Appendix A

Computing Timing Parameters

In computing the timing parameters of a system, it is not enough to consider only the parameters of the transmit and receive registers. The inputs, outputs, and clocks of these registers may suffer additional delays that can change their effective timing parameters. This appendix discusses general rules for handling these delays, and then applies these rules to the computation of the demonstration circuit's timing parameters.

A.1 Effective Timing Parameters

Figure A-1 shows three possible ways in which additional delays can affect a register's timing parameters. In general, delays can occur at a register's data output, data input, or clock input. As shown in the figure, we can model these delays with a buffer having contamination and propagation delays t_{cB} and t_{pB}. Taking the register and the buffer together as a single module, we can then use t_{cB}, t_{pB}, and the register's timing parameters, t_s, t_h, t_c, and t_p,

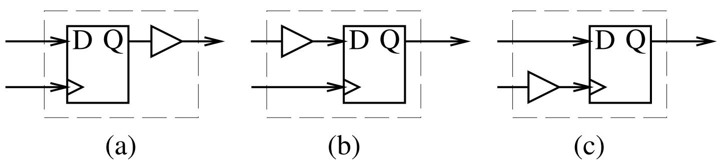

Figure A-1: Possible delay configurations. (a) Output. (b) Input. (c) Clock.

to compute the module's *effective* timing parameters, t'_s, t'_h, t'_c, and t'_p, in different ways depending on the delay configuration.

A.1.1 Delayed Output

The simplest case is that of a delay connected to the register's output, shown in Figure A-1(a), where the delays of the buffer simply add to the output delays of the register, giving the following effective timing parameters:

$$t'_s = t_s, \quad t'_h = t_h \tag{A.1}$$

$$t'_c = t_c + t_{cB}, \quad t'_p = t_p + t_{pB} \tag{A.2}$$

A.1.2 Delayed Input

The next case is that of a delayed input, shown in Figure A-1(b). Here, the delays of the buffer affect the input timing constraints of the module enclosed in the dashed box. The propagation delay through the buffer requires us to present data at the module's input t_{pB} before the setup time, while the contamination delay allows us to change the module's input t_{cB} before the hold time without violating the register's timing constraints. This leads to the following effective timing parameters:

$$t'_s = t_s + t_{pB}, \quad t'_h = t_h - t_{cB} \tag{A.3}$$

$$t'_c = t_c, \quad t'_p = t_p \tag{A.4}$$

A.1.3 Delayed Clock

In the last case of a delayed clock, shown in Figure A-1(c), both the input timing constraints and the output delays are affected. The propagation delay through the buffer requires us to hold the register's inputs stable t_{pB} after the register's hold time, while the contamination delay allows us to delay the input by an extra t_{cB} without violating the setup time. These

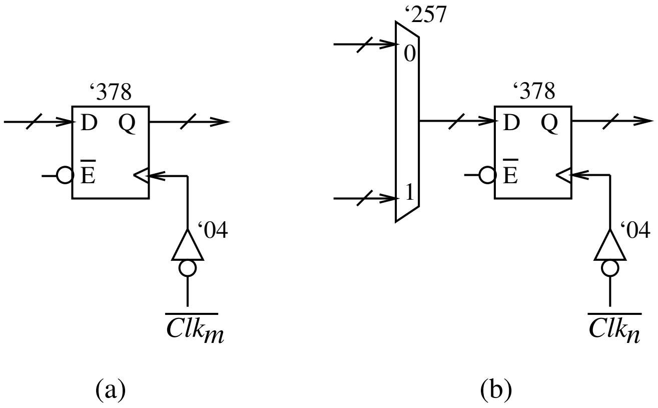

Figure A-2: The demonstration circuit's delays. (a) Transmit register. (b) Receive register.

delays also cause latching to be delayed, and thereby add to the output delays of the register. Thus, we have the following effective timing parameters:

$$t'_s = t_s - t_{cB}, \quad t'_h = t_h + t_{pB} \tag{A.5}$$

$$t'_c = t_c + t_{cB}, \quad t'_p = t_p + t_{pB} \tag{A.6}$$

A.1.4 Applications

Delays at the outputs usually represent combinational logic processing the register's output, as in the case of the transmit modules of Figure 2-1. Similarly, delays at the inputs represent logic processing the register's inputs, such as the receiver's multiplexer in the double-buffered hardware of Figure 4-8. Delays at the clock input can be due to clock skew, phase jitter, or both. In a typical system, these three delay configurations often occur together in a single module. In these cases, the effective timing parameters can be computed by successively applying the appropriate equations.

A.2 The Demonstration Circuit

The relevant circuitry for computing the effective timing parameters of the transmit and receive registers in the demonstration circuit are shown in Figure A-2. (These are taken

	74F378	74F04	74F257
Setup time of D input (ns)	4.0	N/A	N/A
Hold time of D input (ns)	0.0	N/A	N/A
Contamination delay (ns)	3.0	1.5	2.0
Propagation delay (ns)	9.5	6.0	7.0

Figure A-3: Worst-case timing parameters of circuit components.

from Figures 6-9 and 6-10.) The components in these circuits are all from the 74Fxx family, and have the worst-case timing parameters shown in Figure A-3 [33].

The transmit register has a delayed clock, so its effective contamination and propagation delays can be computed as follows using Equation A.6:

$$\begin{aligned} t'_c &= 3.0 \text{ ns } + 1.5 \text{ ns } = 4.5 \text{ ns} \\ t'_p &= 9.5 \text{ ns } + 6.0 \text{ ns } = 15.5 \text{ ns} \end{aligned} \tag{A.7}$$

The receive register has a delayed clock *and* a delayed input, so its effective parameters are computed using Equations A.6 and A.3 successively as follows:

$$\begin{aligned} t'_s &= (4.0 \text{ ns } - 1.5 \text{ ns }) + 7.0 \text{ ns } = 9.5 \text{ ns} \\ t'_h &= (0.0 \text{ ns } + 6.0 \text{ ns }) - 2.0 \text{ ns } = 4.0 \text{ ns} \end{aligned} \tag{A.8}$$

The other effective timing parameters can be computed in a similar manner, but only the ones computed here are relevant for determining communication schedules.

Appendix B

The Scheduling Software

This appendix presents the details of the scheduling software, starting with a discussion of the integer-based time representation scheme mentioned in Chapter 6, and some scheduling algorithm implementation details. A complete listing of the C source code and the LaTeXsupport files are then provided.

B.1 Representing Time with Integers

The original communication scheduler used for generating the data presented in [1] and [2], computed the throughput efficiency of the original algorithm in section 3.3 given M and N as integers and the timing parameters as floating-point numbers equal to the fraction of a clock cycle they represent (e.g., $P = 0.20$ would mean that the propagation edge occurs 20% of a clock cycle after the clock edge). This program was simple and worked well in most cases, but had an unfortunate limitation: it was prone to occasional floating-point rounding errors. These errors caused the program to prohibit data transfers unnecessarily, and resulted in reduced throughput efficiency. If we compare the throughput efficiency table shown in Figure 3-5 with the original one in [2], for example, we would see that even though the timing parameters used are the same, the table in [2] only claims 50% throughput for a transmit-receive ratio of 3/2, instead of the actual 100% throughput.

To avoid this problem, the software used in this thesis performs all computations using only integers. Time is expressed in integer units of dt, defined as:

$$dt = \Delta t / d \tag{B.1}$$

where $\Delta t = T_{MN}/MN$ as defined in Chapter 5, and d is some integer specified by the user. These units are useful because they allow the positions of the two systems' clocks to be represented by exact integers. That is, substituting the definitions of dt and Δt into the formula for the coincidence period, Equation 3.2, we get:

$$\begin{aligned} T_M &= N\Delta t = (N \cdot d) \cdot dt, \\ T_N &= M\Delta t = (M \cdot d) \cdot dt \end{aligned} \tag{B.2}$$

Furthermore, by making d large, we can express arbitrarily small units of time (as long as d does not get so large that the times we want to express do not fit in a 32-bit "long" integer).

Although the software performs all computations internally using units of dt, it allows the user to specify timing parameters in other units as well. As mentioned in section 6.1, the user has a choice of expressing timing parameters in units of dt, in units of Δt, as fractions of a clock cycle, or in absolute units. When expressing time in absolute units, the user has the further choice between using these units with clock division or clock multiplication. Converting timing parameters expressed in units of dt and Δt is straightforward, but converting parameters expressed as fractions or in absolute units requires additional analysis, and is discussed in the following sections.

B.1.1 Using Fractions

As discussed in section 3.5.1, it is useful to specify the timing parameters as fractions of a clock cycle if we presume that our target systems have timing parameters that scale with their minimum clock periods. If a timing parameter P of system m is equal to some fraction x of the system clock edge T_M, and we choose d such that $x = p/d$ for some integer p, then we can express P in units of dt as follows:

$$P = x \cdot T_M = \frac{p}{d} \cdot (N \cdot d) = p \cdot N \tag{B.3}$$

For example, if $M = 5$, $N = 6$, $d = 1000$, and $P = 0.20T_M$, then $p = 200$, and $P = 200 \cdot 6 = 1200$ time units. Note that this representation scheme limits us to using rational numbers for the timing parameters. This is not a problem, however, since we can make these parameters sufficiently close to their actual value by making d large enough.

B.1.2 Using Absolute Units with Clock Division

It may occasionally be desirable to use the same piece of hardware at different frequencies and frequency ratios. In this case, the timing parameters would not scale with the clock period but would be fixed at some value while the clock period changes.

If we are using clock division, then converting the timing parameters into units of dt is easy. Using the clock division circuit from Figure 3-1, we get (from Equations 3.3 and 6.2):

$$\Delta t = 1/f_{high} = T_{high} \tag{B.4}$$

where T_{high} is the period of Clk_{high} in seconds. Thus, if a timing parameter P, on *either* system, is equal to t_p seconds, we can express it in units of dt as follows:

$$P = \frac{t_p}{\Delta t} \cdot d = \frac{t_p}{T_{high}} \cdot d \tag{B.5}$$

where we choose d such that P is an integer.

When taking timing parameters in this mode, the program asks the user to enter t_p and T_{high} as integers. Although it is convenient to use standard time units such as picoseconds or nanoseconds in giving these values, any units that allow t_p and T_{high} to be expressed as integers can be used. This is because it is the *ratio* of t_p to T_{high} that is important, rather than their actual values. In fact, it is preferable to make t_p and T_{high} small integers whenever possible in order to avoid causing overflows during computation.

B.1.3 Using Absolute Units with Clock Multiplication

Using absolute units with clock multiplication is slightly more difficult. Here, the program assumes that Clk_n is being generated from Clk_m using the circuit in Figure 3-2, and asks the user to enter t_p and T_M as integers. From Equation B.3, we have

$$\Delta t = T_M / N \tag{B.6}$$

Thus, if a timing parameter P, on *either* system, is equal to t_p seconds, we can express it in units of dt as

$$P = \frac{t_p}{\Delta t} \cdot d = \frac{t_p \cdot N}{T_M} \cdot d \tag{B.7}$$

again choosing d to make sure that P is an integer. Note that the specific value of N is used here, and that this equation applies to timing parameters of *both* systems m and n. The asymmetry arises because we assume Clk_m to be the source clock and generate Clk_n from it. Thus, T_M is fixed, while T_N changes.

B.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. As can be seen from the source code in the next section, these algorithms were not implemented exactly as they have been described in the text of this thesis.

In implementing the original algorithm from section 3.3, for example, the program does *not* use "wrap-around" modulo arithmetic. That is, to check if it has passed P_{begin} on system m, it actually checks if it has passed $P_{begin+M}$. This makes the algorithm code simpler, and also makes it easier to correctly handle situations with abnormal timing parameters (i.e., timing parameters that are negative or are larger than a clock period).

Some minor modifications were also made to the three run-time scheduling algorithms. Instead of using units of Δt to represent timing parameters as done in Chapter 5, the program uses units of dt. Consequently, whenever M or N are used in the algorithms in Chapter 5 to represent the clock periods of the systems, they are replaced by $M \cdot d$, and $N \cdot d$ respectively. Instead of adding M to e_{old} in line 9 of the basic algorithm in section 5.1, for example, the program would add $M \cdot d$. Representing the timing parameters in this way makes it possible to correctly handle non-integer (in terms of Δt) timing parameters in the generalized algorithm.

B.3 Source Code

This section contains the complete C source code for the scheduling software.

```
/*
 * rcsched.c
 *
 * Scheduler for Rational Clocking, integer version
 *    by Luis F. G. Sarmenta
 *
 *    thesis version, 950525
 *
 * Notes: 1) Here, coincidence cycles are called "beat" cycles, as they
 *           are in Pratt and Ward's original paper.
 *
 *        2) All timing parameters are treated internally as offsets
 *           in the positive time direction from the clock edge they belong
 *           to.  Thus, setup times are internally negative.
 *           However, the user can still enter _positive_ setup times as
 *           these get converted automatically.
 *
 */

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>


/* standard min and max definitions */

#define min(a,b)   ((a) < (b) ? (a) : (b))
#define max(a,b)   ((a) > (b) ? (a) : (b))


/*
 * modulo and div operators modified to work correctly
 *    with negative numbers
 *
 * This code was developed on a Sun (mont-blanc.lcs.mit.edu) with gcc.
 *    Under this system type, the / and % operators work as follows:
 *        22 / 7 =  3;   22 % 7 =  1;   22 / -7 = -3;   22 % -7 =  1;
 *       -22 / 7 = -3;  -22 % 7 = -1;  -22 / -7 =  3;  -22 % -7 = -1;
 *
 *    The way it handles the mod of negative numbers allows us to
 *    define a correct mod operator as follows:
 */
```

```
long mod( long a, long b )
{
   long temp;

   temp = a % b;
   if ( temp < 0 )
      temp += b;
   return( temp );
}

long ndiv( long a, long b )                /*  ndiv(a,b)*b + mod(a,b) = a  */
{
   return( (a < 0) ? ((a-(b-1))/b) : (a/b) );
}

#define ndivup(a,b) ((ndiv((a)+((b)-1), (b))))       /* ndiv then round-up */

long gcd( long a, long b )
{
   if (!b)
      return( a );
   else
      return( gcd( b, a % b ) );
}


/*
 * Some Global Constants
 */


#define MAXMN   64              /* Max. value for M and N. */
#define ROM_B    4              /* bits for specifying M and N to ROM */
#define ROM_N   (1<<ROM_B)       /* equivalent max M and N for ROM */


long d  = 1000;                 /* smallest unit of time = (delta T) / d  */
long Tref;                      /* reference clock's period in picoseconds.
                                   This will be initialized properly later. */

/* Options Variables
 */

int PEEK = 0;                  /* Show intermediate results? */

int DRAWSCHEDS = 1;            /* Draw text schedule diagrams? */

#define DTUNITS 1
#define DLUNITS 2
#define FRUNITS 3
#define CDIVPS  4
#define CMULPS  5

int paramunits = DLUNITS;      /* Parameter Units */
```

```
#define PAIR 1
#define TABLE 2

int compmode = PAIR;            /* Compute Mode */

#define NONE 0
#define ORIG 1
#define GBUFF2 2
#define DBUFFG 3
#define DBUFFL 4
#define MAXALGO 4

int algorithm = 1;              /* Scheduling Algorithm */

#define TEXT  1
#define LDIAG 2
#define LPLOT 3
#define ROM   4

int outmode = LDIAG;            /* Output format */

int usewithchip = 0;            /* Use with prototype chip */

char outfilename[41] = "rcsched.log";       /* Log file name */

FILE *outfile;


/*
 * Primitive operations ('P' stands for primitive)
 *
 * These operations assume that M and N are integers representing
 * the normalized frequencies.
 *
 * Most of these operations work in units of dt = (delta t) / d.
 */

   /* The following macros are self-explanatory. */

#define Pbeatperiod(M,N)      ( (M)*(N)*d )
#define Pperiod(M,N)          ( (N)*d )
#define Pfreq(M,N)            ( 1 / Pperiod((M),(N)) )
#define Pdeltat(M,N)          d


   /* Pclockedge gets ith clock edge of system M  */

#define Pclockedge(M,N,i)     ( (i)*(N)*d )

   /* Pcycle returns the cycle which time t belongs to
    * Pupcycle returns the next cycle at or after time t
    * note that if t is exactly at some edge, then Pcycle and Pupcycle
    * return the same cycle.
    */
```

```
#define Pcycle(M,N,t)         (ndiv( (t), (d*(N)) ))
#define Pupcycle(M,N,t)       (ndivup( (t), (d*(N)) ))

   /* Pshcycle gets the current cycle if t is a timing param edge that is dt
    * units away from its clock edge.
    */

#define Pshcycle(M,N,t,dt)    (Pcycle((M),(N),(t)-(dt)))

   /* Pfracttodt converts from a timing param in the fractional form num/d
    * to its equivalent value in units of dt.
    * (Move this somewhere else?)
    */

#define Pfractodt(M,N,num)    ((num) * ( Pperiod((M),(N)) / d ) )
#define Pdelttodt(M,N,num)    ((num) * ( Pdeltat((M),(N)) ))

   /* The following macros convert from a timing param in units of
    * picoseconds to its equivalent in units of dt, using the reference
    * clock period, Tref.  Pmpstodt is used for clock multiplication,
    * where Tref is the clock period of M.  Pdpstodt is used for
    * clock division, where Tref is Thigh.
    *
    ******
    * Note that for now, these macros round _down_ to the nearest integer.
    * If you want to be conservative, you should really round _down_
    * if the value is contamination delay, but round _up_ otherwise.
    * Possibly fix this later.  For now, just pick d and Tref such that
    * the result is an integer anyway.
    ******
    */

#define Pmpstodt(M,N,num)     (((num) * (N) * d ) / Tref )
#define Pdpstodt(M,N,num)     (((num) * d) / Tref )

   /* Ptptodt converts the value tp to units of dt according to the
    * current units used for specifying tp.
    *
    * This allows the units of tp to change without changing the macros.
    * It also has the nice property that the appropriate scaling rule
    * for different M and N is naturally taken care of.
    *
    * Note that in these macros, parameters called 'dt' are in units
    * of dt, while parameters called 'tp' are in whatever units the scaling
    * rule specifies.
    */

long Ptptodt( long M, long N, long tp )
{
   switch (paramunits) {
      case DTUNITS : return( tp );
      case FRUNITS : return( Pfractodt( M, N, tp ) );
      case DLUNITS : return( Pdelttodt( M, N, tp ) );
      case CMULPS : return( Pmpstodt( M, N, tp ) );
```

```
      case CDIVPS : return( Pdpstodt( M, N, tp ) );
      default: {
                 fprintf(stderr,"Invalid Param Units ...  using dt ...\n" );
                 return( tp );
              }
   }
}


   /* Now we use Ptptodt in Pshedge, which gives timing edge i if
    * timing param is tp away from edge.
    */

#define Pshedge(M,N,i,tp)     ( Pclockedge((M),(N),(i)) + Ptptodt(M,N,tp) )

   /*
    * The following macros return the appropriate cycle of the _other_ clock.
    * (Note the switch in M and N.)
    */

#define PnxtOcycle(M,N,t)        Pupcycle( (N), (M), (t) )
#define PprvOcycle(M,N,t)        Pcycle( (N), (M), (t) )

#define PnxtOshcycle(M,N,t,dt)   Pupcycle( (N), (M), (t)-(dt) )
#define PprvOshcycle(M,N,t,dt)   Pshcycle( (N), (M), (t), (dt) )


/*
 * Data Structures
 *
 */

typedef int  sched[MAXMN];    /* Schedule table */

typedef struct systype {
   long f;             /* frequency = cycles per beat */
   long S,H,C,P;       /* timing params in units of dt = (delta t) / d */
   sched TE, Tsel;     /* transmit register control signals */
   sched TD;           /* input to transmit register */
   sched Tto;          /* cycle to transmit to (mod beat period) */
   sched RE, Rsel;     /* receive register control signals */
   sched RQ;           /* output of receive register */
   sched Rfrom;        /* cycle to receive from (mod beat period) */
   sched cnt;          /* counter value */
   sched checked;      /* Flag to tell if cycle has already been tried. */

   int   perf;         /* Performance as a transmitter. */

   struct systype *other;       /* pointer to the other system  */
}  systype;


/* These macros are the same as the primitive macros but
 * work with systype structs instead.  Note the capitalization of the names.
 */
```

```
#define Mf(M)              (M->f)
#define Nf(M)              ((M->other)->f)

#define Beatperiod(M)      Pbeatperiod( Mf(M), Nf(M) )
#define Period(M)          Pperiod( Mf(M), Nf(M) )
#define Freq(M)            Pfreq( Mf(M), Nf(M) )
#define DeltaT(M)          PdeltaT( Mf(M), Nf(M) )

#define Clockedge(M,i)     Pclockedge( Mf(M), Nf(M), (i) )
#define Cycle(M,t)         Pcycle( Mf(M), Nf(M), (t) )
#define SHcycle(M,t,dt)    Pshcycle( Mf(M), Nf(M), (t), (dt) )

#define Fractodt(M,num)    Pfractodt( Mf(M), Nf(M), (num) )
#define Delttodt(M,num)    Pdelttodt( Mf(M), Nf(M), (num) )

#define Tptodt(M,tp)       Ptptodt( Mf(M), Nf(M), (tp) )
#define SHedge(M,i,tp)     Pshedge( Mf(M), Nf(M), (i), (tp) )

#define NxtOcycle(M,t)     PnxtOcycle( Mf(M), Nf(M), (t) )
#define PrvOcycle(M,t)     PprvOcycle( Mf(M), Nf(M), (t) )
#define NxtOshcycle(M,t,dt) PnxtOshcycle( Mf(M), Nf(M), (t), (dt) )
#define PrvOshcycle(M,t,dt) PprvOshcycle( Mf(M), Nf(M), (t), (dt) )


/* The following macros allow us to access the different timing params
 * by name.  ('TP' stands for timing parameter.)  Valid values for S are
 * S, H, C, and P.  Note that these operations won't work as desired
 * if implemented as functions, and not macros.
 */

#define TP(M,S)                 Tptodt( M, (M->S) )
#define TPedge(M,i,S)           SHedge( M, (i), (M->S) )
#define TPcycle(M,t,S)          SHcycle( M, (t), TP(M,S) )

   /* These macros find the appropriate cycle corresponding to time t if
    * t is a timing parameter edge S.
    */

#define _NxtOTPcycle(M,t,S)     NxtOshcycle( M, (t), TP((M->other),S) )
#define _PrvOTPcycle(M,t,S)     PrvOshcycle( M, (t), TP((M->other),S) )

   /* NxtOTPcycle returns j when asked to find the next Sn edge j of the
    * other system at or after the Sm edge i of the original system (M).
    * PrvOTPcycle is similar but returns the edge at or _before_.
    */

#define NxtOTPcycle(M,i,Sm,Sn)   _NxtOTPcycle( M, TPedge(M,i,Sm), Sn )
#define PrvOTPcycle(M,i,Sm,Sn)   _PrvOTPcycle( M, TPedge(M,i,Sm), Sn )



/*
 * Global Variables
 *
```

```
 */

systype M, N;
long fM, fN;
long Sm, Hm, Cm, Pm;
long Sn, Hn, Cn, Pn;
int quit = 0;

unsigned char ROMimage[65536];



/*
 * Basic Output Routines
 *
 */

void change_outfile( char *filename )
{
      fclose( outfile );
      outfile = fopen( filename, "wt" );
}


void show( char *fmt, ... )
{
   va_list args;

   va_start( args, fmt );
   vprintf( fmt, args );
   if ( (outmode == TEXT) || (outmode == ROM) ) {
      vfprintf( outfile, fmt, args );
   }
   va_end( args );
}

void fshow( char *fmt, ... )
{
   va_list args;

   va_start( args, fmt );
   vfprintf( outfile, fmt, args );
   va_end( args );
}

void show2( char *fmt, ... )
{
   va_list args;

   va_start( args, fmt );
   vprintf( fmt, args );
   vfprintf( outfile, fmt, args );
   va_end( args );
}
```

```
/*
 * Initialization Routines
 *
 */

void ClearROM()
{
   long i;

   for ( i=0; i < 65536; i++ )
      ROMimage[i] = 0xFF;
   for ( i=0; i < (1 << 12); i++ )
      ROMimage[i] = 0x55;
}

/*
 * ClearSched clears the schedule tables used for
 *   transmitting from M to N.
 */

void ClearSched( systype *M, systype *N )
{
   int i;

   for (i=0; i < MAXMN; i++) {
      M->TE[i] = 0;    M->Tsel[i] = -1;   M->TD[i] = -1;
      N->RE[i] = 0;    N->Rsel[i] = -1;   N->RQ[i] = -1;
      M->checked[i] = 0;
      N->checked[i] = 0;
   }
}

/*
 * initcnt fills the cnt array with the proper counter values
 */

void initcnt( systype *M )
{
   int i;

   for (i=0; i < MAXMN; i++) {
      M->cnt[i] = i % (M->f);
   }
}

void initMN( systype *M, long fM, long Sm, long Hm, long Cm, long Pm,
             systype *N, long fN, long Sn, long Hn, long Cn, long Pn )
{
   int i;

   M->f = fM;
   M->other = N;
```

```
   M->S = -Sm;
   M->H = Hm;
   M->C = Cm;
   M->P = Pm;

   N->f = fN;
   N->other = M;

   N->S = -Sn;
   N->H = Hn;
   N->C = Cn;
   N->P = Pn;

   ClearSched( M, N );
   initcnt( M );
   ClearSched( N, M );
   initcnt( N );
}



/*
 * Scheduler Algorithms
 *
 */

/* No Flow Control.  Always use register 0. */

int UnctlTransmit( systype *M, systype *N )
{
   int i;

   for (i = 0; i < M->f; i++) {
         M->TE[i] = 1;
         M->Tsel[i] = 0;
   }
   for (i = 0; i < N->f; i++) {
         N->RE[i] = 1;
         N->Rsel[i] = 0;
   }
   return( 0 );
}

/*
 * Original single-buffer greedy algorithm.
 *
 * This is adapted from Pratt and Ward's original algorithm, described
 * in section 3.3 of my thesis.
 *
 */

int _OrigTransmit( systype *M, systype *N, int begin )
{
   long i, j;
   long i1, j1, nexti;
```

```
int count = 0;
long Pi, Sj, Hj, Ci;
long tcur, tbegin;


      /* STEP 2: Begin at Pbegin.
       */

i = begin;

Pi = TPedge(M,i,P);
Ci = TPedge(M,i,C);
tbegin = Pi + Beatperiod(M);   /* Note that we're _not_ using modulo math */
tcur = Pi;
   if (PEEK) {
      show( "\nBEGIN: i = %ld, Ci = %ld, Pi = %ld \n", i, Ci, Pi );
   }

do {
      /* STEP 3: Pi -> Sj.  Go to next Sj.
       */

   j = NxtOTPcycle(M,i,P,S);
   Sj = TPedge(N,j,S);
   tcur = Sj;
      if (PEEK) {
         Hj = TPedge(N,j,H);
         show( "j = %ld, Sj = %ld, Hj = %ld \n", j, Sj, Hj );
      }

      /* STEP 4: Go to Hj.
       */

   Hj = TPedge(N,j,H);
   tcur = Hj;

      /* STEP 5: Hj -> Ci'.  Go to next Ci.
       */

   nexti = NxtOTPcycle(N,j,H,C);
   Ci = TPedge(M,nexti,C);
   tcur = Ci;
      if (PEEK) {
         Pi = TPedge(M,nexti,P);
         show( "i' = %ld, Ci' = %ld, Pi' = %ld \n", nexti, Ci, Pi );
      }

      /* STEP 6: Go to next Pi and check.
       */

   Pi = TPedge(M,nexti,P);
   tcur = Pi;

      /* STEP 8: If Pbegin is not passed, mark the appropriate
       * schedule entries.
```

```
         */
      if ( tcur <= tbegin )  {
         i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

         M->TE[i1] = 1;
         M->Tsel[i1] = 0;
         M->TD[i1] = count % 0x10;
         M->Tto[mod(i,M->f)] = mod(j,N->f);

         N->RE[j1] = 1;
         N->Rsel[j1] = 0;
         N->RQ[mod(j,N->f)] = count % 0x10;
         N->Rfrom[mod(j,N->f)] = mod(i,N->f);

         count++;

          if (PEEK) {
             show( "TE[%d] and RE[%d] set. \n",
                     mod(i-1,M->f), mod(j-1,N->f) );
          }
      }

         /* STEPS 7 and 9: If you pass or hit Pbegin, then quit, else
          * make i = nexti and repeat the process.
          */

      i = nexti;

   } while (tcur < tbegin);

   return( count );
}

/* _Transmit schedules a transmission from M -> N and returns
 * the number of successful transmissions in a coincidence cycle.
 */

int _Transmit( systype *M, systype *N )
{
   int count, begin;
   int bestcount, bestbegin;
   int best;
   int found;

   best = min( M->f, N->f );
   bestcount = 0;
   bestbegin = 0;
   begin     = 0;

   while ( (bestcount != best) && (begin < M->f) ) {
      ClearSched( M, N );
      count = _OrigTransmit( M, N, begin );
      if (count > bestcount) {
         bestcount = count;
         bestbegin = begin;
```

```
      }
      begin++;
   }

   /* get lowest bestbegin. (if bestcount=best, then bestbegin
    * will also be the lowest bestbegin.)
    */

   if (bestcount != best) {
      ClearSched( M, N );
      count = _OrigTransmit( M, N, bestbegin );
   }
   return count;
}


/*
 * Lazy Double-Buffered Algorithm
 *
 */

int _DBuffLTransmit( systype *M, systype *N )
{
   long x, y;
   long eold, e;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
   int  c, count = 0;
   int  slowtx;

   slowtx = M->f <= N->f;

   if (slowtx) {
      m = Period(N);
      n = Period(M);
   } else {
      m = Period(M);
      n = Period(N);
   }

   P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
   S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

   x = 1;
   if ( slowtx )
      f0 = -( P - S );  /* Remember that S here is -S of S in the paper. */
   else
      f0 = -( H - C );
   y = ndiv( f0, n ) + 1;
   eold = mod( f0, n ) - n;
   R = 0;
      if (PEEK) {
```

```
        show( "f0: %ld, e0: %ld, x0: %ld, y0: %ld\n", f0, eold, x, y );
     }
   for (c=0; c < max(M->f,N->f); c++) {
     e = eold + m;
        if (PEEK) {
           show( "eold: %ld, e: %ld, x: %ld, y: %ld\n", eold, e, x, y );
        }
     if ( e >= 0 ) {
        if (slowtx) {
           i = y; j = x;
        } else {
           i = x; j = y;
        }

        i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

        M->TE[i1] = 1;
        M->Tsel[i1] = R;
        M->TD[i1] = count % 0x10;
        M->Tto[mod(i,M->f)] = mod(j+2,N->f);

        N->RE[j1] = 1;
        N->Rsel[j1] = R;
        N->RQ[mod(j+2,N->f)] = count % 0x10;
        N->Rfrom[mod(j+2,N->f)] = mod(i,M->f);

        R = (~R) & 1;
        y = y + 1;
        e = e - n;

        count++;

     }
     x = x + 1;
     eold = e;
   }

   return( count );
}


/*
 * Lazy Double-Buffered Algorithm
 *
 */

int _DBuffGTransmit( systype *M, systype *N )
{
   long x, y;
   long eold, e;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
```

```
int  c, count = 0;
int  slowtx;

slowtx = M->f <= N->f;

if (slowtx) {
   m = Period(N);
   n = Period(M);
} else {
   m = Period(M);
   n = Period(N);
}

P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

x = 1;
if ( !slowtx )
   x = x - 1;            /* For greedy algo, x = x_0 - 1 = 0 */
if ( slowtx )
   f0 = -( P - S );
else
   f0 = -( S - P );      /* The greedy algo transformation */
y = ndiv( f0, n ) + 1;
eold = mod( f0, n ) - n;
if (!slowtx) {           /* Adjust e0 and y0 for greedy algo */
   if ( mod(f0,n) == 0 )
      y = y - 1;
   if ( eold == -n )
      eold = 0;        /*******/
}
R = 0;
   if (PEEK) {
      show( "f0: %ld, e0: %ld, x0: %ld, y0: %ld\n", f0, eold, x, y );
   }
for (c=0; c < max(M->f,N->f); c++) {
   e = eold + m;
      if (PEEK) {
         show( "eold: %ld, e: %ld, x: %ld, y: %ld\n", eold, e, x, y );
      }
   if (( slowtx && (e >= 0) ) || ( !slowtx && (e > 0) )) {
      if (slowtx) {
         i = y; j = x;
      } else {
         i = x; j = y;
      }

      i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

      M->TE[i1] = 1;
      M->Tsel[i1] = R;
      M->TD[i1] = count % 0x10;
      M->Tto[mod(i,M->f)] = mod(j,N->f);

      N->RE[j1] = 1;
```

```
        N->Rsel[j1] = R;
        N->RQ[mod(j,N->f)] = count % 0x10;
        N->Rfrom[mod(j,N->f)] = mod(i,M->f);

        R = (~R) & 1;
        y = y + 1;
        e = e - n;

        count++;

     }
     x = x + 1;
     eold = e;
   }

   return( count );
}


/*
 * Generalized Algorithm
 *
 */

int _GBuff2Transmit( systype *M, systype *N )
{
   long x, y;
   long eold, e, emax, emin;
   long f0;
   int  R;
   long i, j, i1, j1;
   long P, S, H, C;
   long m, n;
   int  c, count = 0;
   int  slowtx;

   slowtx = M->f <= N->f;
/*                         Uncomment this to test slower system's algo.
   if (slowtx)
      _slowGBuff2Transmit( M, N );
   else {
*/
   if (slowtx) {
      m = Period(N);
      n = Period(M);
   } else {
      m = Period(M);
      n = Period(N);
   }

   P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
   S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

   emax = n - ( (P-C) + (H-S) );
   emin = m - n + ( (P-C) + (H-S) );
```

```
   x = 1;
   if ( !slowtx )
      x = x - 1;            /* For greedy algo, x = x_0 - 1 = 0 */
   if ( slowtx )
      f0 = -( P - S );
   else
      f0 = -( S - P );      /* The greedy algo transformation */
   y = ndiv( f0, n ) + 1;
   eold = mod( f0, n ) - n;
   if (!slowtx) {           /* Adjust e0 and y0 for greedy algo */
      if ( mod(f0,n) == 0 )
         y = y - 1;
      if ( eold == -n )
         eold = 0;    /*********/
   }

   R = ( slowtx ? 0 : 1 );

   for (c=0; c < max(M->f,N->f); c++) {
      e = eold + m;
      if (( slowtx && (e >= 0) ) || ( !slowtx && (e > 0) )) {
         if (slowtx) {
            i = y; j = x;
         } else {
            i = x; j = y;
         }

         i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

         if ( !slowtx )
            if ( R == 0 )
               R = ( (e < emin) ? 1 : 0 );
            else
               R = 0;

         M->TE[i1] = 1;
         M->Tsel[i1] = R;
         M->TD[i1] = count % 0x10;
         M->Tto[mod(i,M->f)] = mod(j,N->f);

         N->RE[j1] = 1;
         N->Rsel[j1] = R;
         N->RQ[mod(j,N->f)] = count % 0x10;
         N->Rfrom[mod(j,N->f)] = mod(i,M->f);

/* Adjust count only to measure single-buffered performance.
 * Note that as a consequence of this, TD would not be incremented
 * whenever TSel is 1.
 */
         if ( R == 0 )
            count++;

         if ( slowtx )
            if ( R == 0 )
```

```
            R = ( (e > emax) ? 1 : 0 );
         else
            R = 0;

      y = y + 1;
      e = e - n;
    }

    x = x + 1;
    eold = e;
  }
/* }                     Uncomment this to test slower system's algo.
*/
  return( count );
}

/*
 * Slower system's algorithm
 *
 */

int _slowGBuff2Transmit( systype *M, systype *N )
{
  long x, y;
  long eold, e, emax, emin;
  long f0;
  int  R;
  long i, j, i1, j1;
  long P, S, H, C;
  long m, n;
  int  c, count = 0;
  int  slowtx;
  long dMN;

/* assume M < N */

    m = Period(N);
    n = Period(M);
    dMN = n % m;

  P = Tptodt( M, M->P );   C = Tptodt( M, M->C );
  S = Tptodt( N, N->S );   H = Tptodt( N, N->H );

  emax = n - ( (P-C) + (H-S) );
  emin = m - n + ( (P-C) + (H-S) );

  f0 = -( P - S );
  y = ndiv( f0, n ) + 1;

  eold = mod( f0, n ) - n;
  e = mod(eold,m);

  R = 0;

  for (c=0; c < M->f; c++) {
```

```
      i = y; j = x;
      i1 = mod(i-1,M->f); j1 = mod(j-1,N->f);

      M->TE[i1] = 1;
      M->Tsel[i1] = R;
      M->TD[i1] = count % 0x10;
      M->Tto[mod(i,M->f)] = mod(j,N->f);

      if ( R == 0 )
         count++;

      if ( slowtx )
         if ( R == 0 )
            R = ( (e > emax) ? 1 : 0 );
         else
            R = 0;

      e = e - dMN;
      if ( e < 0 )
         e = e + m;

      y = y + 1;
   }

   return( count );
}


/*
 * Higher-level Scheduling Routines
 *
 */

int Transmit( systype *M, systype *N )
{
   ClearSched( M, N );
   switch( algorithm ) {
      case NONE : return( UnctlTransmit( M, N ) );
      case ORIG : return( _Transmit( M, N ) );
      case DBUFFL : return( _DBuffLTransmit( M, N ) );
      case DBUFFG : return( _DBuffGTransmit( M, N ) );
      case GBUFF2 : return( _GBuff2Transmit( M, N ) );
      default : return( 0 );
   }
}

int MakeSched( systype *M, systype *N )
{
   int a, b, c, opt;

   opt = min( M->f, N->f );

      if (PEEK) {
         show("M -> N\n");
      }
```

```
   a = Transmit( M, N );
      if (PEEK) {
         show("\nN -> M\n");
      }
   b = Transmit( N, M );

   M->perf = ((double) a / (double) opt) * 100;
   N->perf = ((double) b / (double) opt) * 100;

   c   = min( M->perf, N->perf );

   return( c );
}



/*
 * Display Routines
 *
 */

void PlainSched( systype *M, int *schedptr )
{
   int i;

   for (i=0; i < M->f; i++)
      if (schedptr[i] == (-1))
         show("x");
      else
         show("%X",schedptr[i]);
   show("\n");
}


void DrawSched( systype *M, int *schedptr )
{
   int i, j;
   long Mw, Nw, MNgcd;

   Mw = Nf(M);
   Nw = Mf(M);
   MNgcd = gcd( Mw, Nw );
   Mw /= MNgcd;
   Nw /= MNgcd;

   if ((Mw == 1) || (Nw == 1))
      Mw = 2*Mw;

   show("|");
   for( i = 0; i < M->f; i++ ) {
      for( j = 0; j < Mw-1; j++ ) {
         if (j == (Mw-1)/2 )
            if (schedptr[i] == (-1))
               show("x");
            else
```

```
            show("%X", schedptr[i] );
         else
            show(" ");
      }
      show("|");
   }
   show("\n");
}

#define PrintSched(M,sptr)  ( (DRAWSCHEDS) ? DrawSched(M,sptr) : \
                                            PlainSched(M,sptr)  )


void ShowTimes( systype *M, systype *N )
{
   show("\n");
   show( "Tref = %ld\n", Tref );
   show("TM:  TM = %ld, tS = %ld,  tH = %ld,  tC = %ld,  tP = %ld\n",
         Period(M), TP(M,S), TP(M,H), TP(M,C), TP(M,P));
   show("TN:  TN = %ld, tS = %ld,  tH = %ld,  tC = %ld,  tP = %ld\n",
         Period(N), TP(N,S), TP(N,H), TP(N,C), TP(N,P));
   show("\n");
}

void ShowSched( systype *M, systype *N )
{
   int i;

   show("\n");
   show("M:  f = %ld, S = %ld,  H = %ld,  C = %ld,  P = %ld\n",
          M->f, M->S, M->H, M->C, M->P);
   show("N:  f = %ld, S = %ld,  H = %ld,  C = %ld,  P = %ld\n",
          N->f, N->S, N->H, N->C, N->P);
   show("\n");

   show("M -> N\n");
   show("--------\n");
   show("N.RE   : ");
   PrintSched( N, N->RE );
   show("M.TE   : ");
   PrintSched( M, M->TE );
   show("--------\n");
   show("N.Rsel : ");
   PrintSched( N, N->Rsel );
   show("M.Tsel : ");
   PrintSched( M, M->Tsel );
   show("--------\n");
   show("N.RQ   : ");
   PrintSched( N, N->RQ );
   show("M.TD   : ");
   PrintSched( M, M->TD );

   show("\n");

   show("N -> M\n");
```

```
   show("--------\n");
   show("N.TE   : ");
   PrintSched( N, N->TE );
   show("M.RE   : ");
   PrintSched( M, M->RE );
   show("--------\n");
   show("N.Tsel : ");
   PrintSched( N, N->Tsel );
   show("M.Rsel : ");
   PrintSched( M, M->Rsel );
   show("--------\n");
   show("N.TD   : ");
   PrintSched( N, N->TD );
   show("M.RQ   : ");
   PrintSched( M, M->RQ );
   show("--------\n");

   show("Performance:  M->N = %d%%, N->M = %d%% \n", M->perf, N->perf );
}

void ROMsched( systype *M, systype *N )
{
   long baseaddr, offs;
   long Mi, Nj;
   int  REM, RselM, TE0M, TE1M;
   int  REN, RselN, TE0N, TE1N;

#define nneg( x ) ( ((x) < 0) ? 0 : x )
#define bit0( x ) ( (x) & 1 )


   baseaddr = ((Mf(M)-1) << (ROM_B*2))  + ((Mf(N)-1) << ROM_B);

   for (offs=0; offs < ROM_N; offs++) {
      Mi = (usewithchip ? (Mf(M) - offs + 1) : offs);
      Nj = (usewithchip ? (Mf(N) - offs + 1) : offs);
      Mi = mod( Mi, Mf(M) );
      Nj = mod( Nj, Mf(N) );

      TE0M = bit0( M->TE[Mi] & ~(nneg(M->Tsel[Mi])) );
      TE1M = bit0( M->TE[Mi] & nneg(M->Tsel[Mi]) );
      REM  = bit0( M->RE[Mi] );
      RselM = bit0( nneg( M->Rsel[Mi] ) );
      TE0N = bit0( N->TE[Nj] & ~(nneg(N->Tsel[Nj])) );
      TE1N = bit0( N->TE[Nj] & nneg(N->Tsel[Nj]) );
      REN  = bit0( N->RE[Nj] );
      RselN = bit0( nneg( N->Rsel[Nj] ) );

      ROMimage[baseaddr+offs] = TE0M + (TE1M << 1)
                              + (REN << 2) + (RselN << 3)
                              + (TE0N << 4) + (TE1N << 5)
                              + (REM << 6) + (RselM << 7);
```

```
   }
}

void IntelMCS( systype *M, systype *N )
{
   long baseaddr, offs;
   long chksum = 0;

   baseaddr = ((Mf(M)-1) << (ROM_B*2))  + ((Mf(N)-1) << ROM_B);

   show( ":%02X%04X00", ROM_N, baseaddr );
   chksum = ROM_N + (baseaddr >> 8) + (baseaddr & 0xFF);
   for (offs=0; offs < ROM_N; offs++) {
       show( "%02X", ROMimage[baseaddr+offs] );
       chksum += ROMimage[baseaddr+offs];
   }
   chksum = (-chksum) & 0xFF;
   show( "%02X\n", chksum );
}

void LaTeXsched( systype *M, int *schedptr, int ypos )
{
   int i;
   char s[10];

#define entry(x,y,s) fshow("\\put(%d,%d){\\makebox(%d,2){%s}}\n", x, y, \
                          M->other->f, s )

   for (i=0; i < M->f; i++) {

      if (schedptr[i] == (-1))
         strcpy( s, "*" );
      else
         sprintf( s, "%d", schedptr[i] );

      entry( i*(M->other->f), ypos, s );
   }
}

void LaTeXvals( systype *M, systype *N )
{

#define label(y,s) fshow( "\\put(-1,%d){\\makebox(0,0)[r]{$%s$}}\n", y, s )
#define Lsched( y, sys, S ) label( (y+1), #S ); LaTeXsched( sys, sys->S, y );

   Lsched( 13, N, RE );
   Lsched( 15, N, Rsel );
   Lsched( 17, N, RQ );

   Lsched( -4, M, TE );
   Lsched( -6, M, Tsel );
   Lsched( -8, M, TD );

}
```

```
void LaTeXparams( systype *M, systype *N )
{
   double Mf = M->f;
   double Nf = N->f;

#define set( x, v )   fshow( "\\renewcommand{\\" #x "}{%g}\n", v )
#define gdttodelt( x ) ((double)(x) / (double) d )

   set( M, Mf );
   set( Mo, Mf+1 );
   set( nM, -Mf );

   set( N, Nf );
   set( No, Nf+1 );
   set( nN, -Nf );

   set( MN, Mf*Nf );

   set( CM, gdttodelt(TP(M,C)) );
   set( PM, gdttodelt(TP(M,P)) );
   set( CPM, gdttodelt( TP(M,P) - TP(M,C) ) );
   set( SN, gdttodelt( -TP(N,S) ) );
   set( NSN, Mf+gdttodelt( (TP(N,S)) ) );
   set( HN, gdttodelt( TP(N,H) ) );


}

void LaTeXdiag( systype *M, systype *N )
{
/* Note: This will fclose the current outfile. */
   change_outfile( "sched.tex" );

   fshow( "\\documentstyle{article}\n"
          "\\begin{document}\n"
          "\\setlength{\\unitlength}{.1in}\n"
          "\\newcounter{cM}\n"
          "\\newcounter{cN}\n"
          "\\input{setup}\n"         );
   LaTeXparams( M, N );
   fshow( "\\begin{picture}(%d,31)(-11,-9)\n"
          "\\input{clocks}\n",
          (M->f)*(N->f)+11         );
   LaTeXvals( M, N );
   fshow( "\\end{picture}\n"
          "\n"        );
   LaTeXparams( N, M );
   fshow( "\\begin{picture}(%d,27)(-11,-7)\n"
          "\\input{clocks}\n",
          (M->f)*(N->f)+11         );
   LaTeXvals( N, M );
   fshow( "\\end{picture}\n"
          "\\end{document}\n"        );
   fclose( outfile );
```

```
/* Use "at" here to resume writing log file. */
   outfile = fopen( outfilename, "at" );
}

void singlesched( systype *M, systype *N )
{
   MakeSched( M, N );
   ShowSched( M, N );
   if ( outmode == LDIAG )
      LaTeXdiag( M, N );
   else if ( outmode == ROM ) {
           ROMsched( M, N );
           IntelMCS( M, N );
        }
}

void tablesched( systype *M, systype *N )
{
   int oldM, oldN;
   int tx, rx, maxfreq;
   int LaTeX = ((outmode == LDIAG) || (outmode == LPLOT));

   oldM = M->f; oldN = N->f;
   maxfreq = max( oldM, oldN );

   if ( LaTeX ) {
      fshow( "{\\scriptsize\n"  );
      fshow( "\\begin{tabular}{l|" );
      for (rx=1;rx<=maxfreq;rx++)
         fshow( "r" );
      fshow( "}\nTransmit & \\multicolumn{%d}{c}{Receive Frequency} \\\\ \n",
            maxfreq );
   } else
      show("Transmit\tReceive Frequency\n");

   show2( "Freq." );
   for (rx=1;rx<=maxfreq;rx++) {
      if ( LaTeX )
          fshow( " & " );
      show( "\t");
      show2( "%d", rx );
   }
   if ( LaTeX )
      fshow( " \\\\ \\hline" );
   show2("\n" );
   for (tx = 1; tx <= maxfreq; tx++) {
      show2( "%d", tx );
      if ( LaTeX )
         fshow( " & " );
      show( "\t" );
      for (rx = 1; rx <= maxfreq; rx++) {
         M->f = tx; N->f =rx;
         MakeSched( M, N );
         show2("%d", M->perf );
         if (rx < maxfreq) {
```

```
          if ( LaTeX )
             fshow( " & " );
          show("\t");
       } else {
          if ( LaTeX )
             fshow( " \\\\\n" );
          show("\n");
       }
    }
  }
  if ( LaTeX )
     fshow( "\\end{tabular}\n}" );
  show( "\n" );
  M->f = oldM; N->f = oldN;
}

void ROMtable( systype *M, systype *N, long seg )
{
  int oldM, oldN;
  int tx, rx, maxfreq;
  long chksum;

  oldM = M->f; oldN = N->f;
  maxfreq = max( oldM, oldN );

  chksum = 2 + 2 + (seg >> 8) + (seg & 0xFF);
  chksum = (-chksum) & 0xFF;
  show( ":02000002%04X%02X\n", seg, chksum );

  for (tx = 1; tx <= maxfreq; tx++) {
     for (rx = 1; rx <= maxfreq; rx++) {
        M->f = tx; N->f =rx;
        MakeSched( M, N );
        ROMsched( M, N );
        IntelMCS( M, N );
     }
  }

  show( ":00000001FF\n" );
}

void bigROMtable( systype *M, systype *N )
{
  int algo;
  int oldalgo = algorithm;

  for (algo = 0; algo <= MAXALGO; algo++ ) {
     algorithm = algo;
     ROMtable( M, N, (algo << (3*ROM_B - 4)) );
  }
  algorithm = oldalgo;
}
```

```
/*
 * User Interface (Menu) Data Structures and Routines
 *
 * This is a very simple menu system.  I'm tempted to use C++
 * here, but I'll keep it simple to make it portable.
 *
 */

#define MAXNAME  50
#define MAXITEMS 9

typedef struct procitem {
   char name[MAXNAME];      /* name of the item */
   char value[MAXNAME];     /* string to be printed beside the name */
   void (*proc)(char *);    /* procedure to execute when the item is chosen */
} procitem;

typedef struct procmenu {
   char name[MAXNAME];       /* name of menu */
   int  numitems;            /* number of items in menu */
   procitem *item[MAXITEMS]; /* array of pointers to items */
} procmenu;

typedef char valitem[MAXNAME]; /* valitem is just a string with the name */

typedef struct valmenu {
   char name[MAXNAME];      /* name of menu */
   int  *variable;          /* pointer to the variable to be changed */
   int  numitems;           /* number of items in menu */
   valitem item[MAXITEMS];  /* array of valitems (array of strings)  */
} valmenu;


/*
 * General ValMenu Routine
 *
 */

void do_valmenu( valmenu *M )
{
   char s[10];
   int i, choice;
   int quit = 0;

   printf( "\n%s\n\n", M->name );
   for ( i=0; i < (M->numitems); i++ )
      printf( "%d) %s\n", i+1, M->item[i] );
   do {
      choice = *(M->variable);
      printf( "\nEnter your choice (1-%d) [%d]: ", M->numitems+1,
              choice );
      gets( s );
      sscanf( s, "%d", &choice );
   } while ((choice < 1) || (choice > (M->numitems)));
   *(M->variable) = choice;
```

```
}


/*
 * ValMenu definitions
 *
 */

valmenu PUnits = { "Units for Timing Parameters", &paramunits, 5,
                  { "dt",
                    "delta t",
                    "1/d of clock period",
                    "Absolute units (clk div)",
                    "Absolute units (clk mul)",
                    "", "", "", ""
                  }
                };

valmenu CompMode = { "Compute Mode", &compmode, 2,
                  { "Pair",
                    "Table",
                    "",
                    "", "", "", "", "", ""
                  }
                };

valmenu OutMode = { "Output Format", &outmode, 4,
                   { "Text",
                     "LaTeX schedule diagram",
                     "LaTeX schedule plot",
                     "ROM",
                     "", "", "", "", ""
                   }
                 };

valmenu Algo = { "Scheduling Algorithm", &algorithm, 4,
                { "Single Buffered",
                  "Gen. Online (2 buffers)",
                  "Double Buffered (Greedy)",
                  "Double Buffered (Lazy)",
                  "", "", "", "", ""
                }
              };


/*
 * General ProcMenu Routine
 *
 */

void do_procmenu( procmenu *M )
{
   char s[10];
   int i, choice;
   int quit = 0;
```

```
   do {
      printf( "\n>>> %s <<<\n\n", M->name );
      for ( i=0; i < (M->numitems); i++ ) {
         printf( "%d) %s", i+1, M->item[i]->name );
         if ( M->item[i]->value[0] != 0 )
            printf( "[%s]", M->item[i]->value );
         printf( "\n" );
      }
      do {
         printf( "\nEnter your choice (1-%d, <cr> to go back) : ",
                 M->numitems );
         gets( s );
         if ( (s[0] == 0) )
            choice = (M->numitems+1);
         else
            sscanf( s, "%d", &choice );
      } while ((choice < 1) || (choice > (M->numitems+1)));
      printf("\n\n");
      if (choice != M->numitems+1)
         (*(M->item[choice-1]->proc))( M->item[choice-1]->value );
   } while (choice != M->numitems+1);
}


/*
 * Input Routines
 *
 * These routines as the user for input.  Pressing CR keeps the variable's
 * old value.
 *
 */

char YN[2] = "ny";

void getYN( char *s, int *x )
{
   char temp[20];

   temp[0] = 0;
   printf( "%s (y/n)? [%c] ", s, YN[*x] );
   gets( temp );

   if ( temp[0] == 'y' )
      *x = 1;
   else if ( temp [0] == 'n' )
           *x = 0;
}

int askYN( char *s, int defval )
{
   int x = defval;

   getYN( s, &x );
   return( x );
```

```
}

void getstring( char *msg, char *s )
{
   char temp[40];

   printf( "%s [%s]: ", msg, s );
   gets( temp );
   if ( temp[0] != 0 )
      strcpy( s, temp );
}

void getlong( long *x )
{
   char temp[20];

   gets( temp );
   sscanf( temp, "%ld", x );
}

#define getTP(x)   printf(#x " [%ld] : ", x);  \
                  getlong(&x);


/*
 * Menu item procedures
 *
 */

void do_Params( char *s )
{
   getTP(d);

   printf( "\nM, N [%ld, %ld] : ", fM, fN );
   gets( s );
   sscanf( s, "%ld, %ld", &fM, &fN );

   printf( "\nEnter the following timing parameters in units of %s:\n\n",
           PUnits.item[paramunits-1] );

   if ( (paramunits == CMULPS) || (paramunits == CDIVPS)) {
      getTP(Tref);
   }

   getTP(Sm); getTP(Hm); getTP(Cm); getTP(Pm);
   getTP(Sn); getTP(Hn); getTP(Cn); getTP(Pn);

/* Note: this call uses the _global_ variables M and N. */
   initMN( &M, fM, Sm, Hm, Cm, Pm, &N, fN, Sn, Hn, Cn, Pn );

      if (PEEK) {
         ShowTimes( &M, &N );
      }
   s[0] = 0;
}
```

```
void converttodt( systype *M, systype *N, int oldunits )
{
   paramunits = oldunits;

   Sm = -TP(M,S);   Hm = TP(M,H);   Cm = TP(M,C);   Pm = TP(M,P);
   Sn = -TP(N,S);   Hn = TP(N,H);   Cn = TP(N,C);   Pn = TP(N,P);

   paramunits = DTUNITS;

   initMN( M, fM, Sm, Hm, Cm, Pm, N, fN, Sn, Hn, Cn, Pn );

      if (PEEK) {
         ShowTimes( M, N );
      }
}


void do_PUnits( char *s )
{
   int oldunits = paramunits;

   do_valmenu( &PUnits );
   if ((paramunits != oldunits)) {
      if (paramunits == DTUNITS) {
         if ( askYN( "Convert to dt units?", 0 ) ) {
            printf( "\nConverting to dt units ... " );
            converttodt( &M, &N, oldunits );
            printf( "\n\n" );
         }
      }
   }
   s[0] = 0;
}

void do_CompMode( char *s )
{
   do_valmenu( &CompMode );
   s[0] = 0;
}

void do_OutMode( char *s )
{

   do_valmenu( &OutMode );
   s[0] = 0;
}

void do_Algo( char *s )
{
   do_valmenu( &Algo );
   s[0] = 0;
}

void do_Output( char *s )
```

```
{
   if (compmode == TABLE) {
      if ( outmode == ROM )
         bigROMtable( &M, &N );
      else
         tablesched( &M, &N );
   }
   else
      singlesched( &M, &N );
   s[0] = 0;
   fflush( outfile );
}

void do_OutOpts( char *s )
{
   char temp[41];
   strcpy( temp, outfilename );

   getstring( "Enter Log File Name", outfilename );
   if (strcmp(outfilename, temp)) {
      change_outfile( outfilename );
   }

   getYN( "Show intermediate results", &PEEK );
   getYN( "Draw text schedule diagram", &DRAWSCHEDS );

   getYN( "Use with prototype chip", &usewithchip );
   s[0] = 0;
}

/*
 * ProcItem definitions
 *
 */

procitem I_PUnits = { "Parameter Units", "", do_PUnits };
procitem I_Algo = { "Scheduling Algorithm", "", do_Algo };
procitem I_CompMode = { "Compute Mode", "", do_CompMode };
procitem I_OutMode = { "Output Format", "", do_OutMode };
procitem I_OutOpts = { "Output Options", "", do_OutOpts };

/* Hmm ... this is messy but I need to define this before defining
 * do_Options.
 */

procmenu OptionsMenu = { "Options Menu", 5,
                        { &I_PUnits, &I_Algo, &I_CompMode,
                          &I_OutMode, &I_OutOpts, NULL,
                          NULL, NULL, NULL
                        }
                      };

void do_Options( char *s )
{
   do_procmenu( &OptionsMenu );
```

```
   s[0] = 0;
}



/*
 * ProcMenu definitions
 *
 */

procitem I_Params  = { "Enter Parameters", "", do_Params };
procitem I_Output  = { "Generate Output", "", do_Output };
procitem I_Options = { "Options", "", do_Options };


procmenu MainMenu = { "Main Menu", 3,
                      { &I_Params, &I_Output, &I_Options,
                        NULL, NULL, NULL, NULL, NULL, NULL
                      }
                    };



/*
 * Main Program Routines
 *
 */

void initdisplay()
{
   ClearROM();
   outfile = fopen( outfilename, "wt" );
}

void shutdown()
{
   fclose( outfile );
}


main()
{
   initdisplay();
   do {
      do_procmenu( &MainMenu );
      getYN( "Do you really want to quit", &quit );
   } while (!quit);
   shutdown();
}
```

B.4 LATEX Support Files

The *sched.tex* file produced by the scheduling software requires two support files: *setup.tex* and *clocks.tex*.

```
% setup.tex
%
\newcommand{\M}{5}%
\newcommand{\Mo}{6}%
\newcommand{\nM}{-5}%
%
\newcommand{\N}{6}%
\newcommand{\No}{7}%
\newcommand{\nN}{-6}%
%
\newcommand{\MN}{30}%
%
\newcommand{\CM}{1}%
\newcommand{\PM}{3}%
\newcommand{\CPM}{2}%
\newcommand{\SN}{1}%
\newcommand{\NSN}{4}  %  >>>> Don't forget this!  Set to (M-SN) %
\newcommand{\HN}{1}%
```

```
% clocks.tex
%
\setcounter{cM}{0}
\setcounter{cN}{0}
%
% Draw M cycles
\multiput(0,-1)(0,3){2}{\line(1,0){\MN}}
\multiput(0,-1)(\N,0){\Mo}{\line(0,1){3}}
\multiput(0,1.5)(1,0){\MN}{\line(0,1){.5}}
\multiput(0,-1)(\N,0){\M}{\makebox(\N,2.5){\arabic{cM}}\addtocounter{cM}{1}}
%
% Draw C edges
\multiput(\CM,3)(\N,0){\M}{\line(0,1){1}}
%
% Draw P edges
\multiput(\PM,3)(\N,0){\M}{\line(0,1){1}}
%
% Draw C-P lines
\multiput(\CM,3.5)(\N,0){\M}{\line(1,0){\CPM}}
%
% Draw C's
\multiput(\CM,3)(\N,0){\M}{\makebox(0,0)[r]{\tiny C\,}}
% Draw P's
\multiput(\PM,3)(\N,0){\M}{\makebox(0,0)[l]{\tiny \,P}}
%
%
% Draw N cycles
```

```
\multiput(0,9)(0,3){2}{\line(1,0){\MN}}
\multiput(0,9)(\M,0){\No}{\line(0,1){3}}
\multiput(0,9)(1,0){\MN}{\line(0,1){.5}}
\multiput(0,9.5)(\M,0){\N}{\makebox(\M,2.5){\arabic{cN}}\addtocounter{cN}{1}}
%
% Draw S edges
\multiput(\NSN,7)(\M,0){\N}{\line(0,1){1}}
%
% Draw H edges
\multiput(\HN,7)(\M,0){\N}{\line(0,1){1}}
%
% Draw S
\multiput(\NSN,8)(\M,0){\N}{\makebox(0,0)[r]{\tiny S\,}}
%
% Draw S-Clk lines
\multiput(\NSN,7.5)(\M,0){\N}{\line(1,0){\SN}}
%
% Draw H
\multiput(\HN,8)(\M,0){\N}{\makebox(0,0)[l]{\tiny \,H}}
%
% Draw H-Clk lines
\multiput(\HN,7.5)(\M,0){\N}{\line(-1,0){\HN}}
```

Appendix C

The Prototype Chip

Figure C-1 shows the layout and pin assignments of the prototype rational clocking chip. Details of its features and its intended use are provided in Chapter 6.

Figure C-1: Layout and pinout of the prototype rational clocking chip.

Appendix D

Verilog Source Code

This appendix contains the complete Verilog source code of the run-time scheduling hardware model. The code is divided into five sections of descending hierarchical order. The top-level modules are presented first, followed by the high-level communication scheduling hardware modules. The components that make up the scheduling hardware are presented next, followed by the circuits that compute the constants used by these components. The listing ends with the basic components, such as registers and multiplexers, which are used by all the other higher-level modules.

D.1 Top-Level Modules

```
/* params.vh
 *
 * This file contains the timing parameters in units of Delta t.
 * It is included in a number of modules.  Modify this file to modify
 * the timing parameters.
 *
 */

   `define M 5'd5
   `define N 5'd6

   `define CM 5'd1
   `define PM 5'd3
   `define SM 5'd1
   `define HM 5'd1

   `define CN 5'd1
```

```
   `define PN 5'd3
   `define SN 5'd1
   `define HN 5'd1


/* driver.v
 *
 * Driver / Test Module
 *
 * Luis F. G. Sarmenta 950514
 *
 */

/* Define the algorithm/hardware to use */

`define GENB


/* Define which wave forms to view */

`define VIEW_M_TO_N


module driver;

   parameter width = 5;

   reg clkhi, clr;
   wire clkM, clkN;
   wire [width-1:0] cntM, cntN, dM, dN, qM, qN, qinM, qinN;

   wire TEM,TSelM,REN,RSelN,TEN,TSelN,REM,RSelM;

   wire validN, validM;
   wire dlyREM, dlyREN, dlyRSelM, dlyRSelN;

`include "params.vh"

   parameter SB = 1'b0;

   parameter M = `M;
   parameter N = `N;

   parameter CM = `CM;
   parameter PM = `PM;
   parameter SM = `SM;
   parameter HM = `HM;

   parameter CN = `CN;
   parameter PN = `PN;
   parameter SN = `SN;
   parameter HN = `HN;


/*
```

```
 * Clock Generation (Clock Division) Counters
 *
 */

   fdiv #(width) genclkM( clkM, , clkhi, N, 1'b0, clr );
   fdiv #(width) genclkN( clkN, , clkhi, M, 1'b0, clr );

/*
 * Scheduling Hardware
 *
 */

`ifdef NOCNTL                          /* No flow control */
   /* Index Counters (the other hardware options have built-in
      index counters) */
   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   assign TEM = 1'b1;
   assign TSelM = 1'b1;
   assign REN = 1'b1;
   assign RSelN = 1'b1;
   assign TEN = 1'b1;
   assign TSelN = 1'b1;
   assign REM = 1'b1;
   assign RSelM = 1'b1;
`endif

`ifdef DBLAZY                          /* Double-buffered Lazy */
   dbstfrl #(width) stfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );

   dbftsrl #(width) ftsr( , , REM, RSelM, , , TEN, TSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );
`endif

`ifdef DBGREEDY                        /* Double-buffered Greedy */
   dbstfrg #(width) stfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );

   dbftsrg #(width) ftsr( , , REM, RSelM, , , TEN, TSelN,
                          /* clkhi, */ clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN
                          /* SB */ );
`endif

`ifdef GENB                            /* Generalized Scheduling */
   gbstfr #(width) stfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                          clkhi, clkM, clkN, clr, M, N,
```

```
                          CM, PM, SM, HM, CN, PN, SN, HN,
                          SB );

   gbftsr #(width) ftsr( , , REM, RSelM, , , TEN, TSelN,
                          clkhi, clkM, clkN, clr, M, N,
                          CM, PM, SM, HM, CN, PN, SN, HN,
                          SB );
`endif


/*
 * Data Channels
 *
 */

        /* M to N */

   counten #(width) dataM( , dM, clkM, TEM, clr );

   /* Note: Here we delay RSel by the receiver's C delay to prevent RSel
    * from causing a hold time violation.  */

   assign #(CN*100) dlyRSelN = RSelN;
   assign #(CN*100) dlyREN = REN;
   dbout   #(width,CM*100,PM*100) outM( qM, clkM, dM, TEM, TSelM, dlyRSelN );

   rxregN   #(width) inN( qinN, clkN, qM, dlyREN );
   regen   vN( validN, clkN, REN, 1'b1 );

        /* N to M */

   counten #(width) dataN( , dN, clkN, TEN, clr );

   assign #(CM*100) dlyRSelM = RSelM;
   assign #(CM*100) dlyREM = REM;
   dbout   #(width,CN*100,PN*100) outN( qN, clkN, dN, TEN, TSelN, dlyRSelM );

   rxregM   #(width) inM( qinM, clkM, qN, dlyREM );
   regen   vM( validM, clkM, REM, 1'b1 );


/*
 * Clkhi and Reset Generators
 *
 */

        /* clock period = 100 */

   initial
      forever
      begin
         clkhi = 1;
         #50;
         clkhi = 0;
         #50;
```

```
      end

        /* reset on start */

   initial
      begin
         clr = 0 ;
         #150;
         clr = 1 ;
         #100;
         clr = 0 ;
      end

/*
 * Display Routines
 *
 */

/*  Primitive text output for non-graphics terminals

         always @( cntM )
            $strobe( "cM=%d, %b%b %b%b", cntM[width-1:0],
                     TEM, TSelM, REM, RSelM );

         always @( cntN )
            $strobe( "\t\tcN=%d, %b%b %b%b", cntN[width-1:0],
                     REN, RSelN, TEN, TSelN,  );
*/

/* Graphical Output */

   initial
      begin
         $gr_waves(
`ifdef VIEW_M_TO_N
                   "cntN", cntN[width-1:0],
                   "qinN", qinN[width-1:0],
                   "validN", validN,
                   "REN'", dlyREN,
                   "RSelN'", dlyRSelN,

                   "qM", qM[width-1:0],
                   "q0M", outM.q0[width-1:0],
                   "q1M", outM.q1[width-1:0],
                   "TSelM", TSelM,
                   "TEM", TEM,
                   "dM", dM[width-1:0],
                   "cntM", cntM[width-1:0]
`endif
`ifdef VIEW_N_TO_M
                   "cntN", cntN[width-1:0],
                   "dN", dN[width-1:0],
                   "TEN", TEN,
                   "TSelN", TSelN,
                   "q0N", outN.q0[width-1:0],
```

```
                  "q1N", outN.q1[width-1:0],
                  "qN", qN[width-1:0],

                  "RSelM'", dlyRSelM,
                  "REM'", dlyREM,
                  "validM", validM,
                  "qinM", qinM[width-1:0],
                  "cntM", cntM[width-1:0]
`endif
                );

     end

endmodule
```

D.2 Schedulers and Data Channels

```
/* dbstfrl.v
 *
 * Controller for double-buffering, slower transmitter case (lazy algo)
 *  ( assumes M < N )
 *
 */

module dbstfrl( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                  clkM, clkN, clr,
                  M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, TEM, TSelM, zN, REN, RSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   wire [width-1:0] f0, e0, x0, y0;
   wire [width-1:0] divst, modst;

   assign x0 = 1;
   assign f0 = -(PM+SN);

   commoddiv #(width) compe0y0st( divst, modst, , f0, N );
   forme0    #(width) compe0st( e0, modst, N, f0[width-1] );
   formy0    #(width) compy0st( y0, divst, f0[width-1] );

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdbstl #(width) schM( TEM, TSelM, clkM, cntM, M, N, e0, y0 );
   schdbfrl #(width) schN( REN, RSelN, clkN, cntN, M, N, e0, x0 );

endmodule
```

```
/* dbftsrl.v
 *
 * Controller for double-buffering, slower receiver case (lazy algo)
 *  ( assumes M < N )
 *
 */

module dbftsrl( zM, cntM, REM, RSelM, zN, cntN, TEN, TSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, REM, RSelM, zN, TEN, TSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   wire [width-1:0] f0, e0, x0, y0;
   wire [width-1:0] divst, modst;

   assign x0 = 1;
   assign f0 = CN-HM;

   commoddiv #(width) compe0y0st( divst, modst, , f0, N );
   forme0    #(width) compe0st( e0, modst, N, f0[width-1] );
   formy0    #(width) compy0st( y0, divst, f0[width-1] );

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdbsrl #(width) schM( REM, RSelM, clkM, cntM, M, N, e0, y0 );
   schdbftl #(width) schN( TEN, TSelN, clkN, cntN, M, N, e0, x0 );

endmodule


/* dbstfrg.v
 *
 * Controller for double-buffering, slower transmitter case (greedy algo)
 *  ( assumes M < N )
 *
 */

module dbstfrg( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, TEM, TSelM, zN, REN, RSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
```

```
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   dbstfrl #(width) schMN( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
                           clkM, clkN, clr,
                           M, N, CM, PM, SM, HM, CN, PN, SN, HN );
endmodule


/* dbftsrg.v
 *
 * Controller for double-buffering, slower receiver case (greedy algo)
 *  ( assumes M < N )
 *
 */

module dbftsrg( zM, cntM, REM, RSelM, zN, cntN, TEN, TSelN,
                clkM, clkN, clr,
                M, N, CM, PM, SM, HM, CN, PN, SN, HN );

   parameter width = 5;

   output zM, REM, RSelM, zN, TEN, TSelN;
   output [width-1:0] cntM, cntN;
   input  clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;

   wire [width-1:0] f0, e0, adje0, x0, y0, adjy0;
   wire [width-1:0] divsr, modsr;

   assign x0 = 1;
   assign f0 = SM+PN;

   commoddiv #(width) compe0y0sr( divsr, modsr, , f0, N );
   forme0    #(width) compe0sr( e0, modsr, N, f0[width-1] );
   adjuste0  #(width) adjuste0( adje0, e0, N );

   formy0    #(width) compy0sr( y0, divsr, f0[width-1] );
   assign    adjy0 = (adje0 == 0) ? (y0-1) : y0;

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdbsrg #(width) schM( REM, RSelM, clkM, cntM, M, N, adje0, adjy0 );
   schdbftg #(width) schN( TEN, TSelN, clkN, cntN, M, N, adje0, x0 );

endmodule


/* gbstfr.v
 *
 * Controller for generalized scheduling, slower transmitter case
 *  ( assumes M < N )
 *
 */
```

```
module gbstfr( zM, cntM, TEM, TSelM, zN, cntN, REN, RSelN,
               clkS, clkM, clkN, clr,
               M, N, CM, PM, SM, HM, CN, PN, SN, HN,
               SB );

   parameter width = 5;

   output zM, TEM, TSelM, zN, REN, RSelN;
   output  [width-1:0] cntM, cntN;
   input  clkS, clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;
   input SB;

   wire [width-1:0] f0, e0, x0, y0, DMN, negDMN, e0M, emax;
   wire [width-1:0] divst, modst;
   wire donee0M, doneDMN;

   seqmod #(width) cDMN( DMN, doneDMN, clkS, N, M, clr );
   neg    #(width) cnDMN( negDMN, DMN );

   assign x0 = 1;
   assign f0 = -(PM+SN);

   commoddiv #(width) compe0y0st( divst, modst, , f0, N );
   forme0    #(width) compe0st( e0, modst, N, f0[width-1] );
   formy0    #(width) compy0st( y0, divst, f0[width-1] );

   seqmod  #(width) ce0Mst( e0M, donee0M, clkS, e0, M, clr );

   assign emax = N - ((PM-CM) + (SN+HN));

   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdgst #(width) schM( TEM, TSelM, clkM, cntM,
                          M, negDMN, e0M, y0, emax, SB );
   schdgfr #(width) schN( REN, RSelN, clkN, cntN,
                          M, N, e0, x0, emax, SB );
endmodule


/* gbftsr.v
 *
 * Controller for generalized scheduling, slower receiver case (greedy algo)
 *  ( assumes M < N )
 *
 */

module gbftsr( zM, cntM, REM, RSelM, zN, cntN, TEN, TSelN,
             clkS, clkM, clkN, clr,
             M, N, CM, PM, SM, HM, CN, PN, SN, HN,
             SB );

   parameter width = 5;
```

```
   output zM, REM, RSelM, zN, TEN, TSelN;
   output  [width-1:0] cntM, cntN;
   input  clkS, clkM, clkN, clr;
   input [width-1:0] M, N, CM, PM, SM, HM, CN, PN, SN, HN;
   input SB;

   wire [width-1:0] f0, e0, adje0, x0, y0, adjy0;
   wire [width-1:0] DMN, negDMN, e0M, adje0M, emin;
   wire [width-1:0] divsr, modsr;
   wire donee0M, doneDMN;

   seqmod #(width) cDMN( DMN, doneDMN, clkS, N, M, clr );
   neg    #(width) cnDMN( negDMN, DMN );

   assign x0 = 1;
   assign f0 = SM+PN;

   commoddiv #(width) compe0y0sr( divsr, modsr, , f0, N );
   forme0    #(width) compe0sr( e0, modsr, N, f0[width-1] );
   adjuste0  #(width) adjuste0( adje0, e0, N );

   formy0    #(width) compy0sr( y0, divsr, f0[width-1] );
   assign    adjy0 = (adje0 == 0) ? (y0-1) : y0;

   seqmod  #(width) ce0Msr( e0M, donee0M, clkS, adje0, M, clr );
   adjuste0M #(width) adjuste0M( adje0M, e0M, M );

   assign emin = M + ((PN-CN) + (SM+HM)) - N;


   fdiv  #(width) fdivM( zM, cntM, clkM, M, 1'b0, clr );
   fdiv  #(width) fdivN( zN, cntN, clkN, N, 1'b0, clr );

   schdgsr #(width) schM( REM, RSelM, clkM, cntM,
                          M, negDMN, adje0M, adjy0, emin, SB );
   schdgft #(width) schN( TEN, TSelN, clkN, cntN,
                          M, N, adje0, x0, emin, SB );
endmodule


/* dbout.v
 *
 * Double-bufferd output hardware
 *
 * This module contains two transmit registers a selecting demux.
 * It takes the transmitter's C and P delays as parameters
 * and passes them to txreg.
 *
 */

module dbout( q, clk, d, TE, TSel, RSel );

   parameter width = 1;
   parameter C = 0;
   parameter P = 0;
```

```
   output [width-1:0] q;
   input  clk;
   input [width-1:0] d;
   input TE, TSel, RSel;

   wire en0, en1;
   wire [width-1:0] q0, q1;
   wire dlyRSel;

   txreg #(width,C,P) r0( q0, clk, d, en0 );
   txreg #(width,C,P) r1( q1, clk, d, en1 );
   demux1to2 seldemux( en0, en1, TE, TSel );

   mux1of2 #(width) qmux( q, q0, q1, RSel );

endmodule


/* rxregM.v
 *
 * Receive Register with Enable and specifiable S and H times
 * <<< For System M >>>
 *
 * Note: S and H times are specified by modifying params.vh
 *
 */

module rxregM( q, clk, d, en );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en;

   reg [width-1:0] q;
   wire [width-1:0] q1;
   reg  flag;

`include "params.vh"

   specify
      specparam S = (`SM*100), H = (`HM*100);

      $setuphold( posedge clk &&& en, d, S, H, flag );
   endspecify

   always @(posedge clk)
   begin
      if (en)
         q = d;
   end
```

```
   always @(flag) q = q + 1'bx;

endmodule


/* rxregN.v
 *
 * Receive Register with Enable and specifiable S and H times
 * <<< For System N >>>
 *
 * Note: S and H times are specified by modifying params.vh
 *
 */

module rxregN( q, clk, d, en );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en;

   reg [width-1:0] q;
   wire [width-1:0] q1;
   reg  flag;

`include "params.vh"

   specify
      specparam S = (`SN*100), H = (`HN*100);

      $setuphold( posedge clk &&& en, d, S, H, flag );
   endspecify

   always @(posedge clk)
   begin
      if (en)
         q = d;
   end

   always @(flag) q = q + 1'bx;

endmodule
```

D.3 Scheduler Components

```
/* schdbfrg.v
 *
 * Scheduling hardware for double-buffering, faster receiver, greedy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
```

```
 */

module schdbfrg( RE, RSel, clk, cnt, M, N, e0, x0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0;

   schdbfrl #(width) sch( RE, RSel, clk, cnt, M, N, e0, x0 );

endmodule


/* schdbfrl.v
 *
 * Scheduling hardware for double-buffering, faster receiver, lazy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbfrl( RE, RSel, clk, cnt, M, N, e0, x0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0;

   wire ld;
   wire [width-1:0] x0_1;

   assign x0_1 = x0 - 1;

   ldgen #(width) ldgenN( ld, cnt, x0_1, N );
   egendb #(width) egenN( RE, , clk, ld, M, N, e0, 1'b0 );
   rgendb rgenN( RSel, clk, RE, ld );

endmodule


/* schdbftg.v
 *
 * Scheduling hardware for double-buffering, faster transmitter, greedy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbftg( TE, TSel, clk, cnt, M, N, adje0, x0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
```

```
   input [width-1:0] M, N, cnt, adje0, x0;

   wire ld;
   wire [width-1:0] x0_2;

   assign x0_2 = x0 - 2;

   ldgen #(width) ldgenN( ld, cnt, x0_2, N );
   egendbg #(width) egenN( TE, , clk, ld, M, N, adje0, 1'b0 );
   rgendb rgenN( TSel, clk, TE, ld );

endmodule


/* schdbftl.v
 *
 * Scheduling hardware for double-buffering, faster transmitter, lazy algo
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdbftl( TE, TSel, clk, cnt, M, N, e0, x0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0;

   schdbfrl #(width) sch( TE, TSel, clk, cnt, M, N, e0, x0 );

endmodule


/* schdbsrg.v
 *
 * Scheduling hardware for double-buffering, slower receiver, greedy algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbsrg( RE, RSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   schdbstl #(width) sch( RE, RSel, clk, cnt, M, N, e0, y0 );

endmodule


/* schdbsrl.v
```

```
 *
 * Scheduling hardware for double-buffering, slower receiver, lazy  algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbsrl( RE, RSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   schdbstl #(width) sch( RE, RSel, clk, cnt, M, N, e0, y0 );

endmodule


/* schdbstg.v
 *
 * Scheduling hardware for double-buffering, slower transmitter, greedy algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbstg( TE, TSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   schdbstl #(width) sch ( TE, TSel, clk, cnt, M, N, e0, y0 );

endmodule


/* schdbstl.v
 *
 * Scheduling hardware for double-buffering, slower transmitter, lazy algo
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdbstl( TE, TSel, clk, cnt, M, N, e0, y0 );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, y0;

   wire TE;
```

```
   wire ld;
   wire [width-1:0] y0_1;

   assign TE = 1'b1;
   assign y0_1 = y0 - 1;

   ldgen #(width) ldgenM( ld, cnt, y0_1, M );
   rgendb rgenM( TSel, clk, TE, ld );

endmodule


/* schdgfr.v
 *
 * Scheduling hardware for generalized scheduling, faster receiver
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdgfr( RE, RSel, clk, cnt, M, N, e0, x0, emax, SB );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, N, cnt, e0, x0, emax;
   input  SB;

   wire ld;
   wire [width-1:0] x0_1;
   wire preRE;
   wire [width-1:0] e;

   assign x0_1 = x0 - 1;

   ldgen #(width) ldgenN( ld, cnt, x0_1, N );

   egendb #(width) egenN( preRE, e, clk, ld, M, N, e0, 1'b0 );
   rgengbst #(width) rgenN( RSel, clk, preRE, ld, e, emax );

   nand nand1( noregs, SB, RSel );
   and  and1( RE, preRE, noregs );

endmodule


/* schdgft.v
 *
 * Scheduling hardware for generalized scheduling, faster transmitter
 *  ( assumes M < N, and cnt is the N counter )
 *
 */

module schdgft( TE, TSel, clk, cnt, M, N, adje0, x0, emin, SB );
```

```
   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, N, cnt, adje0, x0, emin;
   input  SB;

   wire ld;
   wire [width-1:0] x0_2;
   wire preTE;
   wire [width-1:0] e;

   assign x0_2 = x0 - 2;

   ldgen #(width) ldgenN( ld, cnt, x0_2, N );

   egendbg  #(width) egenN( preTE, e, clk, ld, M, N, adje0, 1'b0 );
   rgengbsr #(width) rgenN( TSel, clk, preTE, ld, e, emin );

   nand nand1( noregs, SB, TSel );
   and  and1( TE, preTE, noregs );

endmodule


/* schdgsr.v
 *
 * Scheduling hardware for generalized scheduling, slower receiver
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdgsr( RE, RSel, clk, cnt, M, negDMN, adje0M, y0, emin, SB );

   parameter width = 1;

   output RE, RSel;
   input  clk;
   input [width-1:0] M, negDMN, cnt, adje0M, y0, emin;
   input  SB;

   wire preRE;
   wire ld;
   wire [width-1:0] y0_1;
   wire [width-1:0] negM, e;

   assign preRE = 1'b1;
   assign y0_1 = y0 - 1;

   assign negM = -M;

   ldgen #(width) ldgenN( ld, cnt, y0_1, M );

   egendbg #(width) egenM( , e, clk, ld, negDMN, negM, adje0M, 1'b1 );
   rgengbsr #(width) rgenM( RSel, clk, preRE, ld, e, emin );
```

```
   nand nand1( noregs, SB, RSel );
   and  and1( RE, preRE, noregs );

endmodule


/* schdgst.v
 *
 * Scheduling hardware for generalized scheduling, slower transmitter
 *  ( assumes M < N, and cnt is the M counter )
 *
 */

module schdgst( TE, TSel, clk, cnt, M, negDMN, e0M, y0, emax, SB );

   parameter width = 1;

   output TE, TSel;
   input  clk;
   input [width-1:0] M, negDMN, cnt, e0M, y0, emax;
   input  SB;

   wire preTE;
   wire ld;
   wire [width-1:0] y0_1;
   wire [width-1:0] negM, e;

   assign preTE = 1'b1;
   assign y0_1 = y0 - 1;

   assign negM = -M;

   ldgen #(width) ldgenN( ld, cnt, y0_1, M );

   egendb #(width) egenN( , e, clk, ld, negDMN, negM, e0M, 1'b1 );
   rgengbst #(width) rgenN( TSel, clk, preTE, ld, e, emax );

   nand nand1( noregs, SB, TSel );
   and  and1( TE, preTE, noregs );

endmodule


/* egendb.v
 *
 * The EGen module, slower transmitter case
 *
 * This module works as the EGen module for double-buffering when gb is 0,
 * and works as the modified EGen module in the slower system's
 * generalized schedling hardware for the slower transmitter case.
 *
 */

module egendb( E, eout, clk, ld, M, N, e0, gb );
```

```
   parameter width = 1;

   output E;
   output [width-1:0] eout;
   input clk;
   input ld;
   input [width-1:0] M, N, e0;
   input gb;

   wire E;
   wire [width-1:0] eout, negN, eold, e, eN, nxteold;
   wire sgne;
   wire muxsel;

   assign negN = -N;

   muxreg #(width) eoldreg( eold, clk, nxteold, e0, ld );
   adder  #(width) eoplusM( e, eold, M, 1'b0 );
   adder #(width) esubN( eN, e, negN, 1'b0 );

   assign sgne = e[width-1];
   assign E = ~sgne;

   xor x1( muxsel, sgne, gb );
   mux1of2 #(width) neomux( nxteold, eN, e, muxsel );

   mux1of2 #(width) eoutmux( eout, e, eold, gb );

endmodule


/* egendbg.v
 *
 * The EGen module for double-buffering greedy slower receiver algo
 *
 */

module egendbg( E, eout, clk, ld, M, N, e0, gb );

   parameter width = 1;

   output E;
   output [width-1:0] eout;
   input clk;
   input ld;
   input [width-1:0] M, N, e0;
   input gb;

   wire E, notE;
   wire [width-1:0] eout, negN, eold, e, eN, nxteold;
   wire sgne, z;
   wire muxsel;

   assign negN = -N;
```

```
   muxreg #(width) eoldreg( eold, clk, nxteold, e0, ld );
   adder  #(width) eoplusM( e, eold, M, 1'b0 );
   adder #(width)  esubN( eN, e, negN, 1'b0 );
   zero   #(width) ez( z, e );

   assign sgne = e[width-1];

   nor    gtz( E, z, sgne );
   assign notE = ~E;

   xor x1( muxsel, notE, gb );
   mux1of2 #(width) neomux( nxteold, eN, e, muxsel );

   mux1of2 #(width) eoutmux( eout, e, eold, gb );

endmodule


/* ldgen.v
 *
 * LD Gen module
 *   (checks if count == (i-1) mod M and generates LD signal)
 *
 */

module ldgen( ld, count, i, M );

   parameter width = 1;

   output ld;
   input  [width-1:0] count, i, M;

   wire   [width-1:0] i_1, a1, iMo;

   assign a1 = 1;

   addsub #(width) sub1( i_1, i, a1, 1'b1 );
   commod #(width) domod( iMo, i_1, M );

   assign ld = ( count == iMo );

endmodule


/* rgendb.v
 *
 * The RGen module for double-buffering
 *
 */

module rgendb( r, clk, en, ld );

   output r;
   input clk;
```

```
   input en, ld;

   tflopenld rtflop( r, clk, 1'b0, en, ld );

endmodule


/* rgengbsr.v
 *
 * The RGen module for generalized run-time scheduling, slower receiver case
 *
 */

module rgengbsr( r, clk, en, ld, e, emin );

   parameter width = 1;

   output r;
   input clk;
   input en, ld;
   input [width-1:0] e, emin;

   wire ge, q;

   leq #(width) egeqemin( ge, emin, e );
   nor nr1( r, ge, q );
   regencs rreg( q, clk, r, en, 1'b0, ld );

endmodule


/* rgengbst.v
 *
 * The RGen module for generalized scheduling, slower transmitter case
 *
 */

module rgengbst( r, clk, en, ld, e, emax );

   parameter width = 1;

   output r;
   input clk;
   input en, ld;
   input [width-1:0] e, emax;

   wire le, d;

   leq #(width) eleqemax( le, e, emax );
   nor nr1( d, le, r );
   regencs rreg( r, clk, d, en, ld, 1'b0 );

endmodule
```

D.4 Bootup Computation Circuitry

```
/* commod.v
 *
 * combinational mod module
 *
 * This module computes:
 *       mod = (a mod b) for  -3b < a < 3b
 *
 */

module commod( modout, a, b );

   parameter width = 1;

   output [width-1:0] modout;
   input  [width-1:0] a, b;

   wire [width-1:0] mod, modprime;

   commoddiv #(width) cmd( , mod, modprime, a, b );
   mux1of2 #(width) outmux( modout, mod, modprime, a[width-1] );

endmodule


/* commoddiv.v
 *
 * combinational mod / div module -- cascaded twice
 *
 * (see moddiv.v for more details)
 *
 */

module commoddiv( div, mod, modprime, a, b );

   parameter width = 1;

   output [width-1:0] div, mod, modprime;
   input  [width-1:0] a, b;

   wire [width-1:0] nxtmod, nxtdiv, div0;

   assign div0 = 0;

   moddiv #(width) m1( nxtdiv, nxtmod, , , a, b, div0 );
   moddiv #(width) m2( div, mod, modprime, , nxtmod, b, nxtdiv );

endmodule


/* moddiv.v
 *
 * mod / div module
 *
```

```
 * This module computes:
 *     if a >= 0
 *        mod = (a mod b)
 *        div = (a div b)
 *     if a < 0
 *        mod = (a mod b) - b
 *        mod' = (a mod b)
 *        div = -(a div b)         if (a mod b != 0)
 *            = -(a div b) - 1     if (a mod b == 0)
 *
 * This module can be cascaded by connecting mod to a and div to divin.
 * sgnchg would then indicate when the operation is done.
 *
 */

module moddiv( div, mod, modprime, sgnchg, a, b, divin );

   parameter width = 1;

   output [width-1:0] div, mod, modprime;
   output sgnchg;
   input  [width-1:0] a, b, divin;

   wire [width-1:0] div, mod, modprime;
   wire sgna, nsgna, sgnchg;

   assign sgna = a[width-1];
   assign nsgna = ~sgna;

   addsub #(width) add1( modprime, a, b, nsgna );
   xor x1( sgnchg, modprime[width-1], sgna );
   mux1of2 #(width) modmux( mod, modprime, a, sgnchg );
   incinh  #(width) divinc( div, divin, sgnchg );

endmodule


/* seqmod.v
 *
 * sequential mod module
 *
 * This module computes:
 *        mod = (a mod b)
 *
 */

module seqmod( modout, done, clk, a, b, start );

   parameter width = 1;

   output [width-1:0] modout;
   output done;
   input clk;
   input  [width-1:0] a, b;
   input start;
```

```
   wire [width-1:0] mod, modprime, nxtmod;

   seqmoddiv #(width) smd( , mod, modprime, done, clk, a, b, start );
   mux1of2 #(width) outmux( modout, mod, modprime, a[width-1] );

endmodule


/* seqmoddiv.v
 *
 * sequential mod / div module
 *
 * (see moddiv.v)
 *
 */

module seqmoddiv( div, mod, modprime, done, clk, a, b, start );

   parameter width = 1;

   output [width-1:0] div, mod, modprime;
   output done;
   input clk;
   input  [width-1:0] a, b;
   input start;

   wire [width-1:0] div, mod, nxtmod, nxtdiv;

   regencs #(width) curdiv( div, clk, nxtdiv, 1'b1, start, 1'b0 );
   muxreg #(width) curmod( mod, clk, nxtmod, a, start );
   moddiv #(width) compute( nxtdiv, nxtmod, modprime, done, mod, b, div );

endmodule


/* adjuste0.v
 *
 * Adjust module for e0 for greedy slower receiver algo
 *
 */

module adjuste0( newe0, e0, N );

   parameter width = 1;

   output [width-1:0] newe0;
   input  [width-1:0] e0, N;

   wire   [width-1:0] eplusN;
   wire   z;

   adder #(width) ein( eplusN, e0, N, 1'b0 );
   zero  #(width) zcheck( z, eplusN );
   mux1of2 #(width) emux( newe0, e0, eplusN, z );
```

```
endmodule


/* adjuste0M.v
 *
 * Adjust module for e0 mod M for generalized slower receiver algo
 *
 */

module adjuste0M( newe0M, e0M, M );

   parameter width = 1;

   output [width-1:0] newe0M;
   input  [width-1:0] e0M, M;

   wire   [width-1:0] eplusM;
   wire   z;

   adder #(width) ein( eplusM, e0M, M, 1'b0 );
   zero  #(width) zcheck( z, e0M );
   mux1of2 #(width) emux( newe0M, e0M, eplusM, z );

endmodule


/* forme0.v
 *
 * Module to form e0 = (f0 mod N) - N
 *    from the output of the mod/div module
 *
 */

module forme0( e0, modin, N, sgnf0 );

   parameter width = 1;

   output [width-1:0] e0;
   input  [width-1:0] modin, N;
   input  sgnf0;

   wire [width-1:0] e0, eN;

   addsub #(width) add1( eN, modin, N, 1'b1 );
   mux1of2 #(width) outmux( e0, eN, modin, sgnf0 );

endmodule


/* formy0.v
 *
 * Module to form y0
 *    from the output of the mod/div module
 *
```

```
 */

module formy0( y0, divin, sgnf0 );

   parameter width = 1;

   output [width-1:0] y0;
   input  [width-1:0] divin;
   input  sgnf0;

   wire nsgnf0;
   wire [width-1:0] zor1;

   assign nsgnf0 = ~sgnf0;

   assign zor1 = 0 | nsgnf0;

   addsub #(width) add1( y0, zor1, divin, sgnf0 );

endmodule
```

D.5 Basic Components

```
/* adder.v
 *
 * variable width adder with cin
 *
 */

module adder( sum, a, b, cin );

   parameter width = 1;

   output [width-1:0] sum;
   input  [width-1:0] a, b;
   input  cin;

   assign sum = a + b + cin;

endmodule


/* addsub.v
 *
 * variable width adder/subtractor with carry-in
 *
 */

module addsub( sum, a, b, sub );

   parameter width = 1;
```

```
   output [width-1:0] sum;
   input  [width-1:0] a, b;
   input  sub;

   assign sum = sub ? (a-b) : (a+b);

endmodule


/* counten.v
 *
 * countup timer with enable, Asynchronous clear
 *
 */

module counten( z, q, clk, en, clr );

   parameter width = 1;

   output z;
   output [width-1:0] q;
   input clk;
   input en, clr;

   reg [width-1:0] q;

   assign z = (q == 0);

   always @(clr)
      if (clr)
         q = 0;

   always @(posedge clk)
      if (clr)
         q = 0;
      else
         if (en)
               q = q + 1;

endmodule


/* countup.v
 *
 * countup timer with terminal count (tc), Asynchronous clear and set (to tc)
 *
 */

module countup( z, q, clk, tc, clr, set );

   parameter width = 1;

   output z;
   output [width-1:0] q;
   input clk;
```

```
   input [width-1:0] tc;
   input clr, set;

   reg [width-1:0] q;

   assign z = (q == 0);

   always @(clr or set)
      if (clr)
         q = 0;
      else
         if (set)
            q = tc;

   always @(posedge clk)
      if (clr)
         q = 0;
      else
         if (set)
            q = tc;
         else
            q = (q == tc) ? 0 : (q + 1);

endmodule


/* demux1to2.v
 *
 * 1-to-2 demux
 *
 */

module demux1to2( q0, q1, d, sel );

   parameter width = 1;

   output [width-1:0] q0, q1;
   input [width-1:0] d;
   input sel;

   assign q0 = sel ? 0 : d;
   assign q1 = sel ? d : 0;

endmodule


/* fdiv.v
 *
 * Frequency divider, divides by M
 *
 * (Note: fdiv currently doesn't work for M = 1)
 *
 */

module fdiv( clkout, q, clkin, M, clr, set );
```

```
   parameter width = 1;

   output clkout;
   output [width-1:0] q;
   input clkin;
   input [width-1:0] M;
   input clr, set;

   wire z;
   wire [width-1:0] tc;

   wire  clkout;

   assign tc = M-1;

   countup #(width) cnt ( z, q, clkin, tc, clr, set );

   assign clkout = z;

endmodule


/* incinh.v
 *
 * Combinational increment with Inhibit
 *
 */

module incinh( res, a, inh );

   parameter width = 1;

   output [width-1:0] res;
   input  [width-1:0] a;
   input  inh;

   assign res = inh ? a : (a+1);

endmodule


/* leq.v
 *
 * Less-than-or-equal-to comparator
 *   ( checks if a <= b )
 *
 * Note that this version assumes Verilog treats the numbers as
 * unsigned numbers and does its own signed version of leq.
 *
 * Luis F. G. Sarmenta 950409.2128
 *
 */

module leq( res, a, b );
```

```
   parameter width = 1;

   output res;
   input  [width-1:0] a, b;

   assign res = ( a[width-1]
                 ? ( b[width-1]
                      ? ( a >= b )
                      : 1
                   )
                 : ( !b[width-1]
                      ? ( a <= b )
                      : 0
                   )
               );
endmodule


/* mux1of2.v
 *
 * 1-of-2 mux
 *
 */

module mux1of2( q, d0, d1, sel );

   parameter width = 1;
   output [width-1:0] q;
   input [width-1:0] d0, d1;
   input sel;

   assign q = sel ? d1 : d0;

endmodule


/* muxreg.v
 *
 * 1-of-2 mux with storage
 *
 */

module muxreg( q, clk, d0, d1, sel );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d0, d1;
   input sel;

   reg [width-1:0] q;

   always @(posedge clk)
```

```
   begin
      case (sel)
         0 : q = d0;
         1 : q = d1;
      endcase
   end

endmodule


/* neg.v
 *
 * Arithmetic Negator
 *
 */

module neg( nega, a );

   parameter width = 1;

   output [width-1:0] nega;
   input  [width-1:0] a;

   assign nega = -a;

endmodule


/* regen.v
 *
 * Register with Enable
 *
 */

module regen( q, clk, d, en );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      if (en)
         q = d;
   end

endmodule


/* regencs.v
```

```
 *
 * Register with Enable and Synchronous Clear and Set
 * (Clr overrides Set)
 *
 */

module regencs( q, clk, d, en, clr, set );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en, clr, set;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      if (en)
         if (clr)
            q = 0;
         else
            if (set)
               q = 1;
            else
               q = d;
   end

endmodule


/* tflopenld.v
 *
 * a T flip-flop with Enable and Load
 *  ( ld overrides en )
 *
 */

module tflopenld( q, clk, d, en, ld );

   parameter width = 1;

   output [width-1:0] q;
   input clk;
   input [width-1:0] d;
   input en, ld;

   reg [width-1:0] q;

   always @(posedge clk)
   begin
      case (ld)
            0 : q = en ? ~q : q;
            1 : q = d;
```

```
      endcase
   end


endmodule


/* zero.v
 *
 * Zero check
 *
 */

module zero( z, a );

   parameter width = 1;

   output z;
   input  [width-1:0] a;

   assign z = ( a == 0 );

endmodule
```