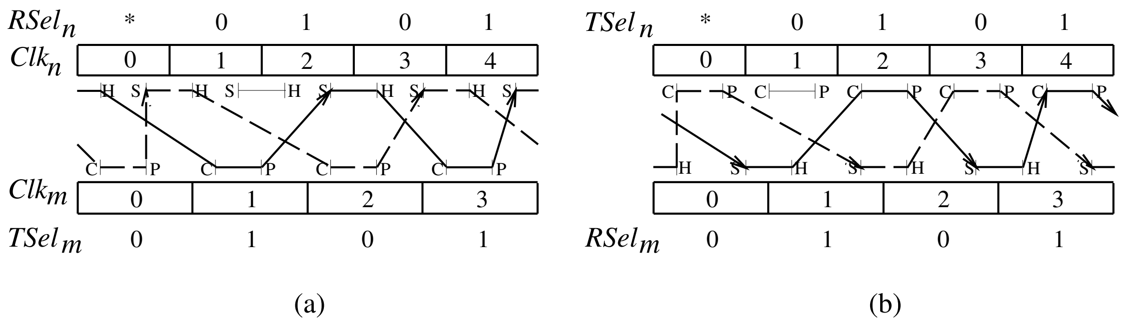

Figure 4-6: Scheduling for double-buffering. (a) Slower transmitter case. (b) Slower receiver case (lazy algorithm).

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

4.2.2 Slower Receiver Case

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

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

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

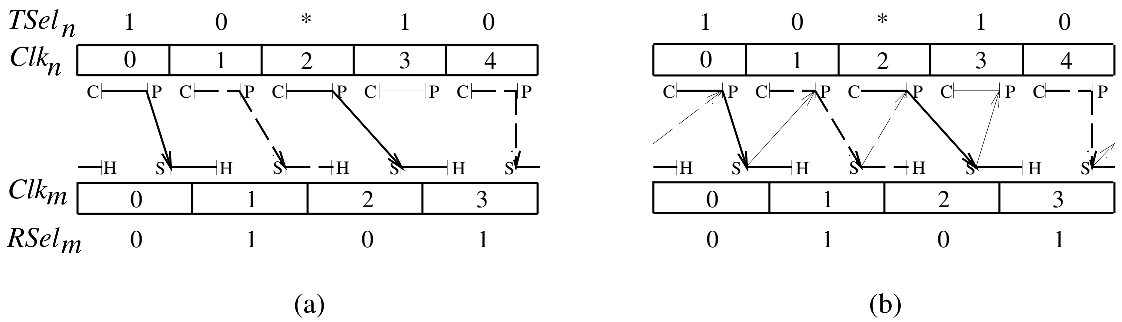

Figure 4-7: The greedy slower receiver algorithm. (a) In reverse time. (b) In normal time.

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

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

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

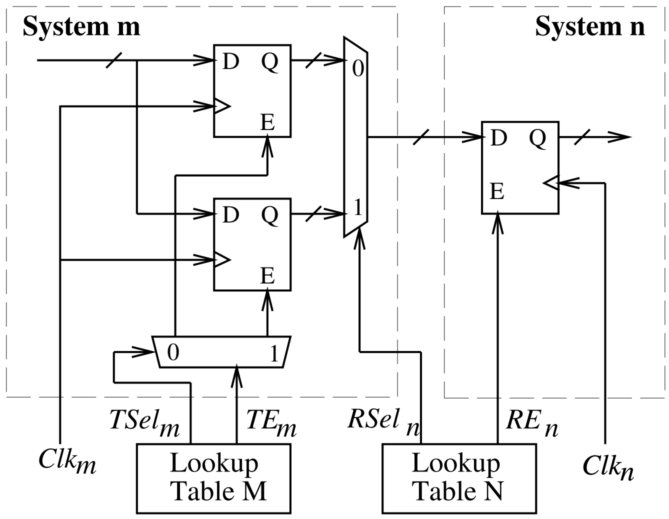

Figure 4-8: Double-buffered hardware (only one direction shown).

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

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

4.3 Hardware Implementation

4.3.1 Register Selection

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

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

4.3.2 Communication Scheduling

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

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

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

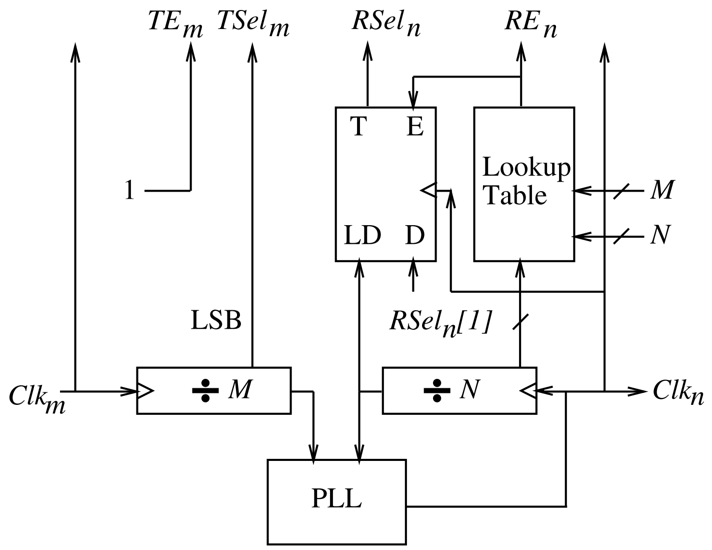

Figure 4-9: Modified scheduling hardware (only one direction shown).

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

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

4.4 Variations

4.4.1 Triple and Quadruple Buffering

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

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

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

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

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

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

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

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

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

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

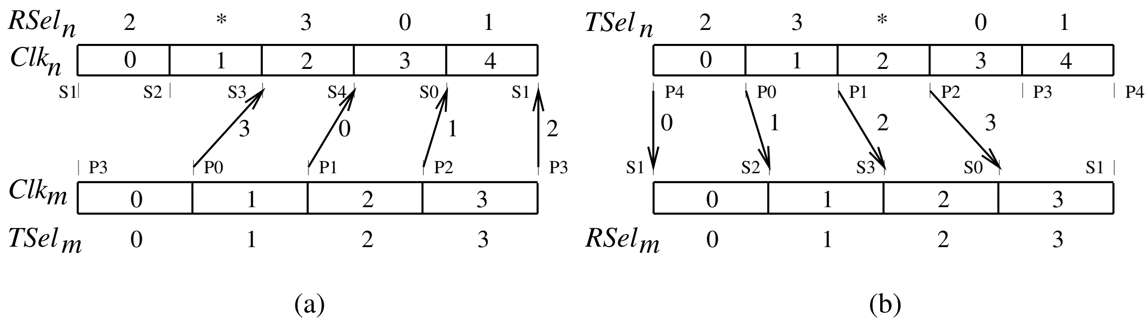

Figure 4-10: Using four buffers. (a) Slower transmitter case. (b) Slower receiver case (greedy algorithm).

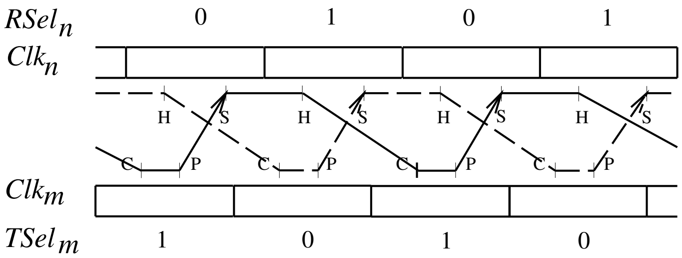

Figure 4-11: Using double-buffering for communication between out-of-phase systems.

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

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

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

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

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