Chapter 6

Simulation and Implementation

The techniques and ideas presented in this thesis were developed and tested with the help of different software and hardware tools. First, communication scheduling software was written to generate lookup tables for different frequency ratios, timing parameters, and scheduling algorithms. This software was used to measure throughput and to test the different algorithms. Then, a CMOS VLSI integrated circuit using programmable lookup tables was designed and fabricated to facilitate actual testing of the rational clocking technique. This chip was used in a breadboard-based circuit that verified and demonstrated the usefulness of the rational clocking techniques and algorithms. Finally, a model of the run-time scheduling was constructed and simulated using Verilog to verify the correctness of the proposed hardware.

This chapter provides functional descriptions of these tools, and discusses various issues that came up during their development and use. Further details about their implementation, including source code and a VLSI layout, are provided in the Appendices.

6.1 Communication Scheduling Software

The communication scheduling software is a text-based menu-driven program written in C that takes timing parameters and frequency ratios as inputs, and produces corresponding data transfer schedules. It can be used for studying scheduling algorithms, and for generating source files for the lookup table ROMs. This section provides a functional description of the different features of the software, and presents some recommendations for improvements. A complete listing of the source code and more detailed discussion about different issues involving the design of the software are available in Appendix B.

Figure 6-1 shows the scheduling software's menu tree. At the top level, the user can either enter the timing parameters, generate output, or change the program options. The meaning of the timing parameters in the first case, and the format of the output generated in the second case, are determined by the options available under the Options Menu. These options are discussed in the following subsections.

6.1.1 Timing Parameter Units

In the scheduling software, time is expressed in integer units of dt, defined as:

$$dt = \Delta t/d \tag{6.1}$$

where

$$\Delta t = T_{MN}/MN \tag{6.2}$$

as defined in Chapter 5, and d is some integer specified by the user. Using these units and a sufficiently large value for d, we can express time with arbitrarily fine precision without the use of floating-point numbers (assuming d does not get so large that the times we want to express do not fit in a 32-bit "long" integer). As discussed further in Appendix B, this not only allows us to avoid rounding-errors associated with floating-point numbers, but also makes it easier to implement the scheduling algorithms.

As Figure 6-1 shows, the software allows the user to express the timing parameters in five different ways. The first two ways (dt and Δt) are self-explanatory. The third ($1/dt$) allows the user to express the timing parameters as fractions of a clock period. For example, if system m's propagation delay, P_m, is 20% of the minimum clock period, and d is equal to its default value of 1000, then the user should enter 200 when asked for P_m. As discussed in section 3.5.1, expressing the timing parameters in this way is useful if we

Main Menu

- Enter Parameters
- Generate Output
- Options Menu

Options Menu

- Parameter Units
  - dt
  - Δt
  - $1/d$ of a clock period
  - Absolute Units with Clock Division
  - Absolute Units with Clock Multiplication
- Scheduling Algorithm
  - Original Single-Buffered
  - Generalized Run-Time (2 buffers)
  - Basic Run-Time Double-Buffered (Greedy)
  - Basic Run-Time Double-Buffered (Lazy)
- Compute Mode
  - Pair
  - Table
- Output Format
  - Text
  - LaTeX schedule diagram
  - LaTeX schedule plot (not implemented)
  - ROM
- Output Options
  - Log file name
  - Show intermediate results?
  - Draw text schedule diagram?
  - Use with prototype chip?

Figure 6-1: The communication scheduling software's menu options.

assume that rational clocking will be used with systems whose timing parameters scale with their minimum clock periods.

The last two ways to express timing parameters, are useful when the timing parameters are fixed, or *absolute*, and do *not* scale with the clock period. The fourth option assumes that clock division is being used, and asks the user for integers representing the timing parameters and the generating clock's period, T_{high}, in some base unit of time (e.g., picoseconds, nanoseconds, etc.). The fifth assumes that clock multiplication is used, and asks the user for T_M instead of T_{high}, where T_M is assumed to be the generating clock's period (i.e., we assume that Clk_n is generated from Clk_m so that T_M remains fixed while T_N varies). These two ways of expressing the timing parameters are useful in situations where the same hardware is to be used at different frequencies. In programming the ROM used in the demonstration circuit of section 6.3, for example, the timing parameters were specified using absolute units with clock division.

6.1.2 Scheduling Algorithms

Four scheduling algorithms were implemented in the program: the original greedy algorithm for single-buffering, the greedy and lazy versions of the run-time scheduling algorithm for double-buffering, and the generalized run-time scheduling algorithm. These algorithms were slightly modified to make them work better with the software's representation scheme for time. More details regarding these modifications can be found in the discussion and source code contained in Appendix B.

6.1.3 Computation Modes

The scheduling software works in one of two computation modes: pair, and table. In pair mode, the software simply uses the currently selected algorithm to generate bidirectional schedules (i.e., two schedules, one for each direction) for the frequency ratio specified by the user through the variables M and N. In table mode, the software computes schedules for all frequency ratios with numerators and denominators ranging from 1 to $\max(M, N)$, and outputs the throughput efficiency in percent for each of these ratios in a two-dimensional table format. Table mode is also used to generate object code files for programming ROMs that work with different frequency ratios.

6.1.4 Output Formats and Options

The scheduling software can produce output in a number of formats, and has a few options available for each of these formats.

Text Output

The simplest output format is text, where the program just emits text to the standard output. This provides a quick and portable (i.e., it works on standard TTY terminals) way to look at the generated output. When using this format, the program also allows the user to keep a log of all generated text output in a separate file. This makes it possible for the user to do several computations in one run through the program, and then review the results at a later time by looking at the log file.

When using text format in pair computation mode, the program shows the schedules for both directions by listing the values of the control signals for each clock cycle, as shown in Figure 6-2. The user has the option of displaying these control signals in a simple character string format as in Figure 6-2(a), or in a formatted text version of a schedule diagram as in Figure 6-2(b). The latter gives a better sense of the timing relationship between the two systems, and is preferable to the former as long as M and N are not so large that the text schedule diagrams become wider than the screen. The program also reports the throughput efficiency in each of the two directions, although this is not shown in Figure 6-2.

Since we cannot draw arrows to explicitly indicate data transfers, we instead show the values of two data lines in the circuit, TD (transmitted data) and RQ (received data). The value of TD during a certain cycle is the value presented by the transmitter at the input of the transmit register during that cycle. If TE is also 1 during that cycle, then TD gets latched into the transmit register at the *next* clock edge, and gets transmitted to the receiver. The value of RQ during a certain cycle is the value at the output of the receive

```
M -> N
-------
N.RE   : 110101
M.TE   : 11011
-------
N.Rsel : 00x0x0
M.Tsel : 00x00
-------
N.RQ   : 301x2x
M.TD   : 12x30

N -> M
-------
N.TE   : 010111
M.RE   : 10111
-------
N.Tsel : x0x000
M.Rsel : 0x000
-------
N.TD   : x1x230
M.RQ   : 30x12
-------
```

(a)

```
M -> N
-------
N.RE   : |  1 |  1 |  0 |  1 |  0 |  1 |
M.TE   : |  1   |  1   |  0   |  1   |  1   |
-------
N.Rsel : |  0 |  0 |  x |  0 |  x |  0 |
M.Tsel : |  0   |  0   |  x   |  0   |  0   |
-------
N.RQ   : |  3 |  0 |  1 |  x |  2 |  x |
M.TD   : |  1   |  2   |  x   |  3   |  0   |

N -> M
-------
N.TE   : |  0 |  1 |  0 |  1 |  1 |  1 |
M.RE   : |  1   |  0   |  1   |  1   |  1   |
-------
N.Tsel : |  x |  0 |  x |  0 |  0 |  0 |
M.Rsel : |  0   |  x   |  0   |  0   |  0   |
-------
N.TD   : |  x |  1 |  x |  2 |  3 |  0 |
M.RQ   : |  3   |  0   |  x   |  1   |  2   |
-------
```

(b)

Figure 6-2: Sample text outputs in pair mode. (a) Without text schedule diagrams. (b) With text schedule diagrams.

Freq.	1	2	3	4	5	6
1	0	0	0	100	100	100
2	0	50	50	100	100	100
3	0	50	33	66	66	100
4	100	100	66	100	75	100
5	100	100	66	75	100	80
6	100	100	100	100	80	100

Figure 6-3: Sample text output in table mode.

register. If RE was 1 during the *previous* cycle, then RQ represents new data just latched into the transmit register at the most recent clock edge. As with determining RE and TE, we use modulo arithmetic when determining which clock cycle receives the transmitted data. That is, the data transfers "wrap-around" the right edge of the diagram.

When using text format in table computation mode, the program generates a simple table of throughput efficiency values with M designating the rows and N designating the columns. The throughput efficiency reported is the throughput efficiency when transmitting data from system m to system n. Figure 6-3 shows an example where the timing parameters are defined as $S = 1$, $H = 1$, $C = 1$, and $P = 3$, in units of Δt.[1] Note that since units of Δt are being used here, the fraction of the clock cycle covered by a timing parameter actually becomes larger as M and N become smaller. That is why the throughput efficiency starts out low with small values of M and N and becomes higher as M and N increases. A similar effect can be seen when using absolute units with clock division.

In both pair or table computation modes, the user has the option of looking at the results of intermediate computations while the program is running. This is useful for debugging code, and also for just gaining a better understanding of how the algorithm works. When using this "peek mode", a log file can become useful since the intermediate computations usually generate a large amount of output.

LaTeX Output

The scheduling software also allows the user to generate files that can be formatted by the LaTeX document preparation system [29]. It can generate graphical schedule diagrams in pair computation mode, and formatted throughput efficiency tables in table computation mode.

In pair computation mode, the program generates a LaTeX file, *sched.tex*, which when compiled, produces schedule diagrams such as that in Figure 6-4.[2] Unfortunately, arrows

[1] These are the same parameters used in the original example in section 3.3, in the examples in Chapter 5, and in the example in Figure 6-2. These are *not* the same as the parameters used in Figures 3-5 and 4-1.

[2] The figure only shows one direction of data transfer. The actual *sched.tex* file contains a separate schedule diagram for each of the directions.

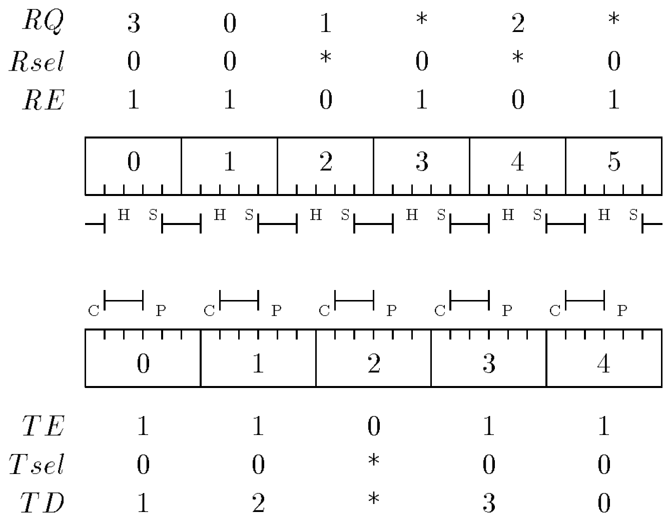

Figure 6-4: A LaTeX format schedule diagram.

cannot be drawn in the schedule diagram because LaTeX's `picture` environment, which is used to generate these diagrams, has no provision for drawing lines of arbitrary slope.

In table computation mode, the program generates a LaTeX `tabular` environment which can be inserted into LaTeX documents to produce tables such as those in Figures 3-5 and 4-1. Figure 6-5 shows another table generated using the timing parameters of the demonstration circuit discussed in section 6.3.

Since generating schedule diagrams in pair computation mode and generating text output can be done independently, the software does both. That is, it displays the results in text format on the standard output, and at the same time generates *sched.tex*. Similarly, when computing in table mode, the program uses text format for the standard output, while using LaTeX format for the output file. In this case, however, the output file is simply the log file, so the user may first have to clip the desired `tabular` environment from the log file and save it into a separate file before being able to use it.

One feature that was planned but not yet implemented is the ability to generate LaTeX format schedule plots such as the one in Figure 5-1. This feature can be implemented using the same techniques used to implement the LaTeX schedule diagrams are implemented.

Transmit	Receive Frequency															
Freq.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	0	50	50	100	100	100	100	100	100	100	100	100	100	100	100	100
3	100	50	100	66	100	100	100	100	100	100	100	100	100	100	100	100
4	100	100	66	100	75	100	100	100	100	100	100	100	100	100	100	100
5	100	100	100	75	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	100	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	100	87	100	88	100	100	100	100	100	100
10	100	100	100	100	100	100	100	100	88	100	90	100	100	100	100	100
11	100	100	100	100	100	100	100	100	100	90	100	90	100	100	100	100
12	100	100	100	100	100	100	100	100	100	100	90	100	91	100	100	100
13	100	100	100	100	100	100	100	100	100	100	100	91	100	92	100	100
14	100	100	100	100	100	100	100	100	100	100	100	100	92	100	92	100
15	100	100	100	100	100	100	100	100	100	100	100	100	100	92	100	93
16	100	100	100	100	100	100	100	100	100	100	100	100	100	100	93	100

Figure 6-5: LaTeX format efficiency table for the demonstration circuit's timing parameters.

ROM Output

The scheduling software provides the capability to generate output in the Intel MCS-86 16-bit Hexadecimal Object file record format [30]. To produce an object file for a "generic" ROM similar to that described in section 3.5.1, the user simply specifies table computation mode and ROM Output format together. This tells the program to output an object file containing not only the appropriate lookup table contents for different frequency ratios, but also those for different algorithms. In this way, the user can specify the scheduling algorithm using the most significant address bits of the ROM. The program also gives the user the option to generate a ROM file specifically for use with the prototype rational clocking chip. When this option is used, the program automatically remaps the ROM to account for certain idiosyncrasies of the rational clocking chip. More details on the address mapping of the ROM, as well as its output format, are provided in section 6.3.4.

When using ROM output format, the program writes the output to both the standard output and to the log file. The ROM format is a text file, not a binary file, so the user can easily edit the log file as necessary. Furthermore, the object file is generated in such a way that the tables produced by different algorithms are stored under separate *segments* [30]. This makes it easy for the user to select only desired algorithms when all of them together would not fit in the ROM.

6.1.5 Results and Recommendations

The communication scheduling software successfully generated correct schedules for different timing parameters, frequency ratios, and algorithms, and gave valuable insights that helped refine the scheduling algorithms. Furthermore, it was successfully used to generate a ROM for use with the prototype rational clocking chip in the demonstration circuit.

Despite its usefulness, however, the software can still benefit from several improvements. Improved graphical output, for example, would be very useful in helping users visualize the communication schedules. This can be achieved by using Postscript instead of the LaTeX `picture` environment. Automatic conversion between different timing parameter units would be another useful feature. Ideally, the user should just be able to enter numbers taken from a data book, and then have the software automatically perform such computations as finding appropriate values for d, and performing conservative rounding when necessary.

Perhaps the most useful feature to have at this point, however, would be the capability to check the correctness of the generated schedules. Right now, the program does not actually check that the schedule generated by an algorithm is error-free. Using the run-time double-buffered lazy algorithm for the slower receiver case, for example, frequently results in erroneous data transfers being done when the smaller of M and N is not even To detect these problems, we might write an independent function that takes a given schedule, attempts to perform data transfers according to it, and reports problems such as setup and hold time violations.

6.2 The Prototype Rational Clocking Chip

The prototype rational clocking chip is a full-custom CMOS VLSI circuit that implements the RAM-based scheduler described in section 3.5.3, and is designed to be flexible enough to be used in a number of experimental configurations. A block diagram of the prototype chip is shown in Figure 6-6. The chip contains three major components: the counters, the lookup tables, and the circuitry for loading the tables on bootup. This section gives a

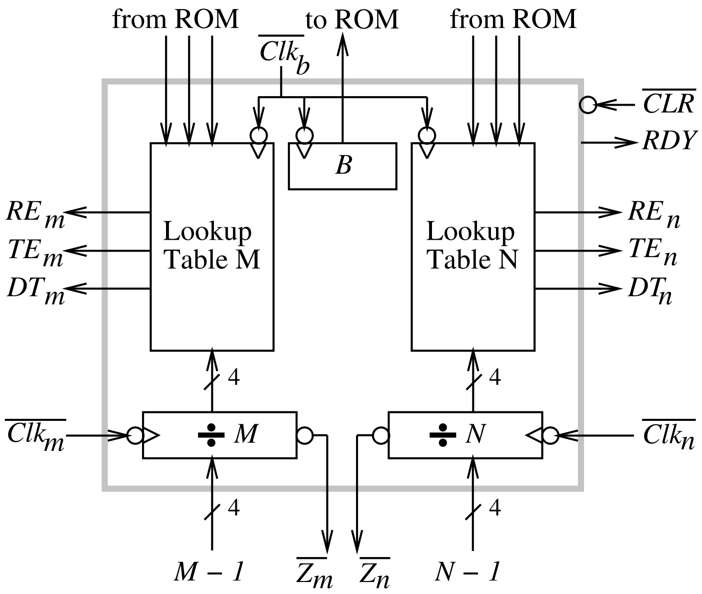

Figure 6-6: The prototype rational clocking chip.

functional description of these components, and presents the results of the fabrication and testing process. A layout of the chip can be found in Appendix C.

6.2.1 The Counters

Each counter in the chip is a 4-bit negative edge-triggered decrementing counter with a terminal count input, and a combinational active-low zero output, $\overline{Z}$, which goes low whenever the count value is zero. To use it as a divide-by-N counter, we set its terminal count to $N-1$. This makes the counter count down from $N-1$ to zero, and causes it to emit a one cycle pulse on $\overline{Z}$ once every N cycles. We can then implement clock multiplication by connecting the counters to an external PLL as in Figure 3-2. We can also use the chip with clock division by simply taking the clock inputs from an external clock division circuit like that in Figure 3-1, and ignoring the $\overline{Z}$ outputs. The external clock division circuit in turn can be implemented using another rational clocking chip.

The counters can be used to generate rationally related clocks with ratios where M and N range from 2 to 16. They cannot be used for ratios where either M or N is 1, however, since the $\overline{Z}$ output would then stay low and not pulse. This problem can be solved easily by either manually connecting the undivided clock directly to the appropriate system, or doubling both M and N.

6.2.2 The Lookup Tables

Within the chip, each counter is used to index a 16-entry, 3-bit wide lookup table. The tables are only 3 bits wide instead of the ideal 4 bits for double buffering (i.e., one bit each for TE, $TSel$, RE, and $RSel$) because the chip was designed before double-buffering was developed. Thus, it only has bits for TE and RE, plus an extra bit for a time-advanced copy of TE should reaction latency compensation be necessary. Fortunately, for experimentation purposes, we can easily get 4 bits by using two copies of the rational clocking chip.

The lookup tables are implemented as serial-in parallel-out (SIPO) shift registers that are loaded at bootup time with the appropriate entries by shifting-in values from a 3-bit input. With the appropriate interface, the lookup tables can be used with a serial ROM, a combinational ROM, a processor running scheduling software, or a run-time scheduling circuit.

6.2.3 The Bootup Control Circuitry

The chip has an asynchronous active-low reset line, $\overline{CLR}$, which clears the tables and counters, and signals the start of the bootup sequence. The chip waits until $\overline{CLR}$ goes high, then proceeds to load the lookup tables on the following 16 clock edges, shifting-in entry 15 first and entry 0 last. The lookup table outputs are invalid during these first 16 cycles, and should not be used. A ready signal, RDY, goes high after the last entry has been loaded to indicate when we can start using the lookup tables.

The lookup tables and the bootup control circuitry use a "bootup" clock, Clk_b, which is independent of the two system clocks. This separate clock allows us to load both systems' lookup tables from a single ROM. It also allows us to load the lookup tables at a slower rate, as may be required if we have a slow ROM, slow run-time scheduling hardware, or are using software to load the lookup table.

Using an independent bootup clock has the disadvantage of making the RDY signal asynchronous to the two system clocks. This problem is easily solved by ignoring RDY altogether, and instead manually determining some minimum number of equivalent system clocks it takes to load the tables and making sure that the systems wait at least that long before attempting to transfer data. Alternatively, a better solution is to use the $\overline{Z}$ output of one of the counters as the bootup clock. This clock would run at the coincidence frequency, and would be synchronous to both systems. We can also use a frequency-divider on the $\overline{Z}$ output if a slower bootup clock is desired.

To make the chip easy to use with conventional, combinational ROM chips, an on-chip bootup counter, B, is provided. This counter, which uses the bootup clock, emits a decreasing 4-bit number that can be used to index a combinational ROM, allowing the ROM to output the appropriate entries as the lookup table is loaded. Ideally, the bootup counter should count from 15 to 0 during the first 16 clock cycles so that the lookup table entries are loaded in the proper sequence. However, since the counter is cleared on reset[3] it actually starts at 0 first before going to 15 and counting down, causing a shift in the lookup table entry sequence. That is, entry 15 of the lookup table would be loaded with entry 0 from the ROM, entry 14 loaded with entry 15 from the ROM, and so on. This shift is easily accounted for, however, by having the software that programs the ROM store the intended contents for lookup table entry x in ROM entry $(x + 1) \bmod 16$ (for some x).

6.2.4 Fabricating and Testing the Chip

The prototype chip was designed using Mentor Graphics' GDT CAD tools. After extensive simulation, the chip design was sent to MOSIS for fabrication and packaging. Approximately two months later, MOSIS sent back four copies of the chip, each packaged in a 40-pin DIP. These chips were then tested on a breadboard under two configurations.

The first test configuration was simple: the three clocks were connected together to a pulse generator, and the lookup table inputs were connected to the bootup counter outputs. In this way, the lookup tables and counters could be tested without the need for other chips.

[3] The bootup counter and the index counters are built from the same design.

	Measured (ns)		Simulated (ns)	
	rising	falling	rising	falling
RE_m	28	28	16	21
$\overline{Z_m}$	24	35	12	20
bit 0 of $count_B$	22	22	10	10

Figure 6-7: Measured and simulated clock-to-Q propagation delays of some chip outputs.

	Measured from	Measured (ns)	Simulated (ns)
Rise time	0 V to 3.2 V	4	3
	0.5 V to 4.5 V	4	1.7
	0 V to 5 V	15	4
Fall time	5 V to 1.8 V	3	3.7
	4.5 V to 0.5 V	3.5	3.7
	5 V to 0 V	5	4.3

Figure 6-8: Measured and simulated rise and fall times of the chip's counter outputs.

A second test configuration involved two copies of the chip, with the first copy generating the two system clocks of the second copy through clock division. In this way, any unwanted glitches on the $\overline{Z}$ output of the first chip would be detected, and the ability of the second chip to take three independent clocks would be tested.

Of the four chips, one was damaged by a human error during testing, while the rest worked as expected. Timing measurements were taken using one of the chips, and were compared with the simulation results. As shown in Figure 6-7, the measured propagation delays were generally higher than their simulated values, but are roughly within a factor of two of their simulated values. This is not abnormal for prototype chips fabricated by MOSIS. The measured and simulated rise and fall times are shown in Figure 6-8. Comparing these values, however, would probably not be meaningful since the simulated times were taken from *inside* the chip (i.e., without including the pin driving circuitry), and thus does not necessarily represent the rise and fall times outside the chip.

The chip was able to operate successfully in the lab at clock speeds up to 25 MHz. It could not be tested above this frequency because the available pulse generator could not go faster. From the timing measurements in Figure 6-7, however, it seems unlikely that the chip's actual maximum frequency would be significantly higher than 25 MHz. Since the chip uses dynamic memory elements, it was also tested at low frequencies. It operated

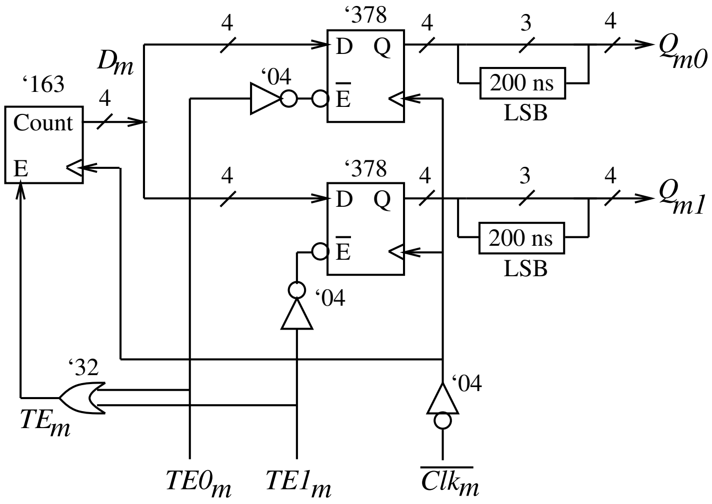

Figure 6-9: The transmitter's circuitry.

successfully at 333 Hz, the pulse generator's lower limit.

6.3 The Demonstration Circuit

A demonstration circuit using the prototype chip was built to test the rational clocking technique and demonstrate its usefulness. This circuit was built on a breadboard and uses off-the-shelf digital components from the 74Fxx family. It employs clock division to generate the system clocks, and implements a unidirectional 4-bit double-buffered data channel from system M to system N. The lookup tables are loaded from a single ROM large enough to allow the user to choose among different tables generated by several algorithms. This section describes the various parts of the demonstration circuit as well as the results of the experiments done on it.

6.3.1 The Transmitter's Circuitry

In the demonstration circuit, system M transmits to system N using a double-buffered data channel. Figure 6-9 shows the transmitter's circuitry. As shown, two 4-bit transmit registers (74F378) are used as in the double-buffered hardware shown in Figure 4-8. Instead of using $TSel$ and TE, however, we use two separate enable signals, $TE0$ and $TE1$, defined as follows:

$$\begin{aligned} TE0 &= TE \cdot \overline{TSel} \\ TE1 &= TE \cdot TSel \end{aligned} \tag{6.3}$$

By programming the lookup tables to generate these signals instead of $TSel$ and TE, we avoid the need for a demultiplexer, and save on hardware and wiring.

For a data source, we use a 4-bit counter (74F163) with its enable control connected to the TE signal derived by ORing $TE0$ and $TE1$. Configured in this way, the counter uses TE as the $D\ Taken$ signal. That is, whenever TE is 1, the counter knows that its current value will be latched by one of the transmit registers on the coming clock edge, and that it is thus safe to place the next value on D_m during the next cycle. Thus, in this case, reaction latency compensation is unnecessary.

Using the manufacturer's specifications for the different components, and making the necessary adjustments to account for the delay through the clock's inverter, we arrive at the following propagation and contamination delays for the two transmit registers (see Appendix A for more details):

$$\begin{aligned} t_{cm} &= 4.5\text{ns} \\ t_{pm} &= 15.5\text{ns} \end{aligned} \tag{6.4}$$

To make decision window violations more detectable, we connect the LSB of the transmit register's 4-bit output to a 200 ns delay element. This widens the transition window by increasing t_{pm} while keeping t_{cm} the same. That is, we now have:

$$\begin{aligned} t_{cm} &= 4.5\text{ns} \\ t_{pm} &= 215.5\text{ns} \end{aligned} \tag{6.5}$$

Furthermore, this scheme makes it possible to simulate the effect of metastable states, which are very hard to catch in practice. By delaying the LSB, we cause an out-of-sequence number to appear at the receiver's input during the transition window. Thus, if the receiver tries to latch its inputs any time during the transition window, it would get a wrong value — even if no metastable states are actually generated.

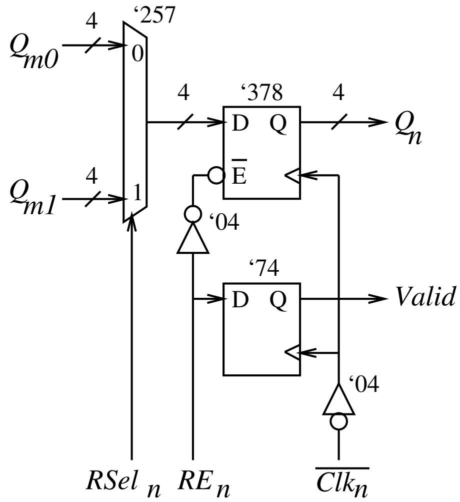

Figure 6-10: The receiver's circuitry.

6.3.2 The Receiver's Circuitry

The receiver's circuitry, shown in Figure 6-10, is simply a direct implementation of the receiver's circuitry in Figure 4-8. An additional register (74F74) is used to latch RE to produce a "valid" bit that tells the receiving system whether the current value of Q_n represents new data, or if it is old and should be ignored.

To calculate the equivalent setup and hold times of the receiver, we must take account not only of the delay through the clock inverter, but also of the additional delay through the multiplexer. Using the information in Appendix A, we arrive at the following setup and hold times:

$$\begin{aligned} t_{sn} &= 9.5\text{ns} \\ t_{hn} &= 4.0\text{ns} \end{aligned} \tag{6.6}$$

6.3.3 Clock Generation and Reset Circuitry

For simplicity, the demonstration circuit uses clock division instead of clock multiplication. This allows us to avoid using a PLL and having to deal with its analog requirements. As shown in Figure 6-11, we implement clock division by taking both system clocks of a rational clocking chip, RCC A, from a single clock, $\overline{Clk_{high}}$, and using the $\overline{Z_m}$ and $\overline{Z_n}$ outputs of

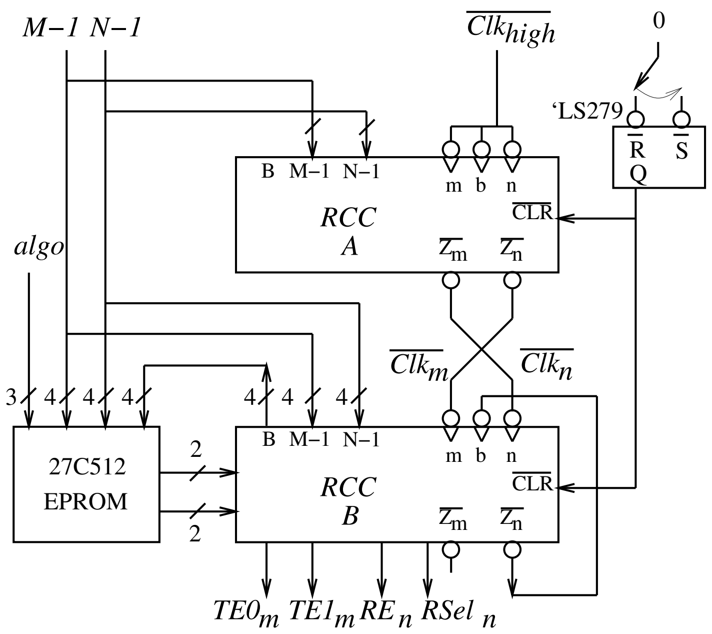

Figure 6-11: The clocking and communication control circuitry.

this chip as the $\overline{Clk_n}$ and $\overline{Clk_m}$ inputs respectively of a second rational clocking chip, RCC B.[4] In the lab, $\overline{Clk_{high}}$ was taken from a pulse generator running at 10 MHz. The timing relationships of the resulting clock waveforms are shown in Figure 6-12. For clarity, the counter values that determine each $\overline{Z}$ output are also shown. Note that by clearing all the counters simultaneously, the asynchronous $\overline{CLR}$ input makes the active edges of the two clocks, $\overline{Clk_m}$ and $\overline{Clk_n}$, coincide naturally when clock division is implemented.

A reset signal for the two rational clocking chips is generated by a TTL R-S flip-flop (74LS279) acting as a debounced switch [31]. To reset the circuit, the user first grounds the $\overline{R}$ input, then disconnects it and grounds the $\overline{S}$ input. This has the effect of first clearing, then setting the flip-flop without generating glitches due to switching mechanics. A TTL chip is required here so as to guarantee that an unconnected input is read as a logical 1. This prevents both $\overline{R}$ and $\overline{S}$ from being 0 at the same time.

[4]In the circuit diagram, a bar over a signal name indicates that the signal is active low. For clock signals, this means that the negative edge is considered to be the active edge.

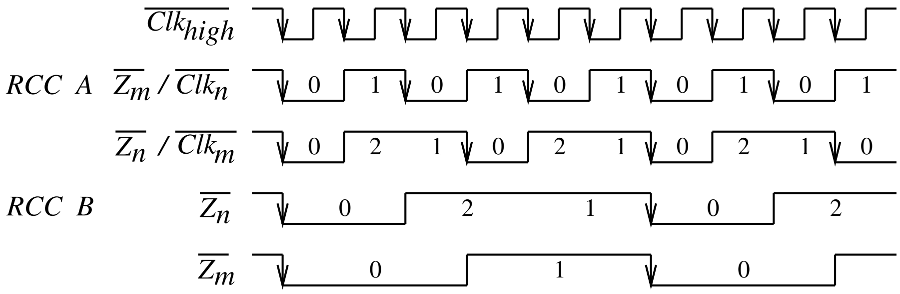

Figure 6-12: The demonstration circuit's clock waveforms.

6.3.4 The Communication Control Circuitry and the ROM

A second copy of the rational clocking chip, RCC B, is used to generate the control signals. As shown in Figure 6-11, RCC B generates the control signals needed to allow system M to transmit to system N. To implement bidirectional transfer, we can simply use a third copy of the chip to generate the other set of control signals.

RCC B is loaded from a combinational ROM (27C512) which is programmed by the communication scheduling software. It takes 15 bits of address input and outputs 8 bits, as shown in Figure 6-13. The lowest-order 4 bits of the address are taken from RCC B's bootup counter, and indicate the current clock cycle whose corresponding control signals are to be read. The next 8 bits specify M and N. The highest order 3 bits allow the user to switch between tables generated by different algorithms. At least one of the 8 possible settings corresponds to having no flow control at all — i.e., TE and RE are fixed at 1, and $TSel$ and $RSel$ are fixed at 0. This allows us to demonstrate clearly the need for flow control. Figure 6-14 shows the bit encoding actually used in the demonstration circuit.

Programming the ROM for use with the rational clocking chips requires some remapping of the lookup tables. First, we must account for the fact that the chip counters decrement instead of increment, as we have assumed so far in this paper and in the communication scheduling software. Then, we must fix the shift caused by the chip's bootup counter. Figure 6-15 shows the relationship between the cycle number ($cycle_m$), the counter value for the cycle $count_m$, and the ROM address (i) where the control signals for this cycle should be stored. From this table, we can determine what goes into some ROM address i

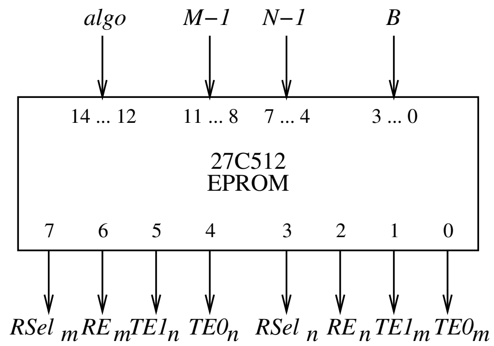

Figure 6-13: The ROM inputs and outputs.

Code	Algorithm
000	No Flow Control
001	Original Single-Buffered
010	Generalized Run-Time
011	Greedy Double-Buffered
100	Lazy Double-Buffered
other	undefined

Figure 6-14: Bit codes for different algorithms.

$cycle_m$	$count_m$	i
0	0	1
1	$M-1$	0
2	$M-2$	$M-1$
3	$M-3$	$M-2$
...	...	...

Figure 6-15: Mapping between cycle number and ROM address (for system m).

as follows:

$$ROM[i] \leftarrow Table[(M - i + 1) \bmod M] \tag{6.7}$$

where $Table[x]$ for some x represents lookup table entry x if we are using an incrementing counter.

6.3.5 Results and Recommendations

The demonstration circuit was tested using different frequency ratios and scheduling algorithms. It performed as expected, and provided a better feel for how rational clocking solves the flow control and synchronization problem. It was especially interesting to see how double-buffering allowed 100% throughput to be achieved where it could not be done with single-buffering. The waveforms produced by this circuit were very similar to those produced by the Verilog simulations shown in the next section.

In order to provide a more realistic demonstration of how rational clocking would actually be implemented in applications, the demonstration circuit should be modified in the future to use clock multiplication instead of clock division. Among other things, this would allow the user to test the robustness of the rational clocking circuitry in the presence of phase jitter and other possible problems involving the PLL.

6.4 Verilog Modelling and Simulation

The prototype chip and the demonstration circuit tested the rational clocking technique with a simple table-based implementation. To test the more complex run-time implementation, a model using the Verilog hardware description language [32] was constructed and simulated using Cadence Systems' Verilog tools. This model included a bidirectional double-buffered data channel, three versions of the run-time scheduling hardware, and circuitry for computing constants such as e_0 and y_0 at bootup. The complete source code is listed in Appendix D.

6.4.1 The Data Channel

The Verilog model contains two double-buffered data channels (one for each direction) implementing the original double-buffered configuration shown in Figure 4-8. The transmit registers in these channels, unlike those in the demonstration circuit, are controlled by TE and $TSel$, instead of $TE0$ and $TE1$. As with the demonstration circuit, counters with enable inputs are used as data sources, and an extra 1-bit register is used by the receiver for the valid bit.

This model does not use a delay element to increase propagation delay as done in the demonstration circuit, but instead uses Verilog features to simulate contamination and propagation delays in the transmit registers, and setup and hold times in the receive registers. These timing parameters can be specified by the user in units of Δt.

6.4.2 The Scheduling Hardware

The Verilog model implements three different versions of the scheduling hardware: the lazy and greedy basic run-time schedulers of section 5.2, and the generalized run-time scheduler of section 5.3. Conditional compilation directives allow the user to select between these versions by simply defining the appropriate names using the `` `define `` compiler directive.

6.4.3 The Bootup Computation Circuitry

In order to work properly, the scheduling hardware requires precomputed values of certain constants such as e_0 and y_0. In the Verilog model, we compute most of these constants at bootup time using the circuits described in section 5.2.4.

The model uses both combinational and sequential versions of the modulo circuit. The combinational version is used for computing e_0, y_0, and the value of $(i-1) \bmod M$ in the *LD Gen* module. The sequential version is used for computing ΔNM, and $e_0 \bmod M$.

Each of the three scheduling hardware implementations uses its own version of the bootup computation circuitry. Some versions use additional circuitry to perform necessary

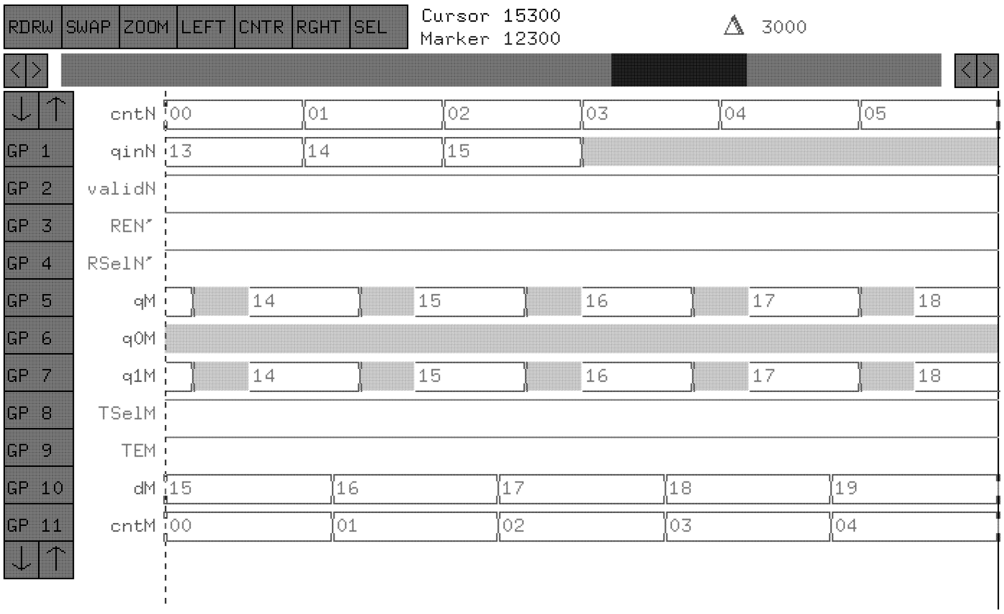

Figure 6-16: Slower transmitter case with no flow control.

adjustments to e_0 and y_0 as discussed in Chapter 5. The generalized run-time scheduler also uses extra circuitry for computing e_{max} and e_{min}.

6.4.4 Results and Recommendations

The three versions of the scheduling hardware were each simulated using several sets of timing parameters and frequency ratios, and the resulting schedules were compared with the schedules generated by the communication scheduling software. All the versions worked as expected, thus proving the correctness of the various run-time scheduling circuits.

Figures 6-16 to 6-19 show sample outputs from the Verilog simulator for the case where $M = 5$, and $N = 6$, and the timing parameters are $S = 1$, $H = 1$, $C = 1$, and $P = 3$, for both systems. In each of these figures, $cntM$ and $cntN$ indicate the index counters' outputs, d represents data at the input of the transmit register, $q0$ and $q1$ are the outputs of the two transmit registers, q is the output of the multiplexer, and qin is the output of the receive register. (Note each of these signal names are followed by N or M in the figure, depending on the system they belong to.) Shaded areas represent unknown values, and occur during the transition windows of the transmit registers and when the receive register suffers a decision window violation.

Figure 6-17: Slower receiver case with no flow control.

Figures 6-16 and 6-17 show the result of not using any flow control at all. It is clear from these figures that we not only suffer data loss as the faster system outruns the slower one, we also encounter a lot of decision window violations. Figures 6-18 and 6-19 show the result of using the generalized run-time scheduler with double-buffering. As shown, by emitting the right control signals at the right times, we successfully solve the flow control and synchronization problems. Furthermore, by using double-buffering, we are able to achieve 100% throughput where it would otherwise not have been possible.

There are still many ways to improve the Verilog model. It may be useful, for example, to modify the generalized run-time scheduler to work with timing parameters that are not integer multiples of Δt. This can be done by making the appropriate registers and multiplexers wider. Most efforts at improving this Verilog model, however, should be directed towards developing and simulating appropriate interfaces for real applications. It would particularly be useful, for example, to start integrating the Verilog model built for this thesis, with the Verilog model of the current NuMesh CFSM.

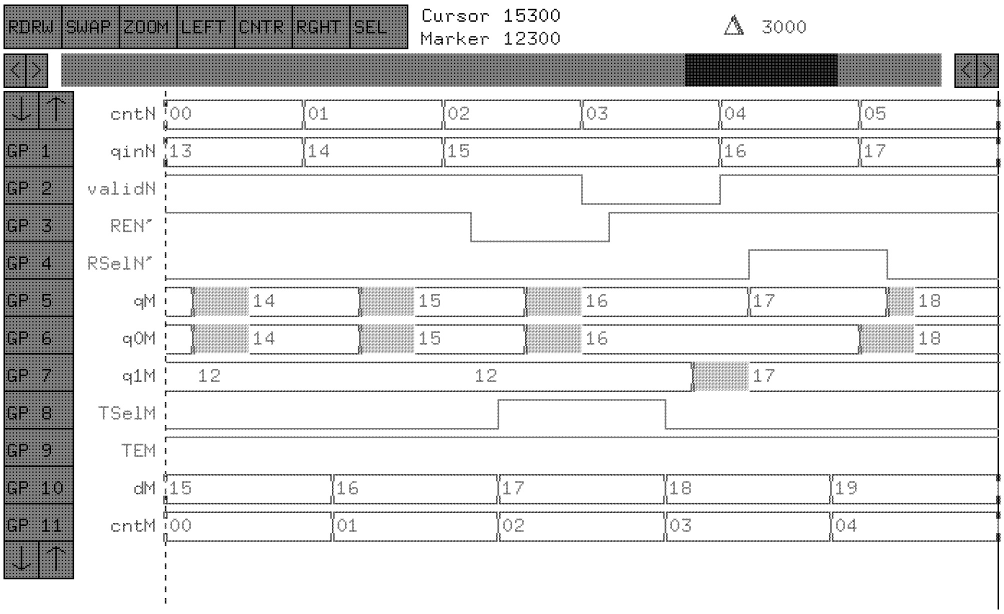

Figure 6-18: Slower transmitter case with generalized run-time scheduling and double-buffering.

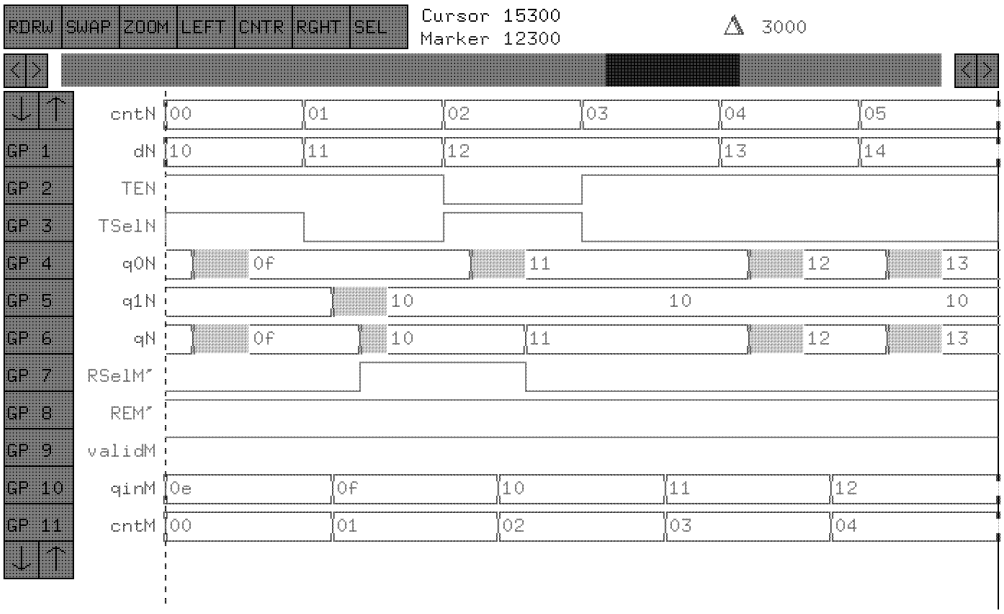

Figure 6-19: Slower receiver case with generalized run-time scheduling and double-buffering.

Chapter 7

Conclusion

Rational clocking addresses the increasingly common problem of providing efficient communication between two systems running at different clock speeds. Traditional solutions to this problem have generally lacked either efficiency or flexibility, and leave much to be desired. Rational clocking provides both efficiency and flexibility by requiring a rational relationship between the two clock frequencies. This rational frequency constraint makes it possible to achieve communication without handshaking and synchronizers for a wide variety of frequency pairs.

The most straightforward way to implement rational clocking is to use lookup tables that generate flow control signals according to a precomputed communication schedule. This schedule is determined given the frequency ratio and the two systems' timing parameters using a simple greedy algorithm which selectively enables and disables data transfers in such a way that timing violations are avoided. The table-based hardware can be made more versatile by using the algorithm on several different frequency ratios and then storing the resulting schedules into a single generic table.

Although the straightforward implementation of rational clocking provides high levels of throughput efficiency for different frequency ratios given typical timing constraints, it cannot maintain this high efficiency given the tighter timing constraints typical in high-performance systems. The double-buffering technique, which uses two transmit registers instead of one, solves this problem by providing a data channel where the possibility of contamination between successive data transfers is eliminated. This contamination-free data channel simplifies the scheduling requirements, and can be proven to guarantee 100% throughput for the slower system in most applications, regardless of frequency ratio and timing parameter values.

The table-based implementation has the disadvantage of trading-off space for flexibility. A generic table accommodating different frequency ratios and timing parameters, for example, requires as much as $O(N^7)$ bits of memory, where N is the maximum possible value for the frequency ratio numerator and denominator. This problem can be solved by avoiding precomputation altogether and taking advantage of the clocks' regularity to schedule communications at run-time using specialized rate multipliers. This run-time scheduling hardware only requires $O(\lg N)$ bits to handle all possible ratios and timing parameters. Two major variants of run-time scheduling have been presented. The basic algorithm works with contamination-free data channels, and is easy to implement. The general algorithm extends the basic algorithm to cases where contamination is possible, and makes it possible to design run-time scheduling hardware that can be used on single-buffered systems, but take advantage of double-buffering when available.

All these techniques were studied and tested using a combination of software, hardware, and simulations. Communication scheduling software was written for testing and using the different scheduling algorithms. A VLSI prototype chip employing programmable lookup tables was fabricated and used in a circuit that demonstrates double-buffering. The different run-time scheduling circuits were modelled and simulated in Verilog. These different tools successfully demonstrated the correctness and usefulness of rational clocking, and provide a base for future research in applying rational clocking to real systems.

7.1 Putting It All Together

The ideas and techniques presented in this thesis each have their own advantages and disadvantages, and can be used together when applying rational clocking on high-performance and high-flexibility systems. Figure 7-1 shows a single circuit that integrates these different techniques to achieve efficiency, flexibility, and cost-effectiveness.

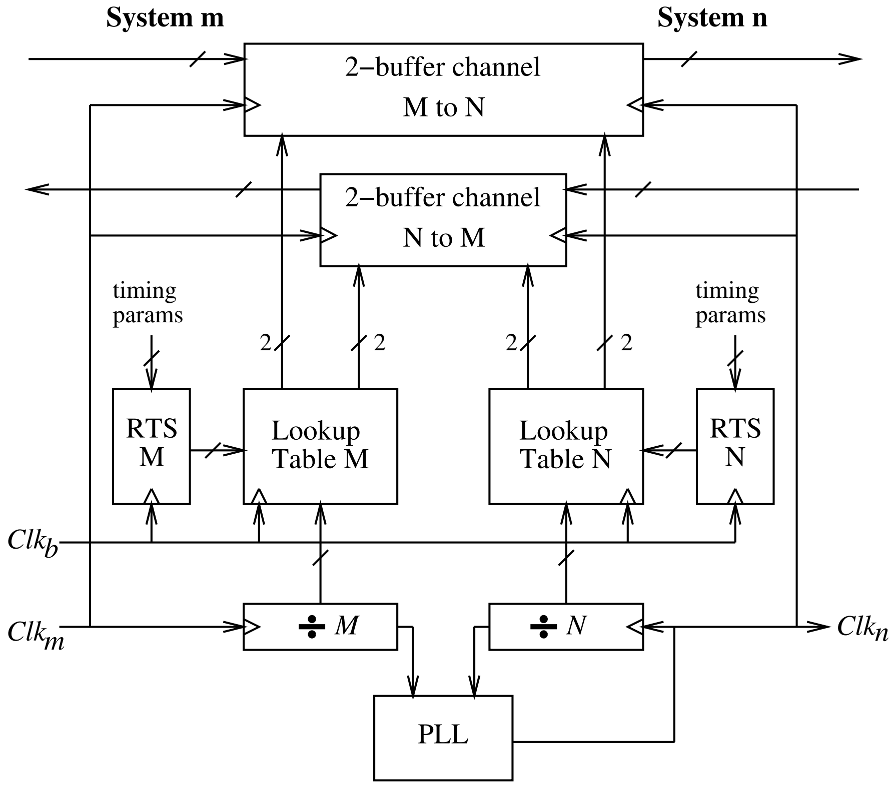

Figure 7-1: Putting it all together in a single circuit.

This circuit provides 100% throughput efficiency for all frequency ratios and timing parameters by using double-buffered data channels. The use of double-buffering also simplifies communication scheduling, and in particular, allows us to use the basic run-time scheduling algorithm, which is easier to implement than the general algorithm and less sensitive to variations in the timing parameter values. Run-time scheduling hardware allows maximum flexibility in choosing frequency ratios and timing parameters without taking-up a large amount of space. It can be used alone to generate control signals continuously, but in very high-performance systems where the computation delay through the run-time scheduler limits the maximum clock speed, is better used together with programmable tables as shown in the figure. These tables would not be unreasonably large or expensive — they need only around $4N$ bits each, where N is typically less than 16 — and thus can be made extremely fast. By loading these tables from the run-time scheduling hardware using a slow bootup clock, we can combine the flexibility of the run-time scheduler with the speed of the small lookup tables.

7.2 Future Research

There is still a significant amount of research, both theoretical and practical, that can be done on various aspects of the rational clocking technique.

On the theoretical side, the properties of the single-buffered algorithms — the original greedy algorithm of section 3.3 and the generalized algorithm of section 5.3 — can be studied further. In particular, it would be useful to determine the exact conditions that lead to sub-optimal throughput efficiency given a certain choice for P_{begin}, and to develop a way to choose the optimal P_{begin} without having to run through all the possibilities. It may be possible to solve this problem by unrolling the loop in the run-time scheduling algorithms and deriving an analytic expression of the control signals as a function of M, N, the timing parameters, and the current clock cycle number. Another good subject for future research is the application of double-buffering to out-of-phase communication. As discussed in section 4.4.2, this can be valuable in multiprocessor systems and should be studied further.

On the practical side, research can be directed towards applying rational clocking in real multiprocessor and uniprocessor systems. This would involve improving the scheduling software, performing more hardware experiments, and developing interfaces. It would also be useful to develop a standard "off-the-shelf" rational clocking chip that would allow system designers to easily incorporate rational clocking into their systems. This circuit would use run-time scheduling, either alone or with programmable lookup tables, to provide flexibility without requiring a large amount of space.

7.3 Concluding Remarks

In this thesis, we presented the rational clocking technique and improved it significantly, making its application in real systems not only more promising, but more feasible as well. More research should now be directed towards actually implementing rational clocking, and making it a standard part of existing and future high-performance computer systems.