RTsim
A register transfer simulator

Jimmy Lam

Computer Science Department
California Institute of Technology

5081:TR:83

RTsim
A register transfer simulator

Jimmy Lam
Computer Science Department
California Institute of Technology
Pasadena, California 91125

5081:TR:83

In Partial Fulfillment of the Requirements for the
Degree of Master of Science

April, 1983

Caltech Silicon Structure Project

ACKNOWLEDGEMENTS

I would like to express my gratitude to my parents for their love, encouragement, and support. Special thanks to Rossana for her patience and understanding. I am also indebted to Caltech whose financial support allowed me to obtain such an excellent education in this country.

I would also like to thank my advisor Randy Bryant for his suggestions and the inspiration received reading his great thesis. His computer architecture course introduced me to this project, and served as a test facility for my simulator. I owe special thanks to my fellow students in the course for their helpful suggestions. I am grateful to my friends - Charles Ng and John Ngai - for their discussions, and Mike Newton for proof reading this thesis.

Special thanks also extend to George Lewicki. Without George and his silicon compiler this project would not have been possible.

ABSTRACT

The growing complexity and size of VLSI processors are demanding extremely accurate, yet efficient, simulation facilities for microcode debugging, logic verification, and system integration. However, reliance on mask iterations to remedy problems on a chip not only raises costs, but also extends the design cycle. Simulation justifies itself in both the turn around time and the design cost. Gate level simulation is one method for reducing errors in a chip design. However, gate level simulation of large designs are extremely expensive, and sometimes impossible when the gate level representation is not known. This thesis attempts to solve this problem by providing a functional level approach, consisting of a register transfer description language, an embedded functional modeling language, a reconfigurable assembler, and a functional simulation program. Mixed-level simulation capability is also provided by allowing the replacement of a functional unit by a transistor network which is being simulated by a switch-level logic simulator.

Table of Contents

List of Figures

1. Introduction

RTsim is a register transfer simulator. It models a system as a network of functional units connected by arbitrary integral delay paths. It is designed to simulate MOS circuitry and thus provides the capability to model MOS capacitance. Like other functional simulators, RTsim provides simulation capability for verifying the logical correctness of a system as well as testing out different architectures in a more efficient and versatile way. But unlike other functional simulators, RTsim is especially designed for silicon compilers. Input specifications are separated into two levels.

The register transfer description (RTD) language provides the capability to specify the network model of the system. In this language, a system is modeled as a network of black boxes connected by arbitrary delay paths. Each black box in the system has two input ports, an output port, and a set of attributes called options. The behavior of the black boxes is described in the embedded RTD element description language and communications between the two levels are restricted to the ports and options. The main advantage of this hierarchical approach is to make a clean separation of the two levels. If we only allow the two levels to interact in a restricted way, a given RTD element description can be written based on its functional specification, independent of the behavior of its neighbors and how they are connected.

RTsim also provides mixed-level simulation capability. In typical top down design methodologies, the user is encouraged to try out a design at an abstract functional level. Then the abstract functional blocks can be reduced step by step towards the actual hardware description[5][7]. RTsim allows the user to replace RTD element descriptions by transistor networks. The switch-level logic simulator MOSSIM II[1][1] will be invoked to simulate the transistor networks whenever necessary.

As design complexity grows, a structured design is always preferred, and has a number of advantages. RTsim encourages microcoded architectures in a variety of ways. A functional block is described as a black box with a set of attributes. In addition, a black box can also carry a symbol table specified in the RTD language. With this symbol table and a little help from various pseudo-ops, the RTsim assembler can assemble and load symbolic instructions into binary form. In a typical simulation, two or more symbol tables may exist at the same time, one for microcode, and the other for macro code. The simulator provides a translation mechanism which allows the binary output of a black box to be translated to symbolic form according to a specific symbol table during simulation. This is particularly useful, for example, in interpreting the output of a micro instruction register.

This thesis is divided into sections describing different aspects of the program. Section 2 describes the simulation model. Section 3 describes the register transfer description (RTD) language. Section 4 describes the RTD element (functional unit) description language. Section 5 describes the MOSSIM interface. Section 6 describes possible improvements. Section 7 documents the simulator commands and usage. Section 8 describes various utilities.

RTsim is written in MAINSAIL[8] (TM)[1] and hence can run on a variety of computer systems.

2. Simulation model

2.1. Steady state computation

RTsim uses a unit delay model[3][6] to compute the steady state of the system. It divides a subcycle[2] into as many iterations as necessary, where each iteration represents a unit time. The states of the current iteration are computed as a function of the states of the last iteration and the functional specifications of the system.

The simulator keeps an event list of all the excited functional blocks (events), where each event indicates a perturbation in the network. The new network states are computed from the events in the event list as a function of the old network states and the functional behavior of the system. A perturbation may propagate if new events are generated due to the previously excited functional blocks. The newly generated events are chained to form the new event list. The simulated period between the old and the new event list is defined to be a unit time. Steady state is reached if all perturbations cease, and the simulator is said to proceed for a subcycle. Some sort of timing which triggers the initial network state computation must be introduced to the system. Good candidates for the initial event list are these elements called "clock" units.

2.2. Propagation delay

As suggested by the naming of the model, the propagation delay through a functional unit is always one[3]. In normal conditions, a path has zero delay. Circumstances may require a path to have an arbitrary delay as a remedy for the unit delay model. Instead of introducing dummy functional units along the path, RTsim allows the user to specify a

[1] Use of RTsim & MOSSIM II requires a MAINSAIL run-time license from Xidak, Inc., Sunnyvale, CA.

[2] A subcycle is the period from one steady state to the next.

[3] It takes an unit time to propagate a signal from the input of a functional unit to its output.

positive integer as a path delay number associated with each path. This allows the user to model "pure" delay functional units as well as arbitrary path delays.

2.3. Signal propagation and error detection

Each functional unit has an input data vector, an input control vector and an output data vector with bit values chosen from the set S={0,1,X,Z}, where 0,1,X and Z represent low, high, invalid (between 0 and 1 inclusive) and high impedance (undriven) respectively.

Although RTsim, as viewed by the user, is a 4-state logic simulator, it actually has seven states. In addition to the four states mentioned above, a signal can either be driven by some functional units or held at its previous state by capacitance. The two signal types must be distinguished because interactions between and among the two classes are different. But within a functional unit, whether a signal is charged or driven does not matter. Only its logic will be important.

So, in order to distinguish the two types of signal, the bit values of all input vectors are expanded to include the set C={0c,1c,Xc}. An input bit value can have state in the set of S union C, where S belongs to the force class and C belongs to the charge class[4].

	0	1	X	Z	0c	1c	Xc
0	0	*	X	0	0	0	0
1	*	1	X	1	1	1	1
X	X	X	X	X	X	X	X
Z	0	1	X	Z	0c	1c	Xc
0c	0	1	X	0c	0c	Xc	Xc
1c	0	1	X	1c	Xc	1c	Xc
Xc	0	1	X	Xc	Xc	Xc	Xc

Where * reports an error.

Figure 1 - Table of interactions

[4] Force class and charge class are fancy notation for driven and stored charged signals. When forced signal encounters charged signal, the forced signal will determine the outcome, except for the Z-state.

To implement the unit delay model, the simulator first evaluates all events in the event list. Then, it goes through the event list once more and picks out those functional units which generate new perturbations, i.e., whose output has changed. The input vectors of all functional units of unit delay away from the new perturbations are converted to charge class. The new inputs indicating a perturbation will be collected from their predecessors. Figure 1 tabulates the result when two functional units happen to drive the same input bit. Note that the order of collecting the inputs will be important. For example, (0 1) X connected to the same bit will produce an X and an error report while 0 (1 X) will only produce an X (see command CATCH for X bit detection). After all inputs are collected, the simulator compares the new value against the old one. If they are not equal, the functional unit will be added to the event list for the next iteration.

2.4. MOS capacitance

Associated with the simulator is a variable called "charge-hold-period". MOS capacitance is modeled by this variable. In normal circumstances, the input node of a functional unit is modeled as though it had infinite capacitance. Both the input data vector and the input control vector will hold their previous values even though the input node is not being driven. But if capacitance modeling is enabled and input is not being driven, the input vectors will be set to Z state after N subcycles, where N is the value of "charge-hold-period". In either case, the input vectors will be in the charge class for the extended period.

2.5. Oscillation detection

Thus far, we assume the number of iterations between subcycles is finite before the system comes to steady state. This is not always the case. If we construct a system with behavior resembling a 3-inverter ring, we create a problem. There is no steady state for this system. So, a maximum number of iterations can be set by the user. If the iteration count ever exceeds the maximum, an error will be reported.

3. Register transfer description (RTD) language

The register transfer description specifies the detailed architecture of the system. The RTD language is provided to describe an architecture where functional units are treated as black boxes at the register transfer level. This leads to a number of advantages which make RTsim different from other functional simulators. The most obvious one is the separation of register transfer description and functional unit description. The simulator

needs only to manage signal propagations and resolve drive conflicts. No explicit knowledge about the functional units need be known. The reverse is also true about the functional units. A functional unit, once written, can be put in a library for code sharing since architectural changes have little or no effect on its behavior description. This makes RTsim ideal for silicon compilers.

3.1. The RTD language

The RTD language has a rather simple syntax and has no case distinction. Refer to appendix C for the BNF description. Only three keywords in a predefined order must be present. The first keyword is "INST" which starts the instance[5] declaration section; the second keyword is "CONN" which starts the connection section; the third keyword is "DESP" which denotes the beginning of an internal description of instances. To get a better understanding of the language, one should refer to appendix D when reading the following sections.

3.1.1. Instance declaration

The "INST" section consists of zero or more instance declaration statements with an ISPS like syntax[2]. This type of statement starts with the instance name and follows with

```
Reg[0:7]\reg1<0:31>
    ( # dual port static 32-bit register array
    dualPort,
    static
    );
Clk\clock<|0:2> (ThreePhase);   # 3-phase clock element
Stk\stack[4]<0:15>;        # 16-bit stack of depth 4
Ram\sram<0:20 | 0:15>; # Static RAM, 5-bit address, 16-bit data
```

Figure 2 - Instance declaration statements

[5] we use instance to denote functional unit since functional unit is an image of a functional description. For example: a chip may have many registers but only one description for a register is needed.

an optional array size. For example:

R[0:7]\dreg[3]<0:31 | 0:31> (dynamic,dualPort) ;

It reads as follows: an instance array named "R" with index from 0 to 7 is declared, where every element in the array is an image of a RTD element[6] "dreg" which has a depth of 3, input data vector of index from 0 to 31 and output data vector of index from 0 to 31. The options "dynamic" and "dualPort" are turned on. The optional depth "[3]" is useful for stack type instance, and is shown here for completeness only.

The above example has all the options of an instance declaration statement. An instance need not be an array. Hence "[0:7]" is optional. An instance may not have data input or output. So, either "0:31" is optional. Refer to figure 2 for examples.

An instance declaration statement does not contain any information about the input control vector explicitly. Based on descriptions given for the input and output data vector, the functional unit is expected to deduce all relevant data about the input control vector, with an additional assumption that the input control vector is always indexed from 0. In the "R[0:7]" example above, a "dreg" element has 5 control inputs which are "refresh", "ld_bus_A", "ld_bus_B", "rd_bus_A", and "rd_bus_B" respectively. Also refer to appendix A.

A number can be represented in four ways in the RTD language. The usual decimal notation is the default; a binary number must be preceded by '; an octal number must be preceded by ~ and a hex number preceded by $.

3.1.2. Connection declaration

The "CONN" section consists of zero or more connection declaration statements. There are basically two types of connection statements. The unidirectional connection statement consists of a group of vectors separated by an = sign. When the ".in", ".out", and ".ctrl" attribute is appended to any instance, it denotes its corresponding input, output and control vector. A unidirectional connection statement begins with an output vector and follows with one or more input or control vectors. For example:

reg.out<0:4> =2 mdl.in<5:9> = ALU.ctrl;

All vectors in the statement will be connected together in a one to many fashion. The direction of data flow is always from the first vector to the rest of the vectors. Notice the bits specification is omitted for "ALU.ctrl". The simulator will be intelligent enough to fill in the first 5 bits. The number following the = sign denotes the path delay between vectors.

The default is 0 which denotes a zero path delay. For the above example, a perturbation in the output of "ctrlStore" will take 2 iterations (time units) before it arrives to the data input of "mdl", while no time is needed before it arrives to the control input of "ALU".

The bidirectional connection statements consist of a group of instances separated by the = sign. For example:

bus<0:31> = regArr = PC = IR<0:31> = Addreg;

The first instance in the statement must be a "biLink" type element (See section 4). The rest of the instances can be either an instance array or just an instance. If an instance is declared as an array, the whole array must be used in a bidirectional statement, i.e., regArr[1] cannot be used, but regArr can. A bidirectional statement can be represented as a set of unidirectional statements. For example:

Bidirectional statement:

biLinkInsName = insName1 = insName2 = ... ;

can be represented by:

biLinkInsName.out = insName1.in = insName2.in = ... ;
insName1.out = biLinkInsName.in;
insName2.out = biLinkInsName.in;
...

No delay number is allowed in a bidirectional statement.

3.1.3. Description declaration

The "DESP" section consists of zero or more description declaration statements. Only descriptions for memory type instances are allowed. A description declaration statement starts with an instance name followed by a block of case statements. It is used to define the symbol table for use by the RTsim assembler. Refer to figure 3 and appendix D for examples.

The basic entry in the case statement is of the form

NUMBER = STRING;

Where "STRING" is the symbolic representation of "NUMBER", the strings "DEF", "ABSOLUTE", "RELATIVE", and "LITERAL" have special meanings and will be explained in the following section.

---

[6] A MAINSAIL module contains the functional description.

The symbol table is divided into groups with group numbers starting from 1, where each group represents a specific instruction format. A group consists of all the fields[7] for a word of the symbolic code. In practice, an architecture may have a highly vertical micro instruction format with the first field of the word denoting different groups of micro instruction.

3.2. RTsim assembler

An assembler is provided to free the user from manipulating the machine instructions directly. This assembler has no symbol table with which to start. Its symbol table must be loaded before the actual assembly process can proceed. Refer to appendix E and F for examples. This assembler has a set of simple rules.

```
Memory
begin
        case out<23:25> in group 1 of
          begin                          # A source
          0 = DEF;                       # default,nop
          1 = PC;                        # program counter
          3 = T2;                        # temporary register
          4 = T1;                        # temporary register
          5 = Rx;                        # use Rx
          6 = Ry;                        # use Ry
        end;
        case out<20:22> in group 1 of
          [0-7] = R[0-7];                # register array
        ...
        ...
end;
```

Figure 3 - Symbol table definition

[7] The bits <23:25> in figure 3 is a field for group 1 instructions.

Instead of the usual notation, all label definitions must be preceded by a colon(:) to distinguish them from keywords. A period(.) is used to start a pseudo-op. Currently only four pseudo-ops have been implemented. ".Load x" loads the symbol table of instance x. It also tells the assembler where to put the code after it is done. This pseudo-op must be used before any code can be assembled. Refer to appendix D for how to define the symbol table. ".Loc n" will instruct the assembler to start assembling the code at location n in the memory and default is 0. ".Word n" will put number n into the current location of memory. Four types of numbers are supported. The default is decimal. Binary number, octal number and hex number should be preceded by ', ~ and $ respectively. ".Align b" sets the current location to a number which contains the bit pattern b and is greater than or equal to its previous value. When specified in binary form, b can be a contiguous bit pattern with optional don't care bits (X) at both ends. The # is the comment character. The rest of the line following a # will be thrown away. For example:

```
        .Load ROM     # load table of instance ROM
        .Loc ~23      # start assemble code at octal 23
:Loop   Jump Loop
        .Word $45     # put hex 45 in this location
```

The symbol table is divided into groups with each group corresponding to one instruction format. Fields of different groups may be overlapped. For example:

```
          Jump R1     # this belongs to Jump group
          Add R1 R2   # this belongs to ALU group
:Addr     nop
```

where the binary representation of "Jump" may be 5 bits wide starting at bit 23, and "Add" is four bits wide starting at bit 24. The presence of "Jump" or "Add" makes the interpretation of its following fields different. While they may both be "R1", the bit patterns they represented may be different since they belong to different instruction groups. So, instruction groups must be clearly defined. A group consists of all the fields for a word of the symbolic code. The first keyword found on a line determines the current group for this symbolic instruction. If a field is missing in the code, the assembler will search for the default keyword in the symbol table. Hence, the default must be defined when building up the symbol table. The default for all fields is denoted by the string "DEF". If the default keyword is not found, the corresponding field in the code will be filled with zeros. All fields must be defined in the RTD file even though no keyword is needed for a particular field. For the current implementation, a field can be at most 31 bits wide.

An address field may be defined using the string "ABSOLUTE", or "RELATIVE". For example:

```
case out<26:32> in group 1 of
  begin                          # jump address field
  0 = ABSOLUTE;
  end;

case out<26:32> in group 1 of
  begin                          # jump address field
  2 = RELATIVE;
  end;
```

This will allocate a seven bits address field starting at bit 26 for group 1 instruction. The difference between an "ABSOLUTE" address field and a "RELATIVE" address field lies in the computation of the address of the label. An absolute address field computes the address of a label as its actual address in memory; while a relative address field computes the address as the difference between the address of label in memory and the location of the current word in memory minus the number. That is:

```
absolute address = label address
relative address = label address - pc - num
```

where "pc" is the current word the assembler is working on, "num" is the number before the string "RELATIVE", i.e., two in this case. Note: for absolute address field, "num" must be zero.

A literal field may be defined using the string "LITERAL". For example:

```
case out<26:32> in group 2 of
  begin                          # literal field
  0 = LITERAL;
  end;
```

This will allocate a seven bits literal field for group 2 instruction. A literal field is used to put a number into a field without going through symbol table translation.

4. RTD element description language

RTD elements are MAINSAIL modules which model the behavior of the corresponding hardware components. The user is encouraged to verify the representations against logic design by using the MOSSIM interface.

4.1. Element design guide

The following sections outline the method for designing an RTD element. The author believes it is best to accompany explanation by examples. Please refer to appendix A and B when trying to understand the following sections. For those who do not know MAINSAIL, please consult the MAINSAIL language manual.

4.1.1. Element declaration

The file "rtsim.h" which resides in the RTsim directory, contains definitions for the element description language. It must be included as a "sourcefile" in MAINSAIL code. A MAINSAIL module can be declared as an RTD element by using the macro "element(modName)" instead of the usual module declaration. For example:

```
begin "dreg"
sourcefile "<RTsim directory>rtsim.h";
element(dreg);
 ...
 ...
end "dreg";
```

This will declare module "dreg" as an RTD element. An RTD element will have the following variables automatically defined:

```
integer noInBase,noOutBase;
sVec excitPtr;
long bits attribute,phase,clockBits;
string lstFile;
pointer(textFile) fp;
lBits ic,cc,oc;
lBits array(0 to *) bitArr;
lBits id,ix,cd,cx,od,ox;
integer noOfIn,noOfCtrl,noOfOut,depth;
string array(1 to *) optArr;
boolean array(1 to *) optVal;
procedure setup;
procedure signal;
procedure display;
procedure setIntern;
```

The first six rows of variables will be used by the simulator for housekeeping purposes. "bitArr" is a variable length bits (lBits) array. This array will be used by memory type instance to model the physical memory. The size of the array must be indicated by using

the integer variable "depth". The size of the array as computed and allocated by the RTsim assembler will be 2**"depth". Refer to appendix B for an example. The next row consists of two input and one output vectors, where "id", "ix" together are the input data vector; "cd", "cx" are the input control vector; and "od", "ox" constitute the output data vector. Every bit of the vector can take on four values in the set S={0,1,X,Z}[8] and its internal representation is shown in figure 4. "noOfIn", "noOfCtrl", and "noOfOut" denote the widths of the input data vector, input control vector, and output data vector respectively. "noOfIn" and "noOfOut" will be set by the simulator from information in the RTD file and should not be changed by the RTD element. "noOfCtrl" should be set by the RTD element since the number of control bits could always be deduced from various information about the input vector and the functional behavior of the element.

Variable "depth" can be set either by the RTD element or by the simulator, depending upon its usage. For example, in

```
Stk\stack[10]<0:15>;
```

"depth" will be set to 10 by the simulator based upon the RTD syntax, but the actual interpretation of the variable will depend on the RTD element. However, in this case, we will undoubtedly interpret it as the depth of stack. If a particular RTD element carries a symbol table, i.e. a memory type element, the RTsim assembler will use the variable "depth" to compute its size.

	d-bit	x-bit
0	0	1
1	1	1
X	1	0
Z	0	0

Figure 4 - Internal representation of bit value

[8] Refer to section 2.3 for meanings of 0, 1, X, and Z.

"optArr" is a string array which contains all the options. In order to tell the simulator what options are available, this array must be allocated and initialized in the "initial" procedure. The simulator will allocate and set the corresponding boolean array "optVal" according to options in the instance declaration statement before calling the "setup" procedure. This will give the procedure a chance to do some useful things.

4.1.2. Order of procedure activation

The "initial" procedure is the first one to be invoked and is optional in an RTD element. If an RTD element has one or more options, the "initial" procedure should be used to initialize the option array. The "setup" procedure will be called once after the simulator finishes parsing the instance declaration statement. It sets up various static information and allocates the lBits. The "signal" procedure contains the behavioral description of the element, and is called many times during the simulation. No static information should be changed in this procedure. The "display" procedure will be called to display the element's internal data structure whenever appropriate. Procedure "setIntern" is called whenever the SET command is invoked. This procedure should allow the user to set essential internal data like the contents of a register, etc. The "setup" procedure, the "setIntern" procedure and the "display" procedure are all optional. If these are irrelevant to the RTD element, they need not be present. All procedures mentioned above have no parameters.

4.1.3. Setup procedure

The optional "setup" procedure is used to set up various static information. Its responsibilities include:

- processes the options turned on by the simulator.
- sets noOfCtrl to reflect the number of control inputs.
- sets its type, i.e., biLink or clock element
- sets size of array for memory element.
- allocates all the local lBits.

There are two attributes for RTD elements. An element can be a "biLink" element or a "clock" element. If an element is a "biLink" element, the macro "setBiLinked" must be called. A "biLink" element functions as a bridge in a bidirectional connection statement as illustrated in section 3.1.2. This type of element should be as simple as possible. In order to be a "clock" element, the macro "setClocked" must be called. A "clock" element is responsible for clock generation based on the simulation variables "systemClock" or "phi". "systemClock" is a long integer variable used by the simulator to keep track of time. It will be incremented once every subcycle. All the "clock" elements in the system will be linked to form the initial event list (Section 2). Refer to section 7 for all the global

variables, macros, and embedded procedures.

4.1.4. Signal procedure

The "signal" procedure in the element module performs the actual simulation of a physical element. Before this procedure is called, the input data and control vectors will be set to their appropriate values. The simulator expects a valid output vector when this procedure returns. Since the simulator will iterate until a steady state has been reached, the signal procedure may be called many times in one subcycle. Hence, the output and the internal state of an RTD element must be a function of the input vectors, simulation clock, and the internal state of the element only. Any dependency on the number of iterations is not allowed.

4.1.5. Display procedure

The optional "display" procedure is used to display the internal data structure of the RTD element. It will be called whenever the simulator is requested to display information about the element. All essential data structure should be displayed to aid debugging during simulation.

4.1.6. SetIntern procedure

The optional "setIntern" procedure is used to set the internal data structure of the RTD element. It will be called whenever the simulator is requested to set internal information of the element. This procedure should prompt the user interactively for information. Consult appendix A for an example.

4.1.7. lBits structure

A lBits is a pointer to a variable length bits data structure. It is of class "bitsClass". A lBits must be allocated before use.

```
class bitsClass
  (
  integer bitCount,bitsC;
  long bits array(1 to *) longBits;
  );
```

"bitCount" is used to keep the number of bits and "bitsC" is used to keep the size of the long bits array where bit 0 is the L.S.B. of "longBits[1]". When lBits is allocated, all bits will be zeroed out. If the user chooses to operate on the lBits directly, he should make sure the unused portion remains zero. A set of lBits manipulation routines is provided in the next

section. The definition of "bitsClass" is given only in the rare case in which the user wants to directly manipulate the lBits. Refer to figure 4 for the internal representation of lBits.

5. MOSSIM interface

In a typical top down design, an architecture is first specified in an abstract functional level. Then, the specifications are reduced step by step towards the actual hardware implementation. During the logic design phase, it is useful to verify portions of a design against the specifications as the design becomes mature. RTsim provides mixed-level simulation capability by allowing the replacement of an RTD element by a transistor network which is simulated by using the switch-level simulator MOSSIM II. Logic design errors can be caught early in the design phase while corrections are still relatively inexpensive. The main problems associated with the RTsim - MOSSIM interface are signal conversion, input/output naming and mapping.

5.1. NDL element

An NDL element is a MAINSAIL module written in the embedded network description language (NDL). NDL allows a network to be defined as a hierarchy of nets, where each net can contain commands to create nodes, transistors, and functional blocks, as well as calls to other nets. Please refer to the MOSSIM II User's Manual for the language. The NDL used here has been slightly modified to adapt to RTsim's conventions.

The file "ndl.mi" which resides in the RTsim directory, contains definitions for NDL. It must be included as a "sourcefile" in the MAINSAIL code. A MAINSAIL module can be declared as an NDL element by using the macro "element(modName)" instead of the usual module declaration. For example:

```
begin "dreg"
sourcefile "<RTsim directory>ndl.mi";
element(dreg);
 ...
 ...
end "dreg";
```

This will declare module "dreg" as an NDL element. An NDL element will have a predefined interface procedure "generate":

```
procedure generate(integer noOfCtrl,noOfIn,noOfOut);
```

where "noOfCtrl", "noOfIn", and "noOfOut" are widths of the input data vector, input control vector, and output data vector respectively. Within the "generate" procedure, the vectors "ctrlIn", "dataIn", and "dataOut" correspond to the input control vector, the input data vector, and the output data vector in an RTD element. They must be declared and allocated. The association between the vector names and the vector's nodes should also be saved by using the macro "keeVecRev". Refer to appendix I for an example.

5.2. Signal conversion

A signal in MOSSIM can have states in the set T={0,1,X}, where 0,1, and X represent the low, high, and invalid states respectively. In addition, a signal can be classified either as a forced or a charged signal[9]. So, a MOSSIM signal can have states in the set of T union D where T belongs to the force class and D={0c,1c,Xc} belongs to the charge class. At the input of an NDL element, an RTsim signal in the set S maps to a MOSSIM signal in the set T with the Z state in S also mapping to the X state in T. The set C maps to the set D with strength information based on the node capacitance specified in the NDL element. At the output of an NDL element, a MOSSIM signal in the set T maps to the set S with the Z state in S unmaped. The set D maps to the set C with all strength information lost. Figure 5 tabulates the signal conversion.

RTsim -> MOSSIM	MOSSIM -> RTsim
0 0	0 0
1 1	1 1
X X	X X
Z X	- Z
0c 0c	0c 0c
1c 1c	1c 1c
Xc Xc	Xc Xc

Figure 5 - RTsim-MOSSIM signal conversion

[9] Refer to section 2.3 for definition of force and charge class signals.

5.3. Input/output mapping

The characteristics of MOS pass transistors create a mapping problem for the I/O interface of an NDL element. While RTsim expects data flow in the simulated system to be unidirectional, a given implementation may utilize bidirectional characteristics of pass transistors. RTsim allows the user to equate "dataOut" to "dataIn" in an NDL element to form a bidirectional vector. Refer to appendix I for an example. If a particular NDL element has a bidirectional vector, RTsim performs a slightly different signal conversion scheme by assuming the node capacitance of the I/O vector is the greatest among all nodes connected to it. As the input of an NDL element, a RTsim signal in the force class converts to a charge class signal in MOSSIM[10] with a "shouldBeForcedFlag" set. During simulation, if a signal changes to opposite logic state, i.e., 1 to 0, 0c to 1, etc., MOSSIM will check the "shouldBeForcedFlag". If the flag is on, MOSSIM forces the signal to the X state indicating a drive conflict. As the output of an NDL element, a MOSSIM signal converts to a RTsim signal as in section 5.2. By using the CATCH command (refer to section 6), drive conflicts which are normally reported as errors, could also be caught at the I/O vector of an NDL element.

6. Future work

The present version of the simulator uses an embedded language to describe the behavior of a hardware component. The description of a concurrent component is difficult to express in the MAINSAIL language. The use of portions of a vector to represent different fields makes things even worse. A better approach is to introduce a new language capable of manipulating variable length bit vectors in 4-state logic with bit vector extraction, masking, etc., as primitive operations. This language should provide an alias mechanism. A logical name can be associated with portions of a vector with the primitive operations intelligent enough to perform field extraction automatically. This makes the code easier to write and easier to read. Another approach is to enhance the RTD language to include certain primitive elements like latches, adders, etc. The behavior of a component can be expressed in terms of a set of legal connections of these primitive elements in the "DESP" section of the language. During simulation, a special routine is called to simulate the behavior of a component together with the primitive library.

The RTsim assembler uses a table lookup method to assemble symbolic code. Macro can be implemented on top of the assembler to customize to a particular instruction format. This is particularly useful when working with a highly vertically instruction format.

[10] MOSSIM II is slightly modified.

For example, the user may wish to use "(Rn)+" to represent a post increment mode which is not possible for the RTsim assembler. "(Rn)+" may be implemented as a macro which translates to two different fields, say "Rn Pos". Finally, the assembler may be used to assemble the symbolic code.

The current implementation does not provide a loader. The RTsim assembler assembles the symbolic code to absolute addresses. A general purpose loader should be implemented, which provides maximum compatibility towards a particular instruction architecture.

The RTD file which describes the architecture is translated into the internal data structure which is used by the simulator during simulation. There is no reason why the RTD description cannot be compiled. The length of a vector is always known during compile time. Customized code can be generated to perform vector operations tailored to a particular vector length. The execution speed of the simulator can be increased significantly by compiling the RTD description into MAINSAIL code. But much more research effort is needed before a simulator compiler can be developed.

7. Simulator commands and usage

7.1. System starts up

RTsim is invoked interactively by typing the command

rtsim<eol>

while at the monitor level. When it starts up, the program will attempt to take command from file "rt.com". All commands executed at start up will not be echoed. After the end of the file is reached, the program will return with the prompt "~ " indicating it is ready to accept user input. In a typical simulation, the user will first parse a register transfer description (RTD) file to build up the data structure. Then micro and/or macro instructions may be assembled and loaded into memory. If a clocking scheme other than two-phase clocking is used, the user must set it up. Simulation can be started by various stepping and tracing commands. Refer to appendix H for a sample run. Only enough letters of a command word need be given to disambiguate it from other commands. The list of possible commands can be displayed by typing "?<eol>". A command with '#' in its first column will be treated as a comment.

7.2. System commands

7.2.1. ASSEMBLE

ASSEMBLE filename

This command assembles an ASM file. Based on the symbol table in the register transfer descriptions, the assembler will assemble and load the symbolic code to the memory instance. In order to use this command, the symbol table of the symbolic constants must be built first (see PARSE). The default file extension is asm. See file nrmin.asm and multiply.asm for examples. The following is a brief summary of various pseudo-ops and number representations, where pc is the current location the assembler working on, n is a number, b is a bit pattern, and x is a string. Refer to section 3.2 for details.

Label:

:label	- label must be preceded by a colon

Pseudo-ops:

.loc n	- set pc to n
.load x	- load symbol table for instance x
.word n	- put number n in memory
.align b	- set pc to n where n contains bit pattern b and is >= old pc.

Number representations:

n	- number in base 10
$n or $b	- number in base 16
~n or ~b	- number in base 8
'n	- number in base 2
'b	- number in base 2, can have X for don't care

7.2.2. BREAK

BREAK [instance [bitVec [*]]]

This command sets a break point at an instance. If i is present, the break point will be set at the internal data of the instance. The presence of the optional * indicates the control input. Otherwise, the data input will be set. It will break before the instance is evaluated with input bitVec. If bitVec is omitted, it will break every time before the instance is evaluated. If no argument is given, all break points currently set will be displayed with the corresponding break no, instance name and data or control bitVec. For example (in base 2):

2 break points:
1: R[0] ZZZZZZZZXXZZ1ZZ0ZZZZZZZ011101110
2: ALU Z11 *

The bitVec must be a stream of legal digits depending on the setting of switch BASE (see SWITCH BASE), where Z denotes a don't care bit and X denotes an undefined bit. If the length of bitVec does not match the length of the input vector, the bitVec will be filled with don't care bits at the left end.

7.2.3. CATCH

CATCH {instanceName}

This command sets flags to catch and report X-bit in the I/O vector of a list of instances during simulation. If no argument is given, all instances will be set. It is particularly useful to catch drive conflicts when using the MOSSIM interface (see section 5).

7.2.4. CHARGE

CHARGE instanceName bitVec [*]

This command sets the data input of an instance to the value of bitVec. If * is present, the control input will be set. As opposed to the FORCE command, this command does not attempt to hold the settings. The bitVec must be a stream of legal digits depending on the setting of switch BASE (see SWITCH BASE), where Z denotes a don't care bit and X denotes an undefined bit. If the length of bitVec does not match the length of the input vector, the bitVec will be filled with don't care bits at the left end.

7.2.5. CLEAR

CLEAR {break point no}

This command clears a list of break points. If no argument is specified, all break points will be cleared. If break point no is specified, the corresponding break point will be cleared (see BREAK).

7.2.6. CLOSELOG

CLOSELOG

This command will close the log file.

7.2.7. COMMENT

COMMENT <any text string>

This command inserts a comment line into the output stream.

7.2.8. CONTINUE or <eol>

CONTINUE | <eol>

This command continues from a pause if in a pause mode or executes the previous command if in an command mode. A pause can be generated by various debug commands (see SWITCH PAUSE, CATCH and SWITCH MONITOR). When in the pause mode, the program will use "Pause : " as a prompt.

7.2.9. CYCLE

CYCLE [n]

This command performs simulation for n cycles (see SWITCH PHASE for definition of a cycle). If n is omitted, it will be set to one.

7.2.10. DISPLAY

DISPLAY {instanceName}

This command displays the I/O information of a list of instances. If instanceName is omitted, all instances are displayed. For example:

"DISPLAY r[0],mir" may produce (in base 16)

```
R[0]   | Ctrl:01 Input:XXXXXXXX Output:ZZZZ*5*7 Value:0003
MIR    | Ctrl:0 Input:1C20030E Output: [0: :reset and zero rese1]
```

where the Z bits denoted high impedance (undriven) state. The X bits denoted undefined state, i.e. between 0 and 1 inclusive. The * bits denoted a combination of X, Z and others mixed together. If SWITCH EXPAND is set, the simulator will display a "* vector" in binary form.

7.2.11. DOCUMENT

DOCUMENT [command]

This command enters the documentation subsystem. Type ? for a list of commands. Type carriage return to leave the subsystem. Type HELP DOCUMENT * for more help information.

7.2.12. DONE

DONE

When used in pause mode, this command can cause CYCLE, SUBCYCLE or PROCEED commands to terminate in the next subcycle. If interrupt is enabled, this command will terminate simulation in progress.

7.2.13. DUMP

DUMP instanceName [filename]

This command dumps the memory content of a memory type instance on to a file. It should be used in conjunction with LOAD command. The default file name is instanceName.mem. The default file extension is mem.

7.2.14. EXAMINE

EXAMINE instanceName [n | n1-n2]

This command examines the content of a memory location. If no argument is specified, all contents will be displayed. It works only for memory type instances.

7.2.15. FORCE

FORCE instanceName bitVec [*]

This command sets the data input of an instance to the value of bitVec. If * is present, the control input will be set. When the input of an instance is set, it will be held to that value until the UNFORCE command is used. The bitVec must be a stream of legal digits depending on the setting of switch BASE (see SWITCH BASE), where Z denotes a don't care bit and X denotes an undefined bit. If the length of bitVec does not match the length of the input vector, the bitVec will be filled with don't care bits at the left end.

7.2.16. FREEZE

FREEZE [filename]

This command saves all information about the current state of the simulator on a file. Since RTD elements are MAINSAIL modules which may have their own or internal variables, the program will go below data types ordinarily supported by MAINSAIL. Primitive data types like address or charAdr will not be supported. So is the internal data of other modules which is binded by the RTD element. Within an RTD element, no dynamic allocation is allowed during the simulation process. All new or newUpperBound statements must be used in the setup or initial procedure (see section four of this thesis). This command does not work with NDL elements. (See NDL and MOSSIM.) The default file name is rt.frz. The default file extension is frz. (Also see RESTORE.) RESTORE)

7.2.17. HELP

HELP [command [subCommand]]

This command enters the help subsystem. Type ? for a list of topics. Type carriage return to leave the subsystem. Type HELP * to get an explanation of all top level commands. Type HELP subsystem * to get an explanation of all subsystem commands.

7.2.18. LIBRARY

LIBRARY {filename}

This command opens a list of library files. RTD or NDL elements needed in the RTD file (see PARSE, NDL and RTD) will be searched through the already opened libraries as

supported by the underlining MAINSAIL language. Libraries are searched in order starting with the most recently opened. The document subsystem (see DOCUMENT) provides the user the facility of module sharing. The user may choose to put his LIBRARY commands in the rt.com file which is executed when system the starts up. The default file extension is lib.

7.2.19. LIST

LIST {instanceName}

This command prints a list of instances in the current simulation environment together with its module name, module type and various attributes. For example:

"LIST r[3],clk" may produce

Instance: R[3]@ Type: %ZDREG
Instance: CLK* Type: CLOCK2

If an instance's module type is preceded by a %, it is an NDL element. Otherwise, it is an RTD element. The presence of the @ character denotes the instance is linked to a bilink element (see PARSE). If the * character is present, it is a clock element.

7.2.20. LOAD

LOAD instanceName [filename]

This command loads the memory content of a memory type instance from a file. The default file name is instanceName.mem. (Also see DUMP.)

7.2.21. MAINSAIL

MAINSAIL compile | debug

This command calls the MAINSAIL compiler or debugger depending upon its argument.

7.2.22. MOSSIM

MOSSIM [mossimCommand]

This command enters the MOSSIM command subsystem. Type ? for a list of commands. Type carriage return to leave the subsystem. Refer to MOSSIM User's Manual for assistance.

7.2.23. NDL

NDL instanceName NDLmoduleName

This command replaces an RTD element by an NDL element. During simulation, the transistor network in the NDL element will be simulated using the switch-level simulator MOSSIM II. (See NETWORK, MOSSIM, RTD, and LIST.) Also refer to the MOSSIM

interface section in this thesis.

7.2.24. NETWORK

NETWORK [filename] [*]

This command generates a network file from all NDL elements. The resulting file which contains a forest of transistor networks, is read in by MOSSIM. (See MOSSIM, NDL, LIST, and RTD.) If * is present, the file is used without regenerating the networks. If no file name is given, the default file rt.ntk is used. This command will be called automatically by the simulation commands if the simulator detects a change in status of the NDL elements. The default file extension is ntk.

7.2.25. OPENLOG

OPENLOG [filename]

This command opens or appends to a log file. All terminal conversations will be recorded on the log file. If the filename is omitted, it will append to the previously opened log file. If no log file is previously opened, it will open the default file rt.log. The default file extension is log. Also see CLOSELOG.

7.2.26. PARSE

PARSE filename

This command parses the syntax and builds the data structure from an RTD file. The default file extension is rtd. For more details, see section 3 of this thesis. Also see file nrmin.rtd for an example.

7.2.27. PROCEED

PROCEED [n]

This command will proceed with the simulation for at most n subcycles. If break point is encountered before the subcycle count is up, this command will first break and then return control to the user in the next subcycle. If n is omitted, it will be set to something large.

7.2.28. QUIT

QUIT

This command terminates the program and closes all opened files.

7.2.29. RESTORE

RESTORE [filename]

This command restores the simulator states saved by the FREEZE command. The default file name is rt.frz.

7.2.30. RTD

RTD {instanceName}

This command reverses the effect of the NDL command.

7.2.31. SET

SET instanceName

This command sets the internal data of an instance. It should prompt the user for the information needed. The detail varies with the RTD elements.

7.2.32. SUBCYCLES

SUBCYCLE [n]

This command performs simulation for n subcycles. A subcycle is defined as the period from one steady state to another. If n is omitted, it will be set to one.

7.2.33. SWITCH

SWITCH [switchName]

This command enters the switch subsystem. Type ? for a list of switches. Type carriage return to leave the subsystem. Type HELP SWITCH * for more help information.

7.2.34. TAKE

TAKE filename

This command re-directs the command input from a file. The default file extension is com. The file rt.com is always executed when RTsim starts up. The user is encouraged to put in commands to set up various switches and search paths. The command interpreter will ignore all lines with '#' in its first column. This command can be nested.

7.2.35. TRANSLATE

TRANSLATE inst1 [inst2] [n1-n2] [*]

This command translates a portion of the output vector of inst1 according to the symbol table defined for inst2. Hence inst2 must be a memory type instance. n1 and n2 are the starting and ending indexes of the vector to be translated. The content of inst2 is assumed not to be changed throughout the simulation. The translated output will not reflect those changes. For example:

~ TRANSLATE mir store
~ DISPLAY mir

may give

MIR | Ctrl:0 Input:1C20030E Output: [0: :reset and zero rese1]

If only inst1 is given, inst1 will be translated according to its own symbol constants. If * is

present, this command will also check and report for identical words in the memory. (Also see UNTRANSLATE)

7.2.36. UNCATCH

UNCATCH {instanceName}

This command reverses the effect of CATCH.

7.2.37. UNFORCE

UNFORCE instanceName [*]

This command releases the holding of the input vector of an instance enabled by the FORCE command. The input vector of the instance will then be collected from its predecessors and its output will propagate until a steady state is reached.

7.2.38. UNTRANSLATE

UNTRANSLATE {instanceName}

This command reverses the effect of TRANSLATE.

7.2.39. UNWATCH

UNWATCH {instanceName}

This command reverses the effect of WATCH.

7.2.40. UPDATE

UPDATE {moduleName}

This command updates all instances of an RTD element module. The instances will first be disposed of, and then binded with the same options turned on.

7.2.41. WATCH

WATCH [phase] {instanceName} [*]

This command sets flags to display information about a list of instances. If * is omitted, all I/O and internal information will be displayed. Otherwise, only internal information of an instance will be displayed. If the instance name is not given, all instances will be set. The phase tells the simulator what phase in a cycle to watch. If omitted, all phases will be watched. The current phase and cycle number can be displayed by switch STATISTIC. (See SWITCH STATISTIC.)

7.3. Document subsystem

7.3.1. DELETE

DELETE elementName

This command deletes a document filed under elementName. Only the owner of the document can delete it by knowledge of its secret code. When used constructively

with FILE and PRINT, the users can share their RTD elements.

7.3.2. FILE

FILE

This command files an RTD document. When invoked, the program will prompt interactively for the various needed information. When used in conjunction with PRINT constructively, the users will be able to share access to their RTD element modules. The DELETE command is provided in case the user chooses to delete or update his document. Also see PRINT and DELETE.

7.3.3. PRINT

PRINT [elementName | searchKey]

This command displays documentation on an element or searches the headers for a searchKey. The searchKey must be a string of alphabets.

With no argument, headers of all entries will be displayed. The following shows a typical entry for an RTD element:

Element : dreg
Designer: lam
Location: <path name>ele.lib
Header : A dual port, static register
Descriptions:

Element descriptions

Also see FILE and DELETE.

7.4. Switch subsystem

7.4.1. BASE

BASE [n]

This command sets the I/O format of vectors to either base 2, 8 or 16 by setting n to 2, 8 or 16 respectively. If n is omitted, the current format will be displayed. The default setting is 2.

Legal formats:
base 2 = {0,1,X,Z}
base 8 = {0,1,2,3,4,5,6,7,X,Z}
base 16 = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,X,Z}

Base 10 is not supported.

7.4.2. CAPACITANCE

CAPACITANCE [on | off]

This command models MOS capacitance. If turned on, the input vectors of an instance will hold their values for N more subcycles after its driving signals are turned off, where the charge hold time N is the setting of SWITCH CHARGE-HOLD-PERIOD. The period N will assume infinite if CAPACITANCE is set to off, which is the default. If no argument is given, the current setting will be displayed.

7.4.3. CHARGE-HOLD-PERIOD

CHARGE-HOLD-PERIOD [n]

This command sets the charge hold period to model MOS capacitance. (See SWITCH CAPACITANCE.) If no argument is given, the current setting will be displayed. The default is 0.

7.4.4. DEBUG

DEBUG [on | off]

This command is used by the author to debug the simulator. The default is off.

7.4.5. EXPAND

EXPAND [on | off]

This command overrides the setting of SWITCH BASE if turned on. All vectors having X or Z bits will be displayed in binary form. The default is off.

7.4.6. INTERRUPT

INTERRUPT [on | off]

This command controls interrupt. If turned on, simulation commands like SUBCYCLE, CYCLE, and PROCEED will yield to any command entered from the terminal. The simulator will resume execution after all terminal commands have been processed. The default is on. If no argument is given, the current setting will be displayed. This command is only available on some systems.

7.4.7. ITERATIONS

ITERATIONS [n]

This command sets the maximum number of iterations to n. Steady state must occur within n iterations or the simulator will report an error. If n is omitted, the number of iterations currently set will be displayed. The iteration variable is set to 16 when the system starts up. In case the user wishs to examine excitation statistics, he can use the SWITCH STATISTIC command.

7.4.8. MONITOR

MONITOR [on | off]

This command monitors the simulation step by step. The simulator will display the I/O data and pause after an instance is evaluated. It is useful to trace the firing sequence and verify the connection scheme. If no argument is given, the current setting will be displayed. The default is off.

7.4.9. PAGEWIDTH

PAGEWIDTH [n]

This command controls the output page width. If no argument is given, the current setting will be displayed. The default is 80.

7.4.10. PAUSE

PAUSE [on | off]

This command sets the PAUSE flag. When PAUSE is on, the simulator will prompt the user whenever an error is reported. If no argument is given, the current setting will be displayed. The default is on.

7.4.11. PHASE

PHASE [n]

This command sets the number of phases in a clock cycle. It will be initialized to two when RTsim starts up. It is useful to implement different clocking conventions. If n is omitted, its current setting will be displayed.

7.4.12. STATISTIC

STATISTIC

This command displays excitation statistics, cycle number, and phase information for the current subcycle.

8. Utilities

8.1. lBits utilities

The following lBits manipulation procedures are provided. This is an attempt to free the element designer from knowing the detail representation of lBits. The unused portion of the lBits is expected to be zero before the procedure call and will stay zero afterwards.

procedure andBits(lbits dst,src);

This procedure performs a bitwise logical and on the two lBits, and puts the result in

dst. If the length of lBits do not match, the length of src will be modified to that of dst.

integer procedure bitCount(lBits src);

The procedure returns the number of 1-bit in src.

bits procedure bitVal(lBits src; integer offset,length);

This procedure returns a set of at most 16 bits starting from a given offset, where bit 0 is the L.S.B.

procedure checkBreak(lBits d,x);

This procedure checks a vector[11] against all internal break points set by the user. It will cause the simulator to break if a match is found.

procedure clearAbit(lBits dst; repeatable integer n);

This procedure clears bit n of dst where bit 0 is the L.S.B.

procedure clrBits(lbits dst,src);

This procedure clears any 1-bit in dst whose corresponding location in src is also 1-bit. If the length of lBits do not match, the length of src will be modified to that of dst.

procedure copyBits(modifies lBits dst; lBits src);

This procedure copies from src to dst. If dst has not been allocated, getBits will be called. If the length of lBits do not match, the length of dst will be modified to that of src.

procedure displayBits(lBits d; optional lBits x; optional integer format);

If lBits x is present, this procedure outputs the vector to the terminal. If not, the lBits d will be output. The output format of the vector depends on the value of the format, which is either 2, 8 or 16.

boolean procedure equBits(lBits src1,src2);

This procedure returns true if src1 equals src2.

procedure extendBits(lBits dst; integer length);

This procedure modifies the length of dst.

procedure extractBits(modifies lBits dst; lBits src; integer offset,length);

This procedure extracts from src a lBits of specific length starting from a specific offset. Dst is allocated if necessary.

lBits procedure getBits(integer length);

Since a lBits variable is a pointer, it is necessary to allocate the structure before using it. This procedure will allocate a lBits of specific length and return the pointer. The newly allocated lBits with be zeroed out.

---

[11] A vector consists of 2 lBits (d and x) where each bit can have values in in the set {0,1,X,Z}. Refer to figure 4.

lBits procedure getMask(integer length,startIndex,endIndex);

This procedure returns a lBit of specific length with bit startIndex to endIndex set to one.

procedure inputInt(produces integer n);

This procedure reads in an integer from the terminal. This is preferred because it will output to the log file if necessary.

procedure inputVec(lBits d,x);

This procedure reads a string from the terminal, converts it to a vector according to the setting of SWITCH BASE, and returns the vector as its arguments. The lBits d and x must be allocated explicitly. If the input is a null string, d and x will not be changed.

boolean procedure isAllOneBits(lBits src);

This procedure returns true if all bits are set to one in src.

boolean procedure isAllZeroBits(lBits src);

This procedure returns true if all bits are set to zero in src.

procedure killBits(repeatable modifies lBits dst);

This procedure disposes the lBits pointed to by dst.

procedure negBits(lBits dst);

This procedure performs a one's complement on dst.

procedure orBits(lbits dst,src);

This procedure performs a bitwise inclusive or on the two lBits, and puts the result in dst. If the length of lBits do not match, the length of src will be modified to that of dst.

procedure setAbit(lBits dst; repeatable integer n);

This procedure sets bit n of dst to one.

procedure setAllOneBits(repeatable lBits dst);

This procedure sets all bits in dst to one.

procedure setAllZeroBits(repeatable lBits dst);

This procedure clears all bits in dst.

procedure setBits(lBits dst,src; integer offset);

This procedure copies src onto dst starting from a specific offset. The length of dst will be modified if necessary.

procedure setBitsWithMask(lBits dst,src,mask);

This procedure will set a portion of dst specified by the mask to that of src. Every bit in dst will be replaced by src if the corresponding mask bit is set.

macro setUndefVec(lBits d,x);

This macro sets the vector to the X state. Refer to figure four for the X state

representation.

procedure shiftBits(lBits dst; integer n);

This procedure does a logical shift right for n bits if n is positive. If n is negative, it will shift left.

integer procedure strToInt(string s; optional integer format);

This procedure converts a string of digits to an integer according to the value of the format which is either 2, 8 or 16.

boolean procedure strToVec(string s; modifies lBits d,x; optional integer format);

This procedure converts a string to a vector and returns true if successful. The conversion depends on the setting of the format which is either 2, 8 or 16. The vector will be allocated if necessary.

integer procedure sumBits(lBits src);

This procedure returns the integer representation of the least significant 15 bits of src.

string procedure vecToStr(lBits d; optional lBits x; optional integer format);

This procedure converts a lBits d to a string. The conversion depends on the setting of the format which is either 2, 8 or 16. If x is present, it converts a vector to a string.

procedure xorBits(lbits dst,src);

This procedure performs a bitwise exclusive or on the two lBits, and puts the result in dst. If the length of lBits do not match, the length of src will be modified to that of dst.

8.2. Others

string procedure curIns;

This procedure returns the name of the currently active instance. CurIns may be used in an RTD element to get its corresponding instance's name.

boolean procedure printVec(string s; optional boolean eject);

This procedure prints a string justified according to page width. (See SWITCH PAGEWIDTH.) If eject is true, a carriage return will be appended to the end of the string. If any carriage return is printed by the procedure, it returns true. Refer to appendix A for an example of its usage.

macro scriptOut(s);

This macro prints the string in the log file if it is opened.

macro errorOut(s);

This macro is the same as macro ttyOut followed by macro pauseIfEnable. See below.

macro ttyOut(s);

This macro is the same as ttyWrite, with the exception that it also outputs to the log file if it necessary. This macro should be called with the following format.

ttyOut([<parameters>]);

macro ttyCOut(s);

This macro is the same as ttyCWrite except it also outputs to the log file if it is opened.

macro pauseIfEnable;

This macro will prompt for user input if SWITCH PAUSE in the command system is set.

macro setBiLinked;

This macro turns an RTD element into a biLink element.

macro setClocked;

This macro turns an RTD element into a clock element.

macro setRefreshClock(refreshClock);

This procedure sets refreshClock to the simulation clock value N subcycles later, where N is the setting of SWITCH CHARGE-HOLD-LIMIT if SWITCH CAPACITANCE is on or infinite if it is off.

boolean macro refreshClockExpired(refreshClock);

This macro returns true if refreshClock equals systemClock.

8.3. Simulation variables

long integer systemClock

This is the simulation clock variable. It is incremented once every subcycle.

integer phi,maxPhi

Phi denotes the current phase in a clock cycle. It counts from zero to maxPhi-1 and back to zero, where maxPhi is set by the SWITCH PHASE (section 6) command.

Appendix A - An RTD element example - dual port register

```
#
#   FILE:   dreg.fun
#          dual port static register
#          5 control lines, from 0 to 4: refresh,ldA,ldB,rdA,rdB
#

begin "dreg"
noCheck;
sourcefile "rtsim.h";                  # file in rtsim directory
element(dreg);

#   Global variables
lBits bd,bx;
integer len;
long integer refClk;                   # keeps the refresh clock limit

procedure setup;
#   set up no of control lines
begin
    noOfCtrl:=5;                  # no of control lines
    len:=noOfIn div 2;                 # get the offset or length for bus B
    if noOfIn mod 2 neq 0 then
      begin
      ttyOut(curIns&": no of bits must be multiple of 2"&eol);
      return;
      end;
    bd:=getBits(len);                  # allocates locals
    bx:=getBits(len);
end;

procedure display;
printVec("Value:"&vecToStr(bd,bx));

procedure setIntern;
begin
    ttyOut("Value: ");                 # prompt for node value
    inputVec(bd,bx);                   # input a vector
    setRefreshClock(refClk);           # data retain period depends on duration
end;

procedure signal;
begin
    bits b;
    if refreshClockExpired(refClk) then     # refresh period expired?
      setUndefVec(bd,bx);      # data become undefined
```

```
if bitVal(cx,0,5) neq '37 then          # something no defined?
   begin
   setUndefVec(bd,bx);      # everything undefined
   setUndefVec(od,ox);
   checkBreak(bd,bx);       # check for internal break
   return;
   end;
b:=bitVal(cd,0,5);                  # extract start from bit 0 for length 5
if b='1 then                # refresh data?
   setRefreshClock(refClk)          # set data retain period
ef not (b msk 'B11001) then         # ldA or ldB?
   begin
   case cvi(bitVal(cd,1,2)) of
       begin
       [0] ;                # no op
       [1]      begin               # ldA
         extractBits(bd,id,0,len);# get bits start from 0 of length len
         extractBits(bx,ix,0,len);   # that is <0:len>
         setRefreshClock(refClk);    # set data retain period
         end;
       [2]      begin               # ldB
         extractBits(bd,id,len,len);# get bits start from len of len len
         extractBits(bx,ix,len,len);   # that is <len:2*len-1>
         setRefreshClock(refClk);    # set data retain period
         end;
       [3] setUndefVec(bd,bx); # fighting
       end;
   end
ef not (b msk 'B00111) then         # rdA or rdB?
   begin
   case cvi(bitVal(cd,3,2)) of
       begin
       [0]      ;                  # no op
       [1]      begin              # rdA
         setBits(od,bd,0);   # copy all bits of bd into od start at 0
         setBits(ox,bx,0);
         end;
       [2]      begin              # rdB
         setBits(od,bd,len); # copy all bits of bd into od start at len
         setBits(ox,bx,len);
         end;
       [3] begin           # rdA and rdB (valid)
         setBits(od,bd,0);   # copy all bits of bd into od start at 0
         setBits(ox,bx,0);
         setBits(od,bd,len); # copy all bits of bd into od start at len
         setBits(ox,bx,len);
         end;
       end;
   end
else                        # fighting
   begin
```

```
        setUndefVec(bd,bx);      # everything undefined
        setUndefVec(od,ox);
        end;
    checkBreak(bd,bx);                  # check for internal break
end;

end "dreg";
```

Appendix B - An RTD element example - static RAM

```
#
#   FILE:   sram.fun
#         contain the sram element for simulation
#

begin "sram"
noCheck;
sourcefile "rtsim.h";                  # file in rtsim directory
element(sram);

#    control vector meaning
#    0x                    tristate
#    10                    write
#    11                    read

lBits bd,bx,b,x;

procedure setup;
begin
    noOfCtrl:=2;              # read/write control, 1=write, 0=read
    depth:=noOfIn-noOfOut;    # no of addr bits
    bd:=getBits(depth);             # allocate address bit
    bx:=getBits(depth);             # allocate address bit
    b:=getBits(noOfOut);      # allocate locals
    x:=getBits(noOfOut);
end;

procedure setIntern;
begin
    integer loc;
    ttyOut("Location: ");     # prompt for location
    inputInt(loc);            # get an integer
    ttyOut("Value   : ");     # prompt for value
    inputVec(b,x);            # get a input line and set
    if loc<0 or loc>bitArr.ubl then # check index
      begin
      ttyOut(curIns&": Index out of bound"&eol);
      return;
      end;
    copyBits(bitArr[loc],b);        # save bit value
end;

procedure signal;
begin
    integer addr;
    if not bitArr then              # array not initialize
```

```
      begin
      ttyOut(curIns&": RAM not initialized"&eol);
      return;
      end;
    if isAllZeroBits(cx) then          # undefined
      begin
      errorOut(curIns&": control input undefined"&eol);
      return;
      end;
    if bitVal(cd,1,1)='0 then          # tristate
      return;
    extractBits(bd,id,noOfOut,depth);
    extractBits(bx,ix,noOfOut,depth);
    if bitVal(cd,0,1)='0 then          # read
      begin
      if isAllOneBits(bx) then
          begin
          addr:=sumBits(bd);           # calculate bit sum of input
          if bitArr[addr] neq nullPointer then
            copyBits(od,bitArr[addr])
          else
            begin
            ttyOut([curIns,": uninitialized location ",addr,
                "(decimal), assume X",eol]);
            setUndefVec(od,ox);
            return;
            end;
          setAllOneBits(ox);           # all output defined
          end
      else
          setUndefVec(od,ox);
      end
    else                          # write
      begin
      if isAllOneBits(ix) then
          begin
          addr:=sumBits(bd);           # calculate bit sum of input
          extractBits(bitArr[addr],id,0,noOfOut);
          end
      ef isAllOneBits(bx) then
          begin
          addr:=sumBits(bd);           # calculate bit sum of input
          bitArr[addr]:=nullPointer;       # clear the pointer
          end
      else
          errorOut([curIns,": input address bit undefined for",
            " memory write",eol]);
      end;
end;

end "sram";
```

Appendix C - BNF description for RTD language

BNF

```
Anything preceded by ' will be a punctuation mark inside the language
{}* -- zero or more times
{}+ -- one or more times
[] -- optional
( | ) -- either one
# -- continue on next line
The rest are pretty much standard


prg ::= eleDec eleCon [ eleDes ] EOF

eleDec ::= INST decBody
decBody ::= { decOne }*
decOne ::= ID [ '[ NUM : NUM '] ] \ ID [ '[ NUM '] ]     #
           [ < bits > ] [ ( ID {, ID}* ) ] ;
bits ::= NUM : NUM [ '| NUM : NUM ] | '| NUM : NUM | NUM : NUM '|

eleCon ::= CONN conBody
conBody ::= { conOne1 | conOne2 }*
conOne1 ::= onePart1 { - [ NUM ] onePart1 }+ ;
conOne2 ::= onePart2 = { onePart2 }+ ;
onePart1 ::= ID [ '[ NUM '] ] . (in|out|ctrl) [ < bitCon > ]
onePart2 ::= ID [ < bitDesp > ]
bitDesp ::= NUM : NUM

eleDes ::= DESP desBody
desBody ::= { oneEle }*
oneEle ::= ID blocks
blocks ::= caseStat | BEGIN { caseStat }+ END ;
caseStat ::=  CASE out < bitDesp > IN GROUP NUM OF subBlocks
subblocks ::= equStat | BEGIN { equStat }+ END ;
equStat ::= NUM = ID ; | [ NUM ] '[ NUM - NUM '] = ID '[ NUM - NUM '] ;

ID ::= { letter }+ { letter | digit | _ }*
NUM ::= ' { 0..1 }+ | ~ { 0..7 }+ | $ { 0..7 | A..F }+ | digit+
letter ::= A..Z
digit ::= 0..9
```

Appendix D - Network model of MIN processor in RTD

```
#
#   FILE:   NRMIN.RTD
#           Based on the MIN processor of "How to Flowchart for Hardware"
#           by Nick Tredennick in the Dec 1981 issue of Computer[4].
#

#   Microword format
#      0-5    next address
#      6-7    address select
#      8-9    constant select
#      10-12  ALU function select
#      13-16  B dest.
#      17-19  B source
#      20-23  A dest.
#      24-26  A source
#      27-28  IRs

#   Instruction format
#      Group 1                    Group 2
#      0-2    Ry          0-2     Ry
#      3-4    Mode        3-6     CC
#      5-7    Rx          7-7     Not
#      8-11   opcode      8-11    opcode

INST   # instances declaration section
    # execution unit declaration
    AO\addreg<0:31 | 0:15>;          # address register, 32 bits in, 16 bits out
    PC\dreg<0:31>;              # program counter
    T2\dreg<0:31>;              # temporary registers
    T1\treg<0:47 | 0:31>;       # temporary reg., 48 bits in, 32 bits out
    R[0:7]\dreg<0:31>;               # general registers
    ALU\minALU<0:47 | 0:19>;         # ALU, 48 bits in, 20 bits out
    K\const<| 0:15>;                 # constant generator, no inputs
    DIN\reg1<0:15>;             # data input register
    DO\reg1<0:15>;              # data output register
    IR\reg1<0:15>;              # instruction register
    IRE\reg1<0:15>;             # instruction register for execution
    Bus\biLink<0:31>;                # bus A and B

    # microprogram control unit declaration
    STORE\rom<0:5 | 0:28>;      # control store
    MIR\latch<0:28> (posTran);       # micro instruction register
    MDL\decod2<0:30 | 0:76>;         # microcode decoder logic
    ID\decod5<0:15 | 0:7>;      # instruction decoder
    RESET\resdis<0:9 | 0:5>;         # reset and dispatch circuit
    MPX\mux<0:15 | 0:3>;        # address input multiplexor
```

```
    FL\latch<0:3> (negTran);          # 1 bit latch
    MAM\addMod<0:7 | 0:3>;      # micro address modifier
    CLK\clock2<| 0:1>;                # 2-phase clock distributor
    ZERO\zero<| 0:3>;                 # output constant zero

    # External Memory
    MEM\sram<0:25 | 0:15>;      # primary memory


CONN  # connection section
    # execution unit connections

    # Bus connections
    Bus = PC = T1 = T2 = R;           # on both bus A and B
    Bus.out = AO.in = ALU.in;         # take input from bus A and B
    Bus.out<0:15> = DO.in;      # take input from bus A only
    DIN.out = Bus.in<16:31>;          # drive bus B only

    K.out = ALU.in<32:47>;      # connect K to ALU
    ALU.out<0:15> = T1.in<32:47>;     # connect ALU to T1
    ALU.out<16:19> = FL.in;           # flag bit
    MIR.out<12:12> = FL.ctrl;         # latch bit
    FL.out = MAM.in;                  # latch flag on ALU instructions
    DO.out = MEM.in;                  # connect DO to MEM
    MEM.out = DIN.in = IR.in;
    IR.out = IRE.in;

    # control unit connections
    IRE.out = ID.in;                  # connect to instruction decoder
    IRE.out<0:2> = MDL.in<24:26>;     # connect to static decode, Ry
    IRE.out<3:6> = MAM.in<4:7>;       # connect condition code mask
    IRE.out<7:7> = MAM.ctrl;          # control negation
    IRE.out<5:7> = MDL.in<21:23>;     # connect to static decode, Rx
    IRE.out<8:11> = MDL.in<27:30>;    # connect to static decode, opCode
    ID.out = MPX.in<8:15>;            # RB, RD
    MAM.out = MPX.in<4:7>;      # BC
    ZERO.out = MPX.in<0:3>;           # zero input
    MPX.out = RESET.in<6:9>;
    RESET.out = STORE.in;       # control store address
    STORE.out = MIR.in;               # latch in micro instruction register
    MIR.out<0:5> = RESET.in<0:5>;     # DB
    MIR.out<6:7> = MPX.ctrl;          # address select
    MIR.out<8:28> = MDL.in<0:20>;     # connect to decoder logic
    CLK.out<0:0> = MDL.ctrl;          # connect to MDL
    CLK.out<1:1> = MIR.ctrl;          # connect to MIR

    AO.out<0:9> = MEM.in<16:25>;      # connect to external memory

    # control lines
    MDL.out<0:2> = AO.ctrl;
    MDL.out<3:7> = PC.ctrl;
```

```
    MDL.out<8:12> = T2.ctrl;
    MDL.out<13:17> = T1.ctrl;
    MDL.out<18:22> = R[0].ctrl;
    MDL.out<23:27> = R[1].ctrl;
    MDL.out<28:32> = R[2].ctrl;
    MDL.out<33:37> = R[3].ctrl;
    MDL.out<38:42> = R[4].ctrl;
    MDL.out<43:47> = R[5].ctrl;
    MDL.out<48:52> = R[6].ctrl;
    MDL.out<53:57> = R[7].ctrl;
    MDL.out<58:60> = ALU.ctrl;
    MDL.out<61:62> = K.ctrl;
    MDL.out<63:65> = DIN.ctrl;
    MDL.out<66:68> = DO.ctrl;
    MDL.out<69:71> = IR.ctrl;
    MDL.out<72:74> = IRE.ctrl;
    MDL.out<75:76> =3 MEM.ctrl;

DESP

    STORE
    begin                       # symbolic microcode definitions for STORE
      case out<0:5> in group 1 of
          begin                 # allocate the next address field
          0 = absolute;         # specify an absolute address field
          end;
      case out<6:7> in group 1 of
          begin                 # address select
          0 = def;              # default is DB address
          1 = BC;               # BC address
          2 = RB;               # select opcode
          3 = RD;               # select addr. mode
          end;
      case out<8:9> in group 1 of
          begin                 # constant select
          1 = plus1;                  # +1
          0 = zero;             # 0
          2 = minus1;                 # -1
          3 = def;              # default, is from B bus
          end;
      case out<10:12> in group 1 of
          begin                 # ALU function select
          0 = def;              # default, nop
          1 = add;              # addition
          2 = sub;              # subtraction
          3 = and;              # logical and
          4 = op;               # static decode
          5 = test;             # test data
          end;
      case out<13:16> in group 1 of
```

```
    begin               # B dest.
    0 = nop;            # nop
    1 = PC;             # program counter
    2 = AO;             # address register
    3 = T2;             # temporary register
    4 = T1;             # temporary register
    5 = Rx;             # use Rx
    6 = Ry;             # use Ry
    9 = RxT2;           # Rx and T2
    10 = RyT2;                  # Ry and T2
    11 = RyAO;                  # Ry and AO
    12 = T2AO;                  # T2 and AO
    end;
case out<17:19> in group 1 of
    begin               # B source
    0 = nop;            # nop
    1 = PC;             # program counter
    3 = T2;             # temporary register
    4 = T1;             # temporary register
    5 = Rx;             # use Rx
    6 = Ry;             # use Ry
    7 = DIN;            # data input register
    end;
case out<20:23> in group 1 of
    begin               # A dest.
    0 = nop;            # nop
    1 = PC;             # program counter
    2 = AO;             # address register
    3 = T2;             # temporary register
    4 = T1;             # temporary register
    5 = Rx;             # use Rx
    6 = Ry;             # use Ry
    7 = DO;             # data output register
    9 = RxT2;           # Rx and T2
    10 = RyT2;                  # Ry and T2
    11 = RyAO;                  # Ry and AO
    12 = T2AO;                  # T2 and AO
    end;
case out<24:26> in group 1 of
    begin               # A source
    0 = nop;            # nop
    1 = PC;             # program counter
    3 = T2;             # temporary register
    4 = T1;             # temporary register
    5 = Rx;             # use Rx
    6 = Ry;             # use Ry
    end;
case out<27:28> in group 1 of
    begin               # instruction registers
    0 = def;            # default, nop
    1 = ldr;            # load IR
```

```
      2 = ldre;           # load IRE
      3 = ldd;            # load DIN
      end;
end;

MEM
begin                     # symbolic macro instructions
  case out<0:2> in group 1 of # in base 2, Ry
      ['0-'111] = R[0-7];       # 0 = R0, to 7 = R7
  case out<3:4> in group 1 of
      begin               # mode
      0 = def;            # default, register direct
      1 = I;              # register indirect
      2 = D;              # base + displacement
      end;
  case out<5:7> in group 1 of      # Rx
      [0-7] = R[0-7];     # 0 = R0, to 7 = R7
  case out<8:11> in group 1 of         # in base 8
      begin               # opcode
      $1 = add;           # operate class
      $2 = sub;
      $3 = and;

      $4 = load;                # general addr. mode
      $5 = test;
      $6 = store;

      $8 = push;                # special instruction
      $9 = pop;
      $a = zero;

      $f = reserved_group_2_instruction;
      end;

  case out<8:11> in group 2 of
      $f = br;            # branch class
  case out<3:7> in group 2 of
      begin               # branch condition
      $0 = NR;            # never
      $10 = def;                # always
      $1 = C;             # carry set
      $11 = NC;           # carry not set
      $2 = N;             # negative
      $12 = NN;           # not negative
      $4 = Z;             # zero
      $14 = NZ;           # not zero
      $8 = V;             # overflow
      $18 = NV;           # not overflow
      end;
  case out<0:2> in group 2 of # in base 2
      ['0-'111] = R[0-7];       # 0 = R0, to 7 = R7
```

```
case out<0:9> in group 3 of
    0 = absolute;        # allocate an absolute address field
end;
```

Appendix E - Microcode for MIN processor

```
RTsim Assembler Listing for nrmin.asm


        #
        #   FILE:   NRMIN.asm
        #           symbolic microcode for the MIN processor
        #

              .load STORE               # load symbol table

              .loc 0
        #   reset
0:      :reset and zero resel                   # get PC at loc 0

        #   addressing mode dispatch (RD dispatch to $0)
        .loc 1
1:      :adrm1 ldd Ry T2A0 RB brzz3             # Ry, indirect
2:      :abdm1 ldd PC A0 add plus1 abdm2    # (Ry + d), base + disp.

        #   Reg-Reg instructions    (RD dispatch to $0)
        .loc 3
3:      :oprr1 Rx nop Ry op oprr2       # add,and,sub
4:      :ldrr1 ldr PC A0 Ry RxT2 add plus1 ldrm2    # load
5:      :tstr1 Rx test zero brzz3       # test
6:      :strr1 ldr PC A0 Rx RyT2 add plus1 ldrm2    # store

        #special instructions (RD dispatch to $0)
        .loc $8
8:      :push1 Ry add minus1 push2      # push
9:      :popr1 ldd Ry A0 add plus1 popr2    # pop
10:     :zero1 ldr PC T2A0 and zero zero2   # zero

        # rest of the reset sequence
        .loc $D
13:     :resel ldd T1 A0 rese2
14:     :rese2 DIN PC brzz3

        # start of branch instruction (RD dispatch to $0)
        .loc $F
15:     :brzz1 ldr Ry A0 add plus1 BC brzz3      # branch on condition

        #conditional branch instructions (BC dispatch to $10)
        .loc $10
16:     :brzz3 ldr PC A0 add plus1 brzz2    # here, if cond. is false
17:     :brzz2 ldre T1 PC RD reset            # here, if cond. is true
```

```
        # rest of mem operate
        .loc $12
18:     :oprm2 T1 DO T2 AO brzz3

        #mem ref. instructions  (RB dispatch to $10)
        .loc $13
19:     :oprm1 Rx nop DIN op oprm2      # add,and,sub
20:     :ldrm1 ldr PC AO DIN RxT2 add plus1 ldrm2  # load
21:     :test1 ldr PC AO DIN T2 add plus1 ldrm2    # test
22:     :strm1 Rx DO T2 AO test zero brzz3  # store

        #   the rest of the addressing mode sequence
        .loc $18
24:     :abdm2 T1 PC abdm3
25:     :abdm3 Ry nop DIN add abdm4
26:     :abdm4 ldd T1 T2AO RB brzz3
        #   the rest of the sequences
27:     :oprr2 ldr PC AO T1 Ry add plus1 brzz2      #  2nd step
28:     :popr2 T1 Ry DIN Rx brzz3       #  2nd step
29:     :push2 Rx DO T1 RyAO brzz3      #  2nd step
30:     :zero2 T2 nop T1 Rx add plus1 brzz2         # zero register Rx
31:     :ldrm2 ldre T2 nop T1 PC test zero RO reset     #  2nd step
```

Appendix F - Sample program for MIN processor

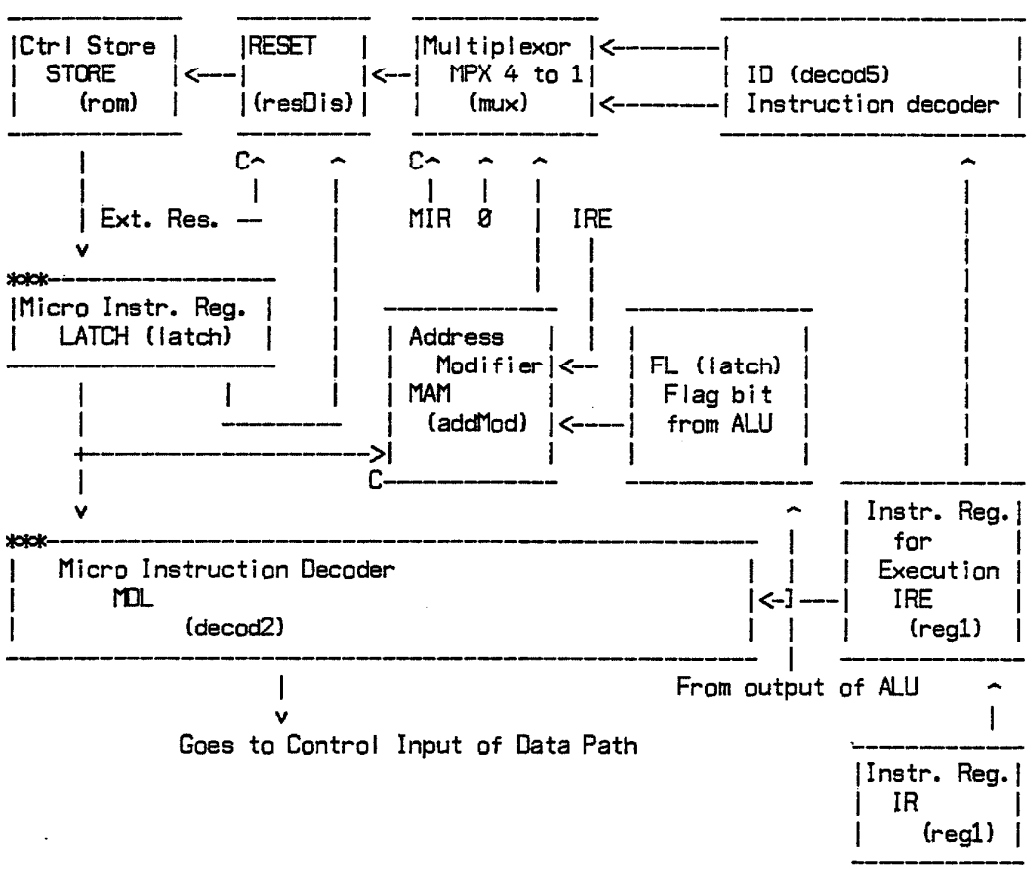
RTsim Assembler Listing for multiply.asm

```
                #
                #       FILE:  multiply.asm
                #              multiply two unsigned numbers
                #

                        .load mem               # load symbol table

                        .loc 0
0:                      shiftAndSum             # put in the entry point
1;      :negl   .word $ffff             # put in a -1

                        .loc $20
                #   multiply two number in R0 and R1, put result back in R3
                :shiftAndSum
32:                     zero R3                         # use R3 as sum
33:                     zero R5                         # absolute reference
34:                     load R6 D R5
35:                     addr2
36:                     load R7 D R5
37:                     negl
38:                     load R4 D R5
39:                     count
40:                     load R2 D R5
41:                     exit
42:     :loop2  add R7 R4               # decrement counter
43:                     br N R2                 # goto addr2, if -ve
44:                     add R3 R3               # shift R3 left by one bit
45:                     add R0 R0               # shift left one bit
46:                     br NC R6                # goto loop2 if not carry
47:                     add R1 R3               # sum R1
48:                     br R6                   # goto loop2 if not zero
49:     :addr2  loop2                   # put in the loop address
50:     :exit   addr2                   # put in addr2 address
51;     :count  .word 16
```

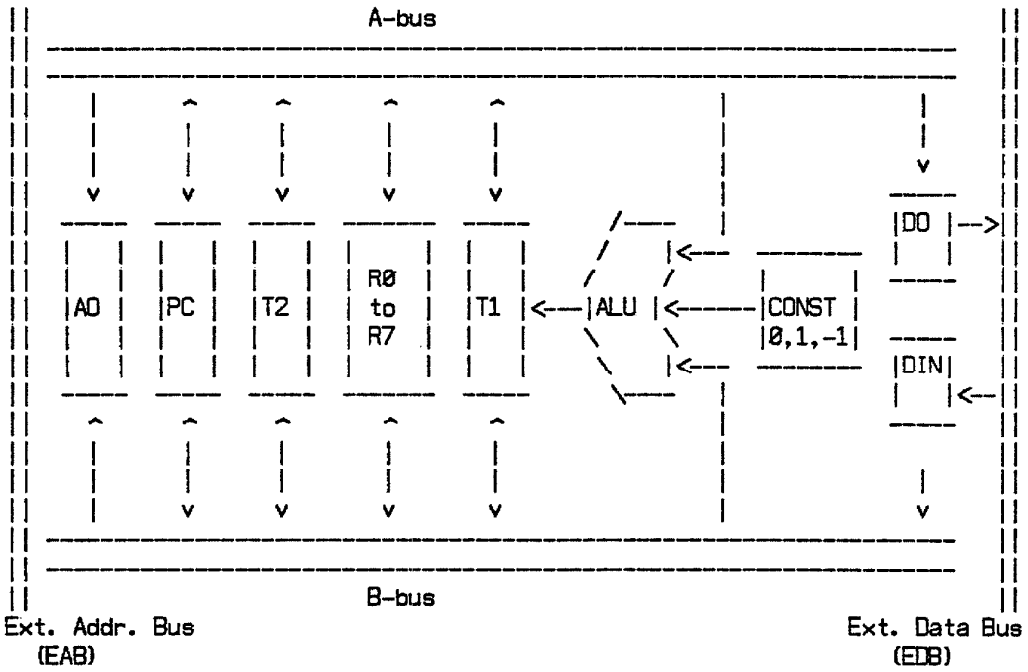
Appendix G - Block diagrams for MIN processor

1) Upper case word = Instance name
2) Word in parentheses = RTD element name
3) C = Control input
4) Blocks with *** connects to CLK(clock2), a 2 phase clock generator
5) Data Path is 16 bits wide

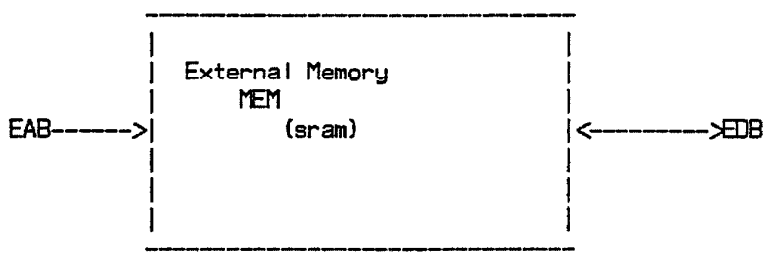
Control Unit:



Data Path:


A-bus
AD
PC
T2
R0
to
R7
T1
ALU
CONST
0,1,-1
DO
DIN
B-bus
Ext. Addr. Bus
(EAB)
Ext. Data Bus
(EDB)

External Memory:



Appendix H - A sample run

```
RTsim Terminal Session
~ take &nrmin.com
~ #
~ # This is a sample architecture contains in Appendix of RTsim User Guide
~ #     '&nrmin.dia' contains the block diagram of the architecture
~ #
~ # All files preceded by '&' could be found in the rtsim directory
~ #
~ switch base 16
~ lib &ele.lib
~ parse &nrmin.rtd
pass 1
pass 2
exec(17)-W: only 10 out of 11 bits have been connected in instance RESET
exec(17)-W: a total of 1 bits not connected
~ assemble &nrmin.asm
pass 1
pass 2
~ assemble &multiply.asm
pass 1
pass 2
~ force reset 1 *
~ subcycle 2
Phase 0
~ force reset 0 *
~ translate mir store
~ translate store *
Processing STORE
~ translate mem
~ watch mir,mem
~ # Try some simulation commands like, subcycle or step, etc.
End of ps:<ssp.rt-sim>nrmin.com
~ comment : watch on MIR, the micro instruction register, MEM, memory
~ cycle 3
 MIR    | Ctrl:0 Input:1C20030E Output: [0: :reset and zero rese1]
 MEM    | Ctrl:0 Input:XXXXXXX Output:ZZZZ
 MIR    | Ctrl:1 Input:000E2310 Output: [D: :rese1 ldd T1 A0 rese2]
 MEM    | Ctrl:0 Input:XXXXXXX Output:ZZZZ
 MIR    | Ctrl:0 Input:000E2310 Output: [D: :rese1 ldd T1 A0 rese2]
 MEM    | Ctrl:2 Input:000XXXX Output: [shiftAndSum]
 MIR    | Ctrl:1 Input:09200511 Output: [E: :rese2 DIN PC brzz3]
 MEM    | Ctrl:0 Input:000XXXX Output:ZZZZ
 MIR    | Ctrl:0 Input:09200511 Output: [E: :rese2 DIN PC brzz3]
 MEM    | Ctrl:0 Input:000XXXX Output:ZZZZ
~ break ire 0034
~ comment : set a break point at IRE with bit vector 0034 (Hex)
```

```
~ break
1 break points:
1: IRE 0034
~ examine store 0-1f
0:      [:reset      and zero rese1]
1:      [:adrm1      ldd Ry T2A0 RB brzz3]
2:      [:abdm1      ldd PC A0 add plus1 abdm2]
3:      [:oprr1      Rx nop Ry op oprr2]
4:      [:ldrr1      ldr PC A0 Ry RxT2 add plus1 ldrm2]
5:      [:tstr1      Rx test zero brzz3]
6:      [:strr1      ldr PC A0 Rx RyT2 add plus1 ldrm2]
7:      none set
8:      [:push1      Ry add minus1 push2]
9:      [:popr1      ldd Ry A0 add plus1 popr2]
A:      [:zero1      ldr PC T2A0 and zero zero2]
B:      none set
C:      none set
D:      [:rese1      ldd T1 A0 rese2]
E:      [:rese2      DIN PC brzz3]
F:      [:brzz1      ldr Ry A0 add plus1 BC brzz3]
10:     [:brzz3      ldr PC A0 add plus1 brzz2]
11:     [:brzz2      ldre T1 PC RD reset]
12:     [:oprm2      T1 DO T2 A0 brzz3]
13:     [:oprm1      Rx nop DIN op oprm2]
14:     [:ldrm1      ldr PC A0 DIN RxT2 add plus1 ldrm2]
15:     [:test1      ldr PC A0 DIN T2 add plus1 ldrm2]
16:     [:strm1      Rx DO T2 A0 add zero brzz3]
17:     none set
18:     [:abdm2      T1 PC abdm3]
19:     [:abdm3      Ry nop DIN add abdm4]
1A:     [:abdm4      ldd T1 T2A0 RB brzz3]
1B:     [:oprr2      ldr PC A0 T1 Ry add plus1 brzz2]
1C:     [:popr2      T1 Ry DIN Rx brzz3]
1D:     [:push2      Rx DO T1 RyA0 brzz3]
1E:     [:zero2      T2 nop T1 Rx add plus1 brzz2]
1F:     [:ldrm2      ldre T2 nop T1 PC add zero RD reset]
~ comment : examine the content of control store, range 0 - 1f
~ # etc...
~ close
Log file sample.run is closed.
```

Appendix I - An NDL example

```
begin "zdreg"
sourceFile "ndl.mi";                    # file in rtsim directory
sourceFile "nmoslb.mi";
element(zdreg);

#       This is the NDL replacement for the RTD element in appendix A

net(dreg(vector ctrl; node inbusA,inbusB,outbusA,outbusB,inNode; iName));
#       A 1-bit Dual port register
node outNode,mid;
beginNet
        small(outNode); small(mid);
        Ntrans(strong,ctrl[0],inNode,outNode);  # ctrl[0] = refresh
        Ntrans(strong,ctrl[1],inbusA,inNode);   # ctrl[1] = ld_A
        Ntrans(strong,ctrl[2],inbusB,inNode);   # ctrl[2] = ld_B
        Ntrans(strong,ctrl[3],outNode,outbusA); # ctrl[3] = rd_A
        Ntrans(strong,ctrl[4],outNode,outbusB); # ctrl[4] = rd_B
        inv(inNode,mid);
        inv(mid,outNode);
endNet;

net(register(vector ctrl,inbus,outbus,inNode; integer width; iName));
integer i;
beginNet
        for i:=0 upto width-1 do    # form a n bits dual port register
              dreg(ctrl,inbus[i],inbus[i+width],
                  outbus[i],outbus[i+width],inNode[i],cvs(i));
endNet;

procedure generate(integer noOfCtrl,noOfIn,noOfOut);
#       this procedure generates the NDL network
#       dataIn and dataOut are the same
begin
        vector ctrlIn,dataIn,dataOut,inNode;
        smallVec(ctrlIn,0,noOfCtrl-1);    # create the control vector
        largeVec(dataIn,0,noOfIn-1);      # create the input data vector
        dataOut:=dataIn;                  # make data input = data output
        keepVecRev(ctrlIn);               # keep symbolic names
        keepVecRev(dataIn);               # NOTE: ctrlIn,dataIn,dataOut
        keepVecRev(dataOut);              #       must be kept.
        smallVec(inNode,0,noOfIn div 2 - 1);
        keepVecRev(inNode);               # keep internal name
        register(ctrlIn,dataIn,dataOut,inNode,noOfIn div 2,ignoreP);
end;

end "zdreg";
```

Appendix J - Runtime

Switch	Network	Total	RTsim	MOSSIM	subcyc.	time/sub.
-	0	43.9	43.9	-	551	0.080
-	1	65.5	45.543	19.957	551	0.119
-	8	209.6	56.305	153.295	551	0.380
Cap. on	0	64.0	64.0	-	551	0.116
Catch X	0	47.9	47.9	-	542	0.088

The above chart tabulates the runtime of the MIN processor architecture outlined in appendix G. All times are measured in DEC system 2060 CPU seconds. The "network" column in the chart shows the number of registers replaced by NDL networks, where each network consists of 87 nodes and 144 transistors. Refer to appendix I for the NDL code. In the third row, register R[3] in the data path is replaced by an NDL network. In the fourth row, registers R[0] to R[7] in the data path are replaced by NDL networks. The "MOSSIM" column in the chart represents the CPU time spent by MOSSIM and the "RTsim" column represents the CPU time spent by RTsim and the RTsim-MOSSIM interface. The "switch" column represents various RTsim switches which are turned on. The "cap. on" switch models MOS capacitance; the "catch X" switch catches X-bit generation during simulation. Refer to section 2 for details.

Appendix K - Installation

The source and documentation files of RTsim and its supporting programs consist of the following (star (*) is the filename wildcard character):

ALLFILES	A list of all files
*.m	MAINSAIL module files
*.fun	RTD element module files
*.mi	MOSSIM interface files
*.h	RTsim interface files
*.s	Interrupt routines
*.mcl	MAINSAIL compiler command files
*.mlb	MAINSAIL librarian command files
*.com	RTsim command files
*.doc	Documentation files

All files must reside in a single directory, hereafter referred to as the "RTsim directory".

The files "rtsim.h" and "syspro.m" must be edited to incorporate system dependent changes. The variable "homePath" in "rtsim.h" must be set to the name of the RTsim directory. Versions of interrupt routines for the DEC-20 (Tops20), VAX (Unix Version 7) are available by setting one of the variables "isDec20" or "isVax" in "rtsim.h" to one. The variable "isGeneral" can be set to one for other systems to disallow interrupts. The files "ele.idx", "ele.hdr", and "ele.doc" are used by the RTsim documentation system. Their protection modes should be set to allow public read/write access.

If switch-level simulator MOSSIM II is available, the variable "hasMossim" in "rtsim.h" should be set to one to enable the RTsim - MOSSIM interface routines. The file "sysmod.mi" should also be edited according to instructions for MOSSIM II.

Once "rtsim.h" and "syspro.m" are updated, the MAINSAIL modules comprising the RTsim programs may be compiled. This is accomplished by running the MAINSAIL Executive and executing the compiler COMPIL with the following commands where <cr> means carriage return:

```
*compil,
>cmdfile rtsim.mcl
><cr>
```

If MOSSIM II is available and "sysmod.mi" is updated, the MAINSAIL modules comprising the RTsim - MOSSIM II interface may be compiled.

```
*compil,
>cmdfile mossim.mcl
><cr>
```

After all of the modules are compiled (and possibly assembled), the Module library files containing the object modules may be created by executing the MAINSAIL Module Librarian MODLIB with the following commands:

```
*modlib,
MODLIB: read rtsim.mlb
MODLIB: <cr>
```

If MOSSIM II is available, the MOSSIM module library "mossim.lib" should be copied from the MOSSIM directory to the RTsim directory. Then, the MAINSAIL Module Librarian MODLIB should be invoked with the following commands:

```
*modlib,
MODLIB: read mossim.mlb
MODLIB: <cr>
```

On some systems, the default number of pages reserved for module storage in library files is insufficient, and MODLIB will return a library full error message. In this case, the user must specify a larger number of pages to reserve as an argument to the "create" command in the appropriate "*.mlb" file. Please refer to the Module Librarian section of the MAINSAIL Utilities User Guide for assistance.

Finally, the executable bootstrap files for RTsim may be created by executing the MAINSAIL configurator CONF with the following commands where the name of the RTsim directory is substituted in place of the string <RTsim directory>:

```
*conf
CONF: bootfilename rtsim.s
CONF: commandstring
rtsim,
preloadlibrary <RTsim directory>rtsim.lib
preloadlibrary <RTsim directory>mossim.lib          (if available)
<cr>
CONF: globalprocedures
flagaddr                        (if isDec20 is 1)
_sigsys                         (if isVax is 1)
_interrupt                      (if isVax is 1)
```

```
_write_address                (if isVax is 1)
_times                        (if isVax and hasMossim are 1)
<cr>
CONF: <cr>
```

The resulting bootstrap file "rtsim.s" may then be assembled and linked to the interrupt routines in "dec20int.s" if "isDec20" is set, or "vaxint.s" if "isVax" is set. The user should refer to the host system's MAINSAIL User Guide for assistance. The resulting executable file may then be copied to the host system's public directory.

With the executable bootstrap file, we can now regenerate the on line documentation systems with the following commands:

```
rtsim
~ take regenerate.com
~ quit
```

The file "message" may be edited to incorporate system dependent messages, which will be printed when RTsim starts up.

References

[1] R. Bryant, M. Schuster, D. Whiting, MOSSIM II: A switch-level simulator for MOS LSI User's Manual, Computer Science Dept., Caltech, Jan. 1983.

[2] M. Barbacci, An introduction to ISPS, in Computer Structures: principles and examples, 1982.

[3] R. Bryant, A switch-level simulation model for integrated logic circuits, Laboratory for computer science, M.I.T., March, 1981.

[4] N. Tredennick, How to flowchart for hardware, computer, Dec. 1981.

[5] W. Cory, Symbolic simulation for functional verification with ADLIB and SDL, 18th Design Automation Conference, 1981.

[6] W. Sherwood, A hybrid scheduling technique for hierarchical logic simulators, 16th Design Automation Conference, 1979.

[7] D. Hill, W. vanCleemput, SABLE: a tool for generating structured, multi-level simulations, 16th Design Automation Conference, 1979.

[8] C. Wilcox, M. Dagenforde, G. Jirak, MAINSAIL language reference manual, Xidak, 1979.