THOR USER'S MANUAL: TUTORIAL AND COMMANDS

Robert Alverson
Tom Blank
Kiyoung Choi
Sun Young Hwang
Arturo Salz
Larry Soule
Thomas Rokicki

Technical Report: CSL-TR-88-348

January 1988

This work was supported by Defense Advanced Research Projects Agency, Contract No.MDA903-83-C-0335.

THOR USER'S MANUAL: TUTORIAL AND COMMANDS

Robert Alverson, Tom Blank, Kiyoung Choi, Sun Young Hwang, Arturo Salz, Larry Soule, and Thomas Rokicki

Technical Report: CSL-TR-88-348

January 1988

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

THOR is a behavioral simulation environment intended for use with digital circuits at either the gate, register transfer, or functional levels. Models are written in the CHDL modeling language (a hardware description language based on the "C" programming language). Network descriptions are written in the CSL language supporting hierarchical network descriptions. Using interactive mode, batch mode or both combined, a variety of commands are available to control execution. Simulation output can be viewed in tabular format or in waveforms. A library of components and a toolbox for building simulation models are also provided. Other tools include CSLIM, used to generate boolean equations directly from THOR models and an interface to other simulators (e.g. RSIM and a physical chip tester) so that two simulations can be run concurrently verifying equivalent operation.

This technical report is part one of two parts and is formatted similar to UNIX manuals. Part one contains the THOR tutorial and all the commands associated with THOR. Part two contains descriptions of the general purpose functions used in models, the parts library including many TTL components, and the logic analyzer model. For first time users, the tutorial in the first report is the best starting place; additionally, the THOR(1) man page is the root of the documentation tree in that all other documents are referenced there.

Key Words and Phrases: behavioral simulation, functional model, hierarchical network description

ACKNOWLEDGMENTS

Many acknowledgments are necessary since THOR's heritage is long. Specifically, the evolution started from the CSIM program at the University of Colorado, where Professor Mike Lightner, and Henry and Beverly Vellandi created the core system and documentation. At Stanford, we've rewritten and greatly extended the original work through the labors of Robert Alverson, Professor Tom Blank, Kiyoung Choi, Dr. Sun Young Hwang, Arturo Salz, Larry Soule and Thomas Rokicki.

DISTRIBUTION TAPE

A copy of the distribution tape can be obtained by writing or calling:

Software Distribution Center
Office of Technology Licensing
Stanford University
350 Cambridge Avenue
Palo Alto, CA 94306

Telephone: (415) 723-0651

1. Introduction

This tutorial is intended for the first time user of the THOR simulation tools. THOR runs under the UNIX operating system. The tutorial assumes that the user is familiar with basic concepts of UNIX, such as directories, files, environment variables, and input/output redirection. Further, the user should understand the C programming language.

The goal of this tutorial is to help the new THOR user understand the following:

- What is functional simulation?
- How are hardware designs described?
- How are input stimuli vectors specified?
- How are signals in a circuit monitored?
- How is the simulator run in batch and interactive modes?
- How can a circuit model be created and incorporated into the simulator?
- How to interface to other simulators?

1.1. Historical Perspective

THOR is a functional simulator written by the VLSI/CAD Group at Stanford University to aid hardware designers to verify their designs. It is based on the CSIM simulator, a conventional event-driven functional/behavioral simulator developed by the VLSI/CAD Group of the University of Colorado, Boulder. THOR was developed to provide hardware designers with an interactive, efficient simulation tool.

1.2. Definitions and Conventions

Some terms are used consistently in this tutorial:

- ***Element*** means an instantiation of a primitive functional model in a circuit, regardless of its level of description. It can be a simple logic gate, whose functional behavior is provided by the system, or a more complex functional block, whose model is provided either by the system or by the user.
- ***Pin*** or ***port*** is a connection point to an element, through which a logic signal flows in and out of the element.
- ***Structure*** or ***topology*** of a circuit is used to represent the interconnection of elements. This can be visualized as a block diagram in which each block is an element and the lines connecting the blocks determine the interconnections.
- ***Nodes*** or ***nets are*** commonly used to refer to the interconnections.
- ***Behavior*** means the response of an element to input stimuli. Behavior can be described for both an element and a network of elements. For each element, the output is simply a function of its inputs and/or its state variables. The mapping of the inputs and state variables to the outputs is defined as the behavior of the element. The behavior of a network is the response to the stimuli of all the elements taken as a whole. The response is determined by a combination of each element's behavior and the interaction with other elements.

Throughout this tutorial, the following conventions are used:

- ">" indicates a prompt displayed on the CRT screen.

- Any terms that appear in a square bracket "[]" are optional.
- Terms in a curly bracket "{}" are selectively used, i.e., {a|b} means a or b.

2. Overview of THOR

This section describes the basic concepts of functional simulation, and the operation of the THOR simulator.

2.1. Why Use and What is a Functional Simulator?

When a hardware design is newly developed, its function needs to be verified. One way to verify the operation of the circuit is to build the actual hardware or prototype, exercise the test vectors and monitor the results. This is costly and time consuming. If we can verify the functions of the design in some faster and cheaper way before the prototype is built, the debug cycle time and cost will be significantly reduced. Functional simulation is one of the most effective methods of design verification. The cost of functional simulation is much less than that of hardware breadboarding. It is easy to prepare input stimuli for functional simulation and to debug and modify the designs. Moreover, the components that are not available at the time of design can be simulated by correctly modeling them.

A functional simulator is a tool for verifying a hardware design by exercising it on a computer. Usually, the hardware descriptions are supplied to the simulator with the description of input stimuli, then the simulator exercises it and gives the result.

2.2. What is THOR?

THOR is a mixed level event-driven simulator with which one can simulate hardware at the behavioral level, register transfer level, **and/or** gate level. It provides all the features required for functional simulation, including generation of input stimuli, monitoring of output results, modeling capability, and interactive simulation control.

THOR also provides an interface to a switch-level simulator, RSIM, and a ***physical tester.***

In **the THOR** system, internal network states are represented by four values: 0 (logical Low), **1** (logical High), **U** (Undefined), and **Z** (Floating or High Impedance).

2.3. Operation of THOR

To simulate a hardware &sign, the user should first prepare a description of the hardware, input stimuli (test vectors), and information on where and how to monitor the simulation results. Then, the information **is** linked with libraries by the **THOR** system to generate an executable simulator by using the **gensim** command. **Figure 1** shows the general view of the THOR simulator. The executable simulator generated by **the gensim** command, can be run without interaction with users ***(batch mode)*** or can be controlled by the user with commands given interactively or through a command file ***(interactive mode).*** The detailed description of those commands can be found in section 6.

2.4. Input Components to THOR System

As mentioned in the previous section, the user should supply the circuit description, input stimuli, and **monitor information to the THOR** simulator. The circuit description consists of the models of circuit elements and their interconnections. Input stimuli and monitors are usually modeled as circuit elements called ***generator*** elements and ***monitor*** elements, respectively.

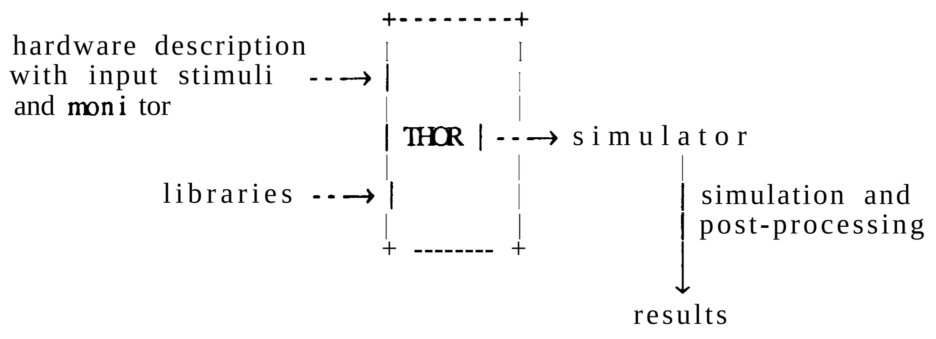

Figure 1. General view of **THOR** simulator

2.4.1. Functional Model

A model describes the behavior of a primitive ***functional*** element. Each model can be used in more than one place in the circuit. In the **THOR** system, models are written in a language called **CHDL (C** Hardware Description Language). **CHDL** is based on the C programming language with added features for hardware modeling. Besides the behavior of a functional element, other descriptions can be included in a functional model, such as port descriptions and initialization routines. ***Generator*** and ***monitor*** elements can also be modeled in the same way as other ***functional*** elements.

Many models are provided in the **THOR** library, which include generic logic gates **(NAND, OR, XOR** gates, etc.), **various TTL** parts, ***generator*** elements **(Up/Down Binary Counters, Clocks** with variable period and phases, etc.), and ***monitor*** elements.

2.4.2. Net List

In **the THOR** system, a ***component-oriented*** net list language, called **CSL,** is used to specify the connections where each statement in the net list specifies an element and the connections of its ports to nodes in the circuit.

Each functional element has various attributes and **CSL** allows each of these attributes to be specified for each element. **CSL** also allows hierarchical description, where a group of elements can be defined and treated as ***an*** entity called a ***subnetwork.*** Each statement in the net list specifies a model (or ***subnetwork)*** name, instance name, all the input, output, and biput connections, and instance-specific values for each element such as output delays and initial values.

2.5. Interfaces

Other design tools **can** be accessed in the **THOR** simulation environment through procedural interfaces. This section describes those interfaces to **other** utilities and &sign tools.

2.5.1. Simview

Simview takes the simulator outputs generated by **the THOR** simulator, and generates user readable data in tabular format.

2.5.2. Analyzer, Banalyzer

Analyzer is **a** monitor program which converts the data generated by the **THOR** simulator into a graphic display.

Banalyzer is a monitor which writes the states of its inputs to a file so that they can be viewed later in graphic forms using the graphic analyzer (program called **ana**) or in a table format using **aview.**

2.53. CSLIM

Some functional models can be written to automatically generate hardware implementations using **CSLIM.** Though it is very hard to synthesize the hardware with arbitrary structures, hardware with regular structures, like **PLAs** can be synthesized with less difficulty. **CSLIM** takes a finite state machine **model** written in a subset of **the THOR** modeling language, **CHDL,** and generates **the PLA** equations which can then be optimized using **espresso.**

2.5.4. Interface to RSIM and Physical Tester

As mentioned before, **THOR** supports the simulation of the circuits with abstraction levels from gate level to **behavioral** level. Even though it does not provide the features for switch level simulation, it provides **a** procedural interface to a switch level simulator **RSIM, so** that **the THOR** functional specification can be used to verify a design at the switch level. Furthermore, verification can be extended to a physical chip tester.

3. Getting Started

In this section, more detailed descriptions are presented on how to simulate a hardware design using the **THOR** simulator.

3.1. Preparing a Net List File

To simulate a hardware design, a net list file describing the top level interconnections of functional elements must be presented to the simulator. Each constituent functional block can be a net list file or a functional module whose behavior is written in **CHDL.**

Each statement in the net list file corresponds to a single element or a subnetwork, and is comprised of various fields specifying the element attributes. The detailed descriptions of these attributes can be found in section 5.

For our first example, we will simulate a circuit which performs the ***exclusive or*** operation. We can implement it with one **XOR** gate whose behavior can be found in the model library, and the input stimuli can be generated using a generator element **(Figure 2). The** net list description for this is:

```
(g=CLOCK)(n=gen1)(o=in1)(s=3)(vs=0, 1, 2);        (1)
(g=CLOCK)(n=gen2)(o=in2)(s=3)(vs=0, 2, 4);        (2)

(f=xor)(n=xor1)(i=in1, in2)(o=out)(do=0);         (3)

(m=BINOUT)(n=mon l)(i=in 1);                      (4)
```

```
                +------+
                | mon1 |
                +------+
                    ↑
                    |
                    |
+------+   in1      |     +--------------+
| gen1 | -----+---→       |              |
+------+                  | XOR gate     | out  +------+
                          | Hardware to  | ---→ | mon3 |
+------+   in2            | be simulated |      +------+
| gen2 | -----+---→       |              |
+------+      |           +--------------+
              |
              |
              ↓
          +------+
          | mon2 |
          +------+
```

Figure 2. Block diagram for simulating **XOR** gate

```
(m=BINOUT)(n=mon2)(i=in2);                    (5)
(m=BINOUT)(n=mon3)(i=out);                    (6)
```

Statements (1) and (2) specify that a **CLOCK** generator is used to provide input stimuli to the circuit. The output port of the clock element named 'genl' is connected to a net named 'inl'. It has three internal states which specify the time to start generating stimuli, the time of transition within one period, and the period of the clock, respectively. The values of these state variables are initialized to 0, 1, and 2 (vs=0, 1, 2). Statement (2) specifies another clock instance 'gen2' driving the node 'in2' with period 2. See **gen(3)** for details of generator elements.

Statement (3) describes the circuit which consists of one **XOR** gate with instance name 'xorl'. It has two input ports connected to nodes 'inl' and 'in2', and one output port connected to a node 'out'. The gate has 0 delay. (The default value of the output delay is one.) For more complex hardware, the description could have many lines rather than one.

Statements (4), (5), and (6) specify monitors, which monitor the states at nodes 'inl', 'in2', and 'out' in binary format. The monitored states are normally redirected to a file for further processing. See **mon(3)** for details of monitor elements.

3.2. Initial Setup

To run the simulator, the core simulator and model libraries need to be linked with user models and net lists. To locate these files, gensim reads values of the following environment variables:

THOR :	root directory
THORLIB :	library files
THORBIN :	object files
THORINC :	include files

If some of these variables are not set, default values are used for them. Default values are as follows:

THOR :	**/projects/cad**
THORLIB :	**THOR/lib**
THORBIN :	**THOR/bin**
THORINC :	**THOR/lib/csim/include**

The THOR system is normally installed in the default directories and the user does not need to set the environment variables. However, the user must put in his path the directory name '/projects/cad/bin' where **gensim is installed:**

```
set path = ($path /projects/cad/bin)
```

3.3. How to Run

After setting the environment variables, an executable simulator can **be** generated by the **gensim** command as follows:

```
gensim root.csl -t time
```

where **root.csl** specifies the net list file of the circuit to be simulated, and **time specifies the** number of simulation time steps. The name of the net list file should be extended with **.csl. The gensim** command will generate an executable file **root.exe,** and the monitored network states are written into the file **root.out,** if not redirected.

Following **is** the sample simulation session for the circuit described in section 3.1. Here, **xor.csl** is the filename of the net list. We want to run the simulator for 3 time steps.

```
> gensim xor.csl -t 3


  .
xor.exe -t 3 > xor.out
> cat xor.out
3
0 mon3 0
0 mon2 1
0 monl 1
1 mon3 1
1 monl 0
2 mon2 0
2 monl 1
3 mon3 0
3 monl 0
```

Without any processing, the result is difficult to read. The command **simview** does some processing on the result to make it much easier to read.

```
> simview xor.out
THOR  OUTPUT

Time units
|      mon3
|      |  mon2
|      |  |  mon1
|      |  |  |
0      0  1  1
1      1  1  0
2      1  0  1
3      0  0  0
```

The simulation can be done interactively by using **the** option **-i** to **the gensim** command. If the executable simulator has already been generated, the **-i** option can be applied when the simulator **is** executed:

```
> xor.exe -i > xor.out
thor $ : 0> enable in1 in2 out
thor $ : 0> step
(out 0)
(in2 1)
(in1 1)
thor $ : 1> step
(out 1)
(in2 1)
(in1 0)
thor $ : 2> step
```

```
(out 1)
(in2 0)
(in1 1)
thor $ : 3> step
(out 0)
(in2 0)
(in1 0)
thor $ : 4> exit
>
```

The **enable** command displays the values of the nodes given as its arguments whenever the simulation stops. The **step** command executes the simulation for one time step. More explanations on ***interactive mode*** **will** be done in section 6.

3.4. Summary

Following is the summarized procedure of a simulation run, assuming that all the models are supplied from the library.

1) Create a net list file: **foobar.csl**

2) **gensim foobar.csl -t 3**

3) **simview foobar.out**

4. Writing Models

In order for hardware designers to simulate the behavior of a hardware function, the functionality of the hardware must first be modeled. The basic concepts needed to create a model using the THOR modeling language and an example of a basic model are described in this section.

4.1. THOR Modeling Language, CHDL

The **THOR** modeling language, **CHDL,** is based on the C programming language but has constructs that have been added to make modeling easier. Any statements in C can be used to model the behavior of an element in **CHDL.** Each model consists of a model statement and three sections: interface, initialization, and behavioral description sections, as shown below.

```
Model statement
{
    Interface section

    Initialization section

    Behavioral description section
}
```

The files for the models must have names which end with **.c.** If the files are in the present working directory, then the user does not need to compile or specify the file name. The command **gensim** automatically compiles and links them. See **gensim(1)** for details.

4.1.1. Model Statment

The first statement of the model is the model statement which consists of the keyword **MODEL** followed by **the name of the model in** parentheses **(e.g. MODEL(xor)).** This name is to be used by **THOR** to identify the model. It is the same name used in the (f=...) field of the CSL description of the model.

4.1.2. Interface Section

In **this** section, **the modeler describes the model's interface to the THOR** simulator. The interface consists of inputs, outputs, biputs (bidirectional i/o), and states. State variables are used by the model to remember values from previous evaluations. They can be viewed as hardware registers or memories.

Element inputs are through the input ports and/or biput ports and outputs are through output and/or biput ports. Outputs and next states are generated based on the inputs and current states. These ports and states are declared in this section of the model of the element.

Data types are used to describe the format of the interface values used as inputs, outputs, biputs, and states. The following data types are allowed:

SIG(name) declares a single port called 'name'.

GRP(name, **n[, {MSB0|LSB0}])** declares an n-bit group of related signals.

VGRP(name) declares a virtual group of related signals.

TSIG(name) declares a local signal.

TGRP(name, n[, **{LSB0|MSB0}]**) declares a group of related local signals.

A SIG data type is used when the required interface is a single wire or signal. If a group of wires or a bus is needed the GRP data type is used (A GRP can also be called a REG or BUS; likewise, VBUS or VREG can be used for VGRP). The width of a group of the signals is specified by the parameter 'n'. The signals in a group are indexed and ordered from 0 to n-l. For example, GRP(inp, 3) means that it has three signals, **inp[0],** inp[l], and **inp[2],** and the first signal is **inp[0].** One optional argument specifies bit ordering of the group when it is used as an integer; LSBO (MSBO) means that the least (most) significant bit corresponds to the bit with lowest index. Default is LSBO. Another optional argument, DESEN, allows the signal or bus to be treated as desensitized. The model code is only invoked when a sensitized variable has changed. Thus, DESEN can be used to inhibit calling a model when, for example, one of the data inputs to an edge triggered latch changes. The model will not be called until a sensitized variable, such as the clock in this case, is changed. The optional arguments follow the bus width in a group definition or the signal name in a signal definition. For the case of a bus definition, the order of LSBO (MSBO) and DESEN does not matter.

A VGRP is a group of signals whose size is unknown at the time of model compilation. This is very useful for describing generic models (such as a general PLA) that can accept a variable width data type. The width of the group of signals is determined by the net list file.

There is an optional argument available which specifies the bit ordering of the group when it is used as an integer; **LSBO (MSBO)** means that the least (most) significant bit corresponds to the bit with lowest index. Default is LSBO. Another optional argument, **DESEN,** allows the signal group, or vgroup to be treated as desensitized. The model code is only invoked when a sensitized variable has changed. Thus, DESEN can be used to inhibit calling a model when, for example, one of the data inputs to an edge triggered latch changes. The model will not be called until a sensitized variable, such as the clock in this case, is changed. The optional arguments follow the bus width in a group definition or the signal name in a signal definition. For the case of a bus definition, the order of LSBO **(MSB0)** and DESEN does not matter. The order of declarations for each data type is important. Note that it should match the order of the signal ports in net list description. Note also that only one VGRP is allowed in each data type declaration and it must be the last declaration.

TSIG and TGRP are used in the same way as SIG and GRP, except that they are used only for local signals. These local signals are not visible from outside world.

The interface section consists of subsections for the declaration of inputs, outputs, biputs, and states. Each subsection starts with the keyword **IN-LIST, OUT-LIST, BI_LIST,** or **ST-LIST,** respectively, and ends with **ENDLIST.** Inside each subsection are declarations for those ports or states with appropriate data types. Not all data types are needed when describing the interface of the model. The modelers should use only those types that are required for the model being described.

4.13. Initialization Section

Before the simulation starts, some models may need initialization for state variables or internal arrays, or to read in configuration files. This initialization can be done in this section of the model. This section begins with the keyword **INIT** and ends with the keyword ENDINIT. Any standard C statements can be used inside this section for initialization. This section is executed once for each model instantiation.

4.1.4. Behavioral Description Section

The behavioral description section is the main body of the model containing the algorithms or descriptions that the modeler is trying **to construct..** This section of the model is executed whenever an input or biput changes. The code should generate new outputs, biputs, and state variables based on its current state and inputs. It comes after interface and initialization sections. All standard C statements are allowed. However, to describe the behavior of the hardware efficiently, constructs are needed that allow data to be moved and changed when the model is evaluated. Capabilities that are not provided by the C programming language, such as data formatting and conversions are provided to the modeler. A list of functions available can be found in the **THOR(l)** man page, and the detailed descriptions of each function can also be found in its own man page.

The final statement before exiting the behavioral description section must be EXITMOD(value); a non-zero ‘value’ indicates an error. Any number of EXITMOD(value) statements can be used.

4.2. Examples

4.2.1. Example 1

Here is a basic model of an exclusive or gate.

```
/*
 *        two-input xor gate
 */

MODEL(xor)
{
 /* Interface section */
   IN_LIST
        SIG(in0);
        SIG(in1);
   ENDLIST;

   OUT_LIST
        SIG(out);
   ENDLIST;

 /* Behavioral description section */
   int i;                    /* Declaration of a variable for internal use */

   out = ZERO;

   if(((inO == ONE) && (in1 == ZERO)) || ((in0 == ZERO) && (in1 == ONE)))
        out = ONE;

   EXITMOD(0);
 }
```

The xor gate has no biput ports and states so no declarations are needed for them in this model. It has no initialization section, either.

4.2.2. Example 2

Here is a model of a latch which makes use of the DESEN option:

```
MODEL(xx273)
{

IN-LIST
        SIG(Clr_b);
        SIG(Clk);
        GRP(InDat, 8,DESEN);
ENDLIST;

OUT-LIST
        GRP(OutDat, 8);
ENDLIST;

ST-LIST
        GRP(SavDat, 8);
        SIG(LastClk);
ENDLIST;

        register int i;

                        /* check for correct # of inputs, outputs & states */

                                        /* UNDEFINED clock or clear input */
        switch (Clr_b) {
        case ZERO:
                SavDat[] = 0;
                break;
        case ONE:
                if( Clk >= UNDEF ) {
                        fsetword(SavDat, 0, 7, UNDEF);
                } else if ( vrise[LastClk][Clk] == 1 ) { /* rising edge      */
                                                        /* latch inputs */
                        for (i=7; i>=0; --i) {
                                SavDat[i] = vmap[ InDat[i] ];

                        }
                }
                break;
        default:
                fsetword(SavDat, 0, 7, UNDEF);
                break;
        }

        fcopy(OutDat, 0, 7, SavDat, 0, 7);

        LastClk = Clk;          /* update last-clock state (internal state) */

        EXITMOD(0);
}
```

4.2.3. Example 3

Following is a more complicated example which models a 4-bit adder with a 4-bit register **(Figure 3).**

```
MODEL(adder)
{
 /* Interface section */
    IN-LIST
           GRP(in,  4);
           SIG(ld);
           SIG(oe);
    ENDLIST;

    OUT-LIST
           SIG(carry);
           TGRP(tout, 5);    /* Local group which is hidden from outside world */
    ENDLIST;

    BI_LIST
           GRP(bus,  4);
    ENDLIST;

    ST-LIST
           GRP(reg, 4);
    ENDLIST;

    int i;                   /* Variables must be declared before initialization section */
```

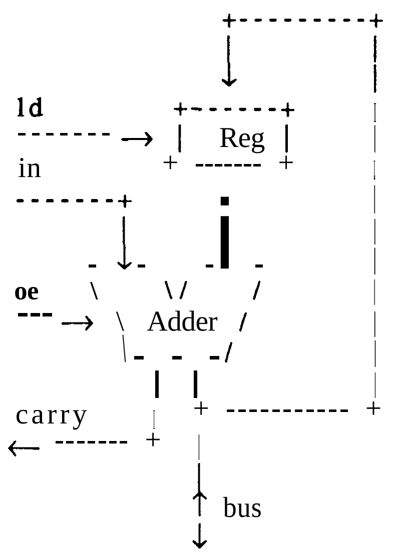

Figure 3.4-Bit Adder with 4-Bit Register

```
/* Initialization section */
   INIT
         reg[3:0] = 0;        /* Initializes state variables */
   ENDINIT;

/* Behavioral description section */
   if(ld == ONE) reg[3:0] = bus[3:0];

   tout[4:0] = in[3:0] + reg[3:0];

   if(oe == ONE)
         bus[3:0] = tout[3:0];
   else
         for(i = 0 ;i <= 3; i++) bus[i] = FLOAT;

   carry = tout[4];

   EXITMOD(0);
```

5. Net List Description

5.1. CSL Format

Net lists **in the THOR** system are described with the language **CSL.** This language describes how the various elements and ***subnetworks*** are connected together. Each **CSL** statement corresponds to a single element or a ***subnetwork.*** A **CSL** statment is comprised of various fields, each field specifying an attribute of an element. The element attributes and their associated field identifiers are shown in Table 1. Note that the three attributes (f=...), (g=...), and (m=...) are selectively used depending on the type of the element and only one of the three can be used in each **CSL** statement.

For example, given a three input **AND** gate that has inputs connected to nodes named ‘a’, ‘b’, ‘c’ and an output connected to a node named ‘d’ and output delay 2. The **CSL** statement for this element would be:

```
(f=AND)(n=gate1)(i=a, b, c)(o=d)(do=2);
```

Note that each field is surrounded by a left and right parenthesis. Each field begins with a field identifier, **‘f=’, ‘n=‘, ‘i=’,** etc. Following the field identifier is information associated to each type of field or attribute.

Anything contained within **‘/*’** and **‘*/’** is ignored. Actually comments are limited to 2000 characters in length due to limitations of lex. So, care should be taken when commenting out large parts of **CSL files.**

CSL supports ***subnetworks.*** A ***subnetwork*** is a group of elements that has a fixed topology and can be referenced as a unit, once defined ***Subnetwork*** can be described in terms of other ***subnetworks.*** A

Attribute	Field
element type	(f=...) for functional element
	(g=...) for generator element
	(m=...) for monitor element
instance name	(n=...)
input ports	(i=...)
output ports	(o=...)
bidirectional ports	(b=...)
initial values	(vo=...) for output ports
	(vb=...) for bidirectional ports
	(vs=...) for states
delays	(do=...) for output ports (default is 1)
	(db=...) for bidirectional ports (default is 1)
number of states	(s=...)

Table 1. Element attributes and associated field identifiers

subnetwork is defined using the following syntax:

```
(sub=...)(i=...)(o=...)(b=...)
{
    CSL statements specifying the structure of the subnetwork.
```

To reference a ***subnetwork, the subnetwork*** type is used in the element type field.

All ***the*** *subnetwork* descriptions can be written into one file with the upper level description. The user can also create files for some or all of the ***subnetwork*** descriptions. Each file should have the same name as ***the subnetwork*** type with an extension '.csl'. In this case, the statement referencing the sub-***network*** should also have the extension '.csl' in its element type field (see the following example).

5.2. Example

For example, the description of xor logic in section 3.1 can be rewritten as follows:

```
(sub=xor)(i=in1, in2)(o=o3);
{
    (f=inv)(n=inv1)(i=in1)(o=in1bar)(do=0);
    (f=inv)(n=inv2)(i=in2)(o=in2bar)(do=0);
    (f=and)(n=andl)(i=inlbar, in2)(o=o1)(do=0);
    (f=and)(n=and2)(i=in1, in2bar)(o=o2)(do=0);
    (f=or)(n=orl)(i=ol, o2)(o=o3)(do=0);
}
```

It can be referenced using:

```
(f=xor)(n=xorl)(i=a, b)(o=c);
```

and would be expanded to:

```
(f=inv)(n=xor1.inv1)(i=a)(o=xor1.in1bar)(do=0);
(f=inv)(n=xorl.inv2)(i=b)(o=xor1.in2bar)(do=0);
(f=and)(n=xor1.and1)(i=xor1.in1bar, b)(o=xor1.o1)(do=0);
(f=and)(n=xor1.and2)(i=a, xorl.in2bar)(o=xor1.o2)(do=0);
(f=or)(n=xorl.orl)(i=xorl.ol, xor1.o2)(o=c)(do=0);
```

The names of internal nodes and element names have been expanded by concatenating the instance name of the *subnetwork.*

If ***the subnetwork*** description is contained in a separate file, the file name should be 'xor.csl' and it should be referenced as follows:

```
(f=xor.csl)(n=xorl)(i=a, b)(o=c);
```

6. Interactive Mode

6.1. How to Enter the Interactive Mode

The THOR simulator can be run interactively. To enter the ***interactive mode,*** run the simulator with **-i** option.

```
root.exe -i[filename] [-q] [-l] [-c]
```

In ***the interactive mode*** of operation, other options are available. If filename is given after **-i** option, the **file** is read as a **command** file and executed. If **-q** option is given, ***quiet mode*** is set and nothing will be displayed while the ***command file*** is executed. Otherwise, all the command lines and their results are displayed as if the commands were given from keyboard interactively. If -1 option is given, a log ***file*** with an extension of **.log** is generated. If **-c** option **is** given, a ***command file*** with an extension of **.cmd** is generated. Following is an example:

```
> xor.exe -i test.cmd -l -c
```

Initially, **the *command file* test.cmd** is read in and each command in the file **is** executed one by one. If the simulator encounters an **exit (or quit)** command while **reading the command file, it exits. Otherwise, after all the commands from the file have been executed, a prompt is displayed for interactive** commands from the user. All the commands supplied from the command file and from the user directly are written into **the** file named **xor.cmd.** All the lines displayed on the screen are stored into the file **xor.log.**

After entering the ***interactive mode*** of simulation, the system writes a prompt on the terminal. The prompt includes the current ***directory*** path and current simulation time maintained by the simulator. Network hierarchy is considered similar to the file system hierarchy. A file can be considered as an element, or a node, while a ***directory*** can be considered as a ***subnetwork.*** To confine the range of the scope to a particular **subnet, the change** command can be used.

The user can respond to the prompt by entering **a** command followed by option fields. Unambiguous abbreviations are accepted. For example, **st** and **ste** are acceptable as the command **step. The *wildcard*** characters ***** and ? are allowed in arguments. For the detailed description of each interactive command, **see intTHOR(l).**

6.2. Example Session of Interactive Mode Simulation

Following is a gate level description of exclusive or logic mentioned in section 5.2.:

```
/* Top level CSL description */
/* Filename is xortest.csl */
(g=CLOCK)(n=gen1)(o=in1)(s=3)(vs=0, 1, 2);
(g=CLOCK)(n=gen2)(o=in2)(s=3)(vs=0, 2, 4);
(f=xor.csl)(n=xorl)(i=inl, in2)(o=out);


/* CSL description of xor */
(sub=xor)(i=in1, in2)(o=o3)
ι
  (f=inv)(n=inv1)(i=in1)(o=in1bar)(do=0);
```

```
    (f=inv)(n=inv2)(i=in2)(o=in2bar)(do=0);
    (f=and)(n=andl)(i=inlbar, in2)(o=o1)(do=0);
    (f=and)(n=and2)(i=in1, in2bar)(o=o2)(do=0);
    (f=or)(n=orl)(i=ol, o2)(o=o3)(do=0);
}
```

Following is an interactive simulation session of the above example:

```
> xortest.exe -i

thor $ : 0> ena in1 in2 out              ... ena is an abbreviation of enable.

thor $ : 0> dump                           ... Dumps all the enabled nodes.
(out U)                                    ... All the nodes are initialized to
(in2 U)                                        UNDEF by default.
(in1 U)

thor $ : 0> st                          ... Steps one time period.
(out 0)                                    ... Dumps all the enabled nodes and
(in2 1)                                        elements when the simulation stops.
(in1 1)

thor $ : 1> cha xorl                     ... Working directory is changed to '$.xor1'.

thor $.xorl : 1> exa invl.                 ... Examine the element 'invl'.
Element xorl .inv 1 :
    Input pins :
        (input_pin[0-0](indat[0-0]) 1)     ... 'indat' is the name of input pin#0. It has value 1.
    Output pins :
        (output_pin[0-0](outdat[0-0]) 0)

thor $.xorl : 1> conn inv2.              ... Shows the connectivity of the element 'inv2'.
input nodes to element xor1.inv2:
        input_pin[0](indat) --- node (in2 1)
                                           ... Input pin#0 is connected to the node 'in2' whose value is 1.
output nodes from element xor1.inv2:
        (output_pin[0](outdat) 0) --- node (xor1.in2bar 0)

thor $.xorl : 1> conn in2bar             ... Shows the connectivity of the node 'in2bar'.
inputs to node xor1.in2bar:
        xor1.inv2 (output_pin[0](outdat) 0)
                                           ... The node is driven by the output pin#0 of element
                                               'xor1.inv2' with value 0.
pins driven by node xorl .in2bar with value 0:
        xor1.and2 input_pin[1](indat[1])

thor $.xorl : 1> st
(out 1)
(in2 1)
(in1 0)

thor $.xorl : 2> cha ^                  ... Working directory is changed to upper level.

thor $ : 2> bre (out 0)                 ... Sets break point.
thor $ : 2> bre                        ... Shows break points set.
[4] (out 0)

thor $ : 2> go +5                          ... Simulates 5 steps.
break at 3: [4] (out 0)                    ... Break point(node 'out' has value 0)
(out 0)
(in2 0)
(in1 0)

thor $ : 4> cle *                          ... Clears all the break points set..

thor $ : 4> bre

thor $ : 4> mark                           ... Marks current simulation time which is 4.

thor $ : 4> go 7
(out 0)
(in2 0)
(in1 0)
thor $ : 8> restart                      ... Restarts from marked time 4.

thor $ : 4> sho max-time                   ... Shows the value of max-time.
(max-time 1000)

thor $ : 4> set (max-time 7)

thor $ : 4> go                          ... Simulates to max-time.
(out 0)
(in2 0)
(in1 0)

thor $ : 8> exit                           ... Exits.

>
```

7. Tips and Common Errors

7.1. Self Scheduling

During simulation, each element is only evaluated whenever one of its input or biput ports changes, The states and outputs of the element are determined depending on its behavior for the values of its input or biput ports. However, there are elements whose internal states or output values change without any stimuli from external world. For example, the generator element, **CLOCK,** changes its output value periodically, though it has no input ports. To simulate such an element correctly, it is necessary for the model to be able to schedule itself for future evaluation. This can be done by calling the procedure **self-sched** in the behavioral description section of the model. Following is a simplified model of the generator, CLOCK, which shows the usage of self-schedule.

```
/* CLOCK()
 *       Clock driver function
 *                 - uses 2 parameters [Ttrans,Tperiod].
 *                 - and always starts from time 0.
 */

MODEL(CLOCK)
{

    OUT_LIST
          SIG(clk);
    ENDLIST;

    ST-LIST
          SIG(Ttrans);
          SIG(Tperiod);
    ENDLIST;

    /* we must be either at the end of a clock cycle or
     * at the duty cycle transition. Schedule both
     * evaluations at clock cycle end
     */

    if(current_time % Tperiod == 0) /* at end of clock cycle, or at start */
    {
          /* sched for duty cycle transition */
          self_sched(Ttrans, SELFO);
          /* sched for period */
          self-sched(Tperiod, SELFO);
    ,
    clk = (current-time % Tperiod) < Ttrans ? ONE : ZERO;

    EXITMOD(0);
}
```

The external variable **current-time** keeps the current simulation time. The procedure call to **self_sched(Ttrans, SELFO)** schedules the evaluation of the element at **(current_time + Ttrans)** with **priority SELFO.** The priority **SELF0** indicates that the element has the highest **priority** and thus should be evaluated before any other elements.

Similarly, the function **self_unsched** removes the scheduled element already scheduled for future evaluation from the event queue.

7.2. Getting Instance Names from within a Model

When debugging a hardware design or its behavioral model, it is sometimes useful to have, in a model, its instance names defined by its caller or the node names connected to the element.

When an element is evaluated by executing its behavioral model, the procedures **mname()** and **iobname(type, index)** returns the pointer to the element's name (model's instance name) and node name connected to the specified port of the element, respectively.

Consider the following statement in a net list file.

```
(f=xor)(n=xor1)(i=in1, in2)(o=out);
```

The function **mname()** in the 'xor' model returns its instance name 'xorl', and the function **iobname(CINPUT,** 2) returns the **node name** connected to the second port of the element, 'in2'.

7.3. Tquote

The user can simulate his or her design by applying the input stimuli of very regular patterns using the generator elements such as CLOCK or COUNT. However, arbitrary patterns are sometimes necessary to simulate designs. In THOR, the user can create arbitrary patterns using a generator element called **Tquote.** The generator **Tquote** reads in the input stimuli from a file created by the user. Following is an example CSL description.

```
(g=Tquote)(n=input.stim)(o=stim1, stim2)(s=3)(vs=2);
```

where 'input.stim' specifies the name of the stimulus file provided by **the** user. **Tquote** has three internal states: the first one is the period and the other two are for internal use and need not be specified by the user, In the above example, the values of its **outports** 'stiml **'** and 'stim2' are read in from the stimulus file **'input.stim'** every two simulation time steps.

Each statement in the stimulus file begins with either a T or **a *.** The ***** mark designates a comment that terminates by a **;.** The line starting with T character specifies an input vector which is enclosed in a pair of quotes. Each bit of its input vector can have one of the following values: 0, **1, u (U),** or z **(Z).** Note that the number of values inside the quotes must match the number of the output ports of the element specified in the **CSL** statement. A stimulus file for the above example looks like:

```
* This is a comment line;
T'OO'
T'01'T'0z'...
T'UZ'
```

At time 0, the ports 'stiml' and 'stim2' get values 0 and 0, respectively. Then, after 2 time steps, they get 0 and 1, and so on.

7.4. Zero Delay Problems

It is sometimes very convenient to simulate the hardware with all the output delays of each element set to zero (zero delay simulation). Especially for ***combinational logic*** with different numbers of elements from one input port to one output port, the results of zero delay simulation are much easier to examine.

For efficient simulation with zero delay models, it is required that the circuit be ordered topologically. Let's consider the circuit in **Figure 4.** In zero delay simulation, when there occurs an event at node **'n2', all** of **the** elements **'a',** 'b', 'c', and 'd' will be evaluated. Due to these evaluations, new events can be created at nodes **'n3', 'n4',** and **'n5'.** Thus, depending on the order of evaluations, the elements 'b', 'c', and 'd' may be re-evaluated. This, in turn, makes the elements 'c' and 'd' be re-evaluated, and finally the element 'd' should be re-evaluated again.

In **THOR,** there is no evaluation ordering. Therefore, the zero delay elements are repeatedly evaluated until the proper values are calculated (any zero delay elements in a loop may oscillate).

7.5. Handling Biputs

Handling biputs is somewhat tricky and the user should be careful when writing models having biputs. In **THOR,** if a node is driven by an element with an **UNDEF** signal, the signal value at the node becomes **UNDEF** and **all** the **fanout** elements (i.e., all the elements whose input or biput ports are connected to **the node) are** driven with **the UNDEF** signal. Assume that a biput port of an element is connected to a node. If the user writes the element's model in such a way that it sets its biput port to **UNDEF** when it **gets UNDEF** through the biput port, the node will forever be stuck at **UNDEF** once it **gets UNDEF.** As an example, let's consider modeling a pull up resistor. It is likely the user might write the model as follows:

```
MODEL(Rpu)
{
   BI_LIST
        SIG(r);
   ENDLIST;

   switch( r )
   {
        case UNDEF:
           r = UNDEF;
           break;
        case FLOAT:
           r = ONE;
           break;
```

```
     n1   +---+   n3  +---+   n4  +---+  n5  +---+
---->  | a |---->  | b |---->  | c |---->  | d  |---+
       +---+        +---+        +---+       +---+
         ↑            ↑            ↑           ↑
     n2  |            |            |           |
--------+-----------+-----------+-----------+
```

Figure 4. Example circuit showing the zero delay problem

```
        default :
            r = FLOAT;
            break;
    }
    EXITMOD(0);
}
```

Once the node connected to the biput port of the above model gets **UNDEF,** it would be stuck at UNDEF forever. This problem can be avoided as follows:

```
MODEL(Rpu)
{
    BI_LIST
        SIG(r);
    ENDLIST;

    switch( r )
    {
        case FLOAT:
            r=ONE;
            break;
        &fault :
            r = FLOAT;
            break;
    }
    EXITMOD(0);
}
```

7.6. Writing an Edge-Triggered Model

A level-sensitive element operates on the levels of its inputs, i.e. it generztes its outputs based on the levels of its inputs. On the other hand, an edge-triggered element operates on the level transitions of its inputs. Therefore, its model must store previous levels of its inputs to check level transitions. Following is an example model for a D-type positive-edge-triggered flip-flop:

```
MODEL(DFF)
{
    IN_LIST
        SIG(Clk);
        SIG(D);
    ENDLIST;

    OUT-LIST
        SIG(Q);
        SIG(Q_b);
    ENDLIST;

    ST-LIST
        SIG(SavQ);
        SIG(LastClk);
    ENDLIST;

    if ( Clk >= UNDEF )          /* undefined clock input */
```

```
        SavQ = UNDEF;

else if ( vrise[LastClk][Clk] == 1 )        /* rising edge */
        SavQ = vmap[ D ];

Q-b = vinv[ (Q = SavQ) ];

LastClk = Clk;                        /* store last Clk input */

EXITMOD(0);
```

8. Examples

8.1. A 4-Bit Adder, Functional Net List Description

Following is a CSL description of a 4-bit adder. It has two generator elements(COUNT) for the stimuli and two monitor elements, **BINOUT** and **HEXOUT**, to monitor the results. See **gen(3)** and mon(3) for details of generator and monitor elements.

```
/****************************************************************/
/* CSL description of a 4-bit adder with stimuli */

(g=COUNT)(n=gen1)(o=in1[3-0])(s= l)(vs=l);
(g=COUNT)(n=gen2)(o=in2[3-0])(s=1)(vs=16);

(f=add4)(n=adder)(i=in1[0-3], in2[0-3])(o=sum[0-3], cout)(do=0, 0, 0, 0, 0);

(m=BINOUT)(n=mon1)(i=cout);
(m=HEXOUT)(n=mon2)(i=sum[3-0]);
/****************************************************************/
```

The functional model of a 4-bit adder shown below can be used with the above **CSL** description.

```
/****************************************************************/
/* Functional model of a 4-bit adder */

MODEL(add4)
{
   IN_LIST
        GRP(in1, 4);
        GRP(in2, 4);
   ENDLIST;

   OUT-LIST
        GRP(sum, 5);              /* MSB is used for carry out */
   ENDLIST;

   /* if any input is not {ZERO, ONE}, make all outputs UNDEF */
   if(fckbin(in1, 3, 0) != PASSED || fckbin(in2, 3, 0) != PASSED)
   {
        fsetword(sum, 4, 0, UNDEF);
        EXITMOD(0);
   }

   sum[4:0] = in1[3:0] + in2[3:0];

   EXITMOD(0);
}
/****************************************************************/
```

In this model, ‘fckbin’ is a library function which checks that the signals in a group have values in a set **{ZERO, ONE}.** ‘fsetword’ is also a library function which sets signal values to a specified value. See **fckbin(3)** and **fsetword(3)** for details.

8.2. A 4-Bit Adder, Gate Level

This section shows how the 4-bit adder in section 8.1 can be described in gate level. Top level CSL description is almost the same:

```
/****************************************************************/
/* CSL description of a 4-bit adder with stimuli */

(g=COUNT)(n=gen1)(o=in1[3-0])(s= l)(vs=l);
(g=COUNT)(n=gen2)(o=in2[3-0])(s=1)(vs=16);

(f=add4)(n=adder)(i=in1[0-3], in2[0-3])(o=sum[0-3], cout);

(m=BINOUT)(n=mon1)(i=cout);
(m=HEXOUT)(n=mon2)(i=sum[3-0]);
/****************************************************************/
```

Note that there is no delay information for the outputs of the 'adder' because it is not a primitive element but a ***subnetwork*** now. The CSL description of the ***subnetwork*** is as follows:

```
/****************************************************************/
/* CSL description of a 4-bit adder */

(sub=add4)(i=in1[0-3], in2[0-3])(o=sum[0-3], cout)
{
   (f=halfadd)(n=hadd)(i=in1 [O], in2[0])(o=c0, sum[0]);
   (f=fulladd)(n=faddl)(i=inl[ 1], in2[ 1], c0)(o=c1, sum[ 1]);
   (f=fulladd)(n=fadd2)(i=in1[2], in2[2], c1)(o=c2, sum[2]);
   (f=fulladd)(n=fadd3)(i=in1[3], in2[3], c2)(o=cout, sum[3]);
}
/****************************************************************/
```

This ***subnetwork*** consists of four ***subnetworks:*** one half adder and three full adders as &scribed below.

```
/****************************************************************/
/* CSL description of a half adder */

(sub=halfadd)(i=inl, in2)(o=cout, sum)
{
   (f=xor)(n=xor1)(i=in1, in2)(o=sum)(do=0);

   (f=and)(n=and1)(i=in1, in2)(o=cout)(do=0);
}
/****************************************************************/


/****************************************************************/
/* CSL description of a full adder */

(sub=fulladd)(i=inl, in2, cin)(o=cout, sum)
{
   (f=xor)(n=xor1)(i=in1, in2)(o=a)(do=0);
   (f=xor)(n=xor2)(i=a, cin)(o=sum)(do=0);
```

```
    (f=and)(n=andl)(i=inl, in2)(o=b)(do=0);
    (f=and)(n=and2)(i=in1, cin)(o=c)(do=0);
    (f=and)(n=and3)(i=in2, cin)(o=d)(do=0);
    (f=or)(n=orl)(i=b, c)(o=e)(do=0);
    (f=or)(n=or2)(i=e, d)(o=cout)(do=0);
}
/*****************************************************************/
```

where all the functional elements are primitive elements and their models are in the library.

8.3. Functional Model of a 4-Bit ALU

```
/*****************************************************************/
/* Functional model of a 4-bit ALU */

MODEL(alu4_4)
{
    IN-LIST
        GRP(mode, 2);   /* 2 mode lines.
                           00 - add
                           01 - sub
                           10 - and
                           11 -or
                        */
        GRP(in_a, 4);   /* data in A */
        GRP( in-b, 4);  /* data in B */
        SIG(cin);       /* carry in */
    ENDLIST;

    OUT-LIST
        GRP(out, 4);    /* data out */
        SIG(cout);      /* carry out */
    ENDLIST;

    /* numeric representation of A, B, cin and out */
    int a, b, result;

    /* if any input is not {ZERO, ONE}, make all outputs UNDEF */
    if (fckbin(in_a, 0, 3) != PASSED || fckbin(in_b, 0, 3) != PASSED ||
        fckbin(mode, 0, 1)  != PASSED || fsckbin(cin) != PASSED)
    {
        fsetword(out, 0, 3, UNDEF);
        cout = UNDEF;
        EXITMOD(0);
    }

    /* Convert to numeric representations. */
    /* This explicit conversion is not necessary, though */
    /* it sometimes helps code efficiency */
    a = in_a[3:0];
    b = in_b[3:0];
    cin = cin == ZERO ? 0:1;

    cout = ZERO;
    /* perform the function */
    switch(mode[])
    {
    case 0:                          /* add */
         result = a + b + cin;
         break;

    case 1:                          /* subtract */
         result = a - b + cin;
         break;

    case 2:                          /* and */
         result = a & b;
         break;

    case 3:                          /* or */
         result = a | b;
         break;
    }

    /* unpack the result to the output */
    out[3:0] = result;

    /* set carry out */
    if (result > 15) cout = ONE;

    EXITMOD(0);
}
/*****************************************************************/
```

NAME

ana

SYNOPSIS

ana filename

DESCRIPTION

Ana takes as input a file created by the banalyzer monitor and pipes the information into the graphic analyzer. The behavior of the analyzer is the same as when it is being run interactively,

SEE ALSO

banalyzer(3) analyzer(3) aview(1)

NAME

aview – prints banalyzer monitor output in tabular format

SYNOPSIS

aview [-s] [-f format | -ff format-file] [input-file]

DESCRIPTION

Aview **takes the input-file,** a file created by the banalyzer monitor, and prints the results in tabular format. If no input-file is specified, *aview* will read from standard input. The rows represent the time and the columns represent the signals being monitored. The results are printed on the standard output, and the format of the output can be specified by using the **-f** or **-ff** switches (see below).

The **–s** option prints a shortened output file where output is only printed when the signals change.

The -f option allows the user to specify a format for the output. The following argument will be used as the format command.

The -ff option indicates that the following argument is the filename which contains the formatting commands.

Formatting Commands The formatting commands have the following syntax. Note that non-terminals ***are*** shown ***in*** *italic* and terminals **are** shown in **bold.** Symbols enclosed by '[]' imply optional parameters, symbols enclosed by '{}' imply one or more instances:

FormatComm => *FormatEntry* { ***separator*** *FormatEntry* **}**

separator **=>** *:* **| \n**

FormatEntry **=> [*Name* =]** *SignalList* **[@ *Base*]**

SignalList **=>** *signalName* **{ ,** *signalName* **}**

signalName **=> string | *Iterator***

Iterator **=> string [number - number]**

Name **=> string**

Base **=> binary | octal | hex | x**

where ***Name*** will be the name printed for the specified group of signals and their value will be shown in the specified ***Base.*** The default name is the common prefix of a group of signals, and the default base is hex.

The signals will be printed (left to right) in the order in which they are given in the formatting commands. Signals that appear in the input file but are not listed in any formatting command will be printed after (to the right) of the formatted signals, using the following format:
as a group of signals in base hex if any contiguos signals have a common prefix, otherwise they will be printed as single signals in binary base. This is the default when no formatting commands are given.

SEE ALSO

ana(1) analyzer(3) banalyzer(3)

AUTHOR

Someone should claim responsibility

NAME

cio – THOR netlist compiler

SYNOPSIS

cio [–c] [–d file] [-t technology] [–v] csl-file

DESCRIPTION

CIO reads in the **csl-file** specified in the **CSL** language (**CSL** is fully described below) and produces a flattened netlist for the THOR simulator. It is ususally called **by the *gensim*** program rather than directly by the user. The available options are:

–c puts *CIO* into incremental mode. This reflattens only the changed parts of the network specified in the csl-file. This is useful when only one file has been changed.

-d file specify the root-name of the output files

-t technology
define the default technology (not implemented yet)

-*v* Produce verbose output

THE CSL LANGUAGE

A schematic diagram visually specifies how the various parts of a circuit are connected together; likewise a **CSL** file is textually describes how the various hardware *models* of a circuit are connected together. Among the models that can be connected in a **CSL** file are: generic logic gates such as NAND, OR, XOR, etc.; various TTL parts and user supplied models; generators to supply digital waveforms and stimulus to the circuit; monitors to trace selected net-values; and sub-nets (much like 'hardware macros'). These elements are specified in the CSL file as follows:

(f=func)(n=name) options;

This specifies that a new model of type **func** should be added to the schematic with the name **name.** Here **func** can be either one of the supplied models (like AND, **xx74,** etc.), a user model (a model compiled with **the** *MKMOD* program where **the string func** is the name given in **the MODEL(func)** statement which must be compiled from the file **func.c**), or it can be the name of **a sub-netlist.** If it is a sub-list **func** may either specify the file that contains the sub-definition, in which case it must end with **'.csl',** or it specifies the sub-list name defined in the current file. (NOTE: it is usually convenient to put sub-list declarations in separate files) If so, *CIO* will read **in the** file **func.** This is handy for expressing the hierarchy of a circuit.

(m=monitor_type) (n=name) options;

Specifies a monitor where monitor-type is one of HEXOUT, BINOUT, **SPACE,** libanalyzer, or any other monitor described in mon(3).

(g=generator type)(n=name) options;

Specifies a generator where generator-type is one of the supplied generators, Tquote, CLOCK, ONE, ZERO, FLOAT, or any other generator described in gen(3)

(sub=sub-net type) (i=input_list) (o=output_list) (b=biput_list)

{other declarations} Specifies a sub-net (kind of like a hardware subroutine call).

The options that can be specified for functional elements, generators, and monitors are (NOTE: not all of these make sense with the different types)

(i=input_list)

A list of the input nets connected to the element. The net names of **the input-list** must be in the same order as they are declared in the IN-LIST section of the model.

(o=output_list)

A list of the output nets connected to the element in the same order as the OUT-LIST section of the model.

(b=biput_list)

A list of the biput nets connected to the element in the same order as the BI_LIST section of the model.

(vs=initial states)
The initial values of the state variables. You don't need to specify one value for each state (for example if you have a memory with 4096 bytes of state you can specify only one number and the first state value will change).

(s=# of state variables)
This is required ONLY if the vs option is specified. The number of state variables in the ST-LIST section of the model. (GRP(x,8) counts as 8 and SIG(x) counts as 1)

(do=output delays)
the delay of the output signals (one number per wire or bit in a bus) specified in simulator time. (NOTE: default value is 1)

(db=biput delays)
delays for the biputs (NOTE: default value is 1)

The CSL specification may contain C-like comments (i.e. anything contained within '/*' and '*/' is ignored. (actually comments are limited to 2000 characters in length due to limitations of lex so be careful when commenting out large parts of CSL files)

NAMING

Any legal 'C'-name can be used to label **a net.** Busses are specified by brackets (e.g. bus[7-0] for eight bits or bus[2] for a single bit). If you want to leave an output unconnected the net name 'unc' should be used as a place-holder. (e.g. (f=xx74ab)(n=Dflipflop)(i=pre_b, clk, d, clr_b) (o=q_out, unc)(s-2);)

EXAMPLE

(Besides the one given here, examples of CSL files, models and stimulus files are contained in the directory /projects/cad/doc/THOR/examples)

An example CSL file for a micro-processor might look like:

main.csl would contain the highest level components:

```
(f=datapath.csl)(n=dp)(i=data[0-31], control[10-1])(o=flags[3], unc)(b=memory[0-15]);

/* a 256 by 16 bit memory (4096=256x16 i.e. one state variable per byte) */

(f=memory)(n=mem)(i=read, write, addr[8-1])(b=memory[16-1])(s=4096)(db=5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5);

/* HEXOUT displays its inputs in hex format */

(m=HEXOUT)(n=memory_data)(i=memory[16-1]);
```

the file datapath.csl would contain the sub-net declaration for the datapath:

```
(sub=datapath)(i=dp_data[32-1], control[10-1])(b=memory[16-1]);

{

/* a 3-input or gate with a delay of 3 'units' */

    (f=or)(n=decode_or)(i=control[10-9], dp_data[4])(o=decodex)(do=3);

    other declarations go here

}
```

SEE ALSO

THOR(1), MKMOD(1), CSLIM(1), ANALYZER(3), SIMVIEW(1).

AUTHOR

Henry and Beverly Velandi did the initial coding and maintenance. Andy **Maas** added the incremental capabilities. Updates and maintenance by Larry Soule (soule@mojave).

BUGS

Path names are relative to the starting directory rather than the current directory **-** this limits the hierarchy to two levels (top and sub-directories).

Error messages are a little vague.

The technology specification file is not implemented.

Sub-net calls should be parameterizable but they aren't.

Please report any other bugs/suggestions to **soule@mojave.stanford.edu**

NAME

CSL – THOR **netlist** language

DESCRIPTION

THE CSL LANGUAGE

A schematic diagram visually specifies how the various parts of a circuit are connected together; likewise a CSL file is textually describes how the various hardware ***models*** of a circuit are connected together. Among the models that can be connected in a CSL file are: generic logic gates such as NAND, OR, XOR, etc.; various TTL parts and user supplied models; generators to supply digital waveforms and stimulus to the circuit; monitors to trace selected net-values; and sub-nets (much like 'hardware macros'). These elements are specified in the CSL file as follows:

(f=func) (n=name) options;

This specifies that a new model of type **func** should be added to the schematic with the name **name.** Here **func** can be either one of the supplied models (like AND, xx74, etc.), a user model (a model compiled with the ***MKMOD*** program where the string **func** is the name given in **the** MODEL(**func**) statement which must be compiled from the file **func**.c), or it can be the name of a sub-netlist. If it is a sub-list **func** may either specify the file that contains the sub-definition, in which case it must end with **'.csl',** or it specifies the sub-list name defined in the current file. (NOTE: it is usually convenient to put sub-list declarations in separate files) If so, *CIO* will read **in the** file **func.** This is handy for expressing the hierarchy of a circuit.

(m=monitor_type) (n=name) options;

Specifies a monitor where monitor-type is one of HEXOUT, BINOUT, SPACE, iibanalyzer, or any other monitor described in mon(3).

(g=generator type) (n=name) options;

Specifies a generator where generator-type is one of the supplied generators, Tquote, CLOCK, ONE, ZERO, FLOAT, or any other generator described in ***gen(3)***

(sub=sub-net type) (i=input_list) (o=output_list) (b=biput_list)

{other declarations} Specifies a sub-net.

The options that can be specified for functional elements, generators, and monitors are (NOTE: not all of these make sense with the different types)

(i=input_list)

A list of the input nets connected to the element. The net names of the **input-list** must be in the same order as they are declared in the IN-LIST section of the model.

(o=output_list)

A list of the output nets connected to the element in the same order as the OUT-LIST section of the model.

(b=biput_list)

A list of the biput nets connected to the element in the same order as the BI_LIST section of the model.

(s=# of state variables)

The number of state variables in the ST-LIST section of the model. (GRP(x,8) counts as 8 and SIG(x) counts as 1)

(vs=initial states)

the initial values of the state variables

(do=output delays)

the delay of the output signals (one number per wire or bit in a bus) specified in simulator time. (NOTE: default value is 1)

(db=biput delays)

The CSL specification may contain C-like comments (i.e. anything contained within '/*' and '*/' is ignored. (actually comments are limited to 2000 characters in length due to limitations of lex so be careful when commenting out large parts of CSL files)

NAMING

Any legal 'C'-name can be used to label a net. Busses are specified by brackets (e.g. bus[7-0] for eight bits or bus[2] for a single bit). If you want to leave an output unconnected the net name 'unc' should be used as a place-holder. (e.g. (f=xx74ab)(n=Dflipflop)(i=pre_b, clk, d, clr_b) (o=q_out, unc)(s=2);)

EXAMPLE

(Besides the one given here, examples of CSL files, models and stimulus files are contained in the directory /projects/cad/doc/THOR/examples)

An example CSL file for a micro-processor might look like:

main.csl would contain the highest level components:

```
(f=datapath.csl)(n=dp)(i=data[0-31], control[10-1])(o=flags[3], unc)(b=memory[0-15]);
/* a 256 by 16 bit memory (4096=256x16 i.e. one state variable per byte) */
(f=memory)(n=mem)(i=read, write, addr[8-1])(b=memory[16-1])(s=4096)(db=5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5);
/* HEXOUT displays its inputs in hex format */
(m=HEXOUT)(n=memory_data)(i=memory[16-1]);
```

the file datapath.csl would contain the sub-net declaration for the datapath:

```
(sub=datapath)(i=dp_data[32-1], control[10-1])(b=memory[16-1]);
{
/* a 3-input or gate with a delay of 3 'units' */
    (f=or)(n=decode_or)(i=control[10-9], dp_data[4])(o=decodex)(do=3);
    other declarations go here
```

SEE ALSO

THOR(l), MKMOD(l), CSLIM(1), ANALYZER(3), SIMVIEW(1).

AUTHOR

Henry and Beverly Velandi did the initial coding and maintenance. Andy Maas added the incremental capabilities. Updates and maintenance by Larry Soule (soule@mojave).

BUGS

Path names are relative to the starting directory rather than the current directory - this limits the hierarchy to two levels (top and sub-directories).

Error messages are a little vague.

The technology specification file is not implemented.

Please report any other bugs/suggestions to soule@mojave.stanford.edu

NAME

cslim – PLA Generator for THOR

SYNOPSIS

cslim [-xsqd[n]] [file]

DESCRIPTION

CSLIM takes a finite state machine model written in a subset of THOR(I), and calculates the PLA equations for ***espresso(l)*** optimization and eventually a PLA generator.

USAGE

If a file name is given on the command line, input is read from that file; otherwise, input is read from ***stin.*** The output is written ***to stdout,*** and is usually piped through espresso. For final PLA's, use the -do qm option to increase the optimization level, realizing that this will slow it down. Thus,

cslim ***infile.c*** **| espresso -do qm >** ***foo.tt***

The **-q** option turns off the checking for unset outputs, and initializes all outputs to undefined at the beginning of the model. By default, the program will insure that each output is set in every possible flow through the model, and complain about those that might not be set. This flag will turn off those complaints.

The **-x** option turns off the handling of undefined cases in switch statements. If you perform a switch on a group, and do not specify the full range of possible values, and don't provide a default condition, the program will assume that the other cases are not possible, and add the cases not mentioned to the don't care set. If you turn on this option, empty default statements will not be handled like this.

The **-s** option allows you to automatically store state in your PLA for those state bits that are not assigned If you set this option, the generated PLA will be larger, but it will stay in the same state unless you specifically assign it to be something else. Normally, if you do not assign the state in every possible flow through the program, it is an error that will be caught by the unset outputs checking.

The **-d** option in cslim turns on the debug mode; there are three levels of debug, turned on by **-d, -d2,** and **-d3.** Each level prints out increasing amounts of information.

INPUT FORMAT

Input is a restricted THOR model of a finite state machine, Sections of code you want ignored by **CSLIM** can be enclosed in **{ { }}** or **/*{ { */ /*}}*/;** this is essential for all parts of the THOR model which do not pertain directly to the logical function of the PLA, such as clock triggering. The type checking in CSLIM is more rigorous than that in THOR; you cannot assign an arbitrary integer value to **a SIG, even if it is a state SIG; you must use funpack()** and **fpack()** (or their **abbreviations grp[]** and **grp[n:m];** see **mkmod(1))** on a declared **GRP. Only** the switch and if control constructs are allowed. Conditionals are limited to -- and **!=,** with **&&, ||, !,** and **()** operators. Each comparison must compare a SIG to a signal type (ONE, ZERO, FLOAT, or UNDEF) or the value of **funpack()** of a **GRP** to an integer. Each switch element must have a single label, and must be terminated by a break. Nesting of control structures is arbitrary, allowing, for instance, easy RESET of a PLA. All input, output, or state variables not a direct part of the PLA must be 'commented out' with the double curly braces above. Output signals can be assigned multiple times; for instance, all outputs might be initialized at the head of the model, and then conditionally changed.

OUTPUT FORMAT

The output is a fully specified function consisting of an ON-set and an OFF-set, in the 'fr' format of espresso. The input, output, and state signal names are also written for inclusion in the PLA. See *espresso(5)* for a description of the format.

DIAGNOSTICS

In case of a syntax error, the offending line and a character pointer is printed out. Currently no description of the error is printed; a parse failure is simply returned. If a GRP is compared to a SIGNAL or a SIG is compared to an integer, a warning message is printed.

SEE ALSO

espresso(l), mkmod(1), espresso(5), THOR(l).

AUTHOR

Tomas Rokicki (rokicki@sushi.stanford.edu).

BUGS

Output diagnostics are terrible.

You can't specify the order of the bits of a group in the output equations.

The use of **/*{{*//*}}*/** and **{{}}** is rather ugly.

If a state variable is not assigned during execution of the model in THOR, and then checked, the THOR model execution will not reflect the PLA generated. Do not do this.

NAME

gensim – generates the simulator for the given network

SYNOPSIS

gensim root.csl...
[user-model.0]...
[-t #time-steps]...
[-x 1 **[-i** 1 **[-f** 1

root.exe [-i] [-f] [-t #time-steps] [> output-file]

DESCRIPTION

Gensim generates the executable simulator. It runs the CSL compiler ***cio*** on the specified network files, links the simulator with the user defined models, and generates the simulator for the network. Note that the network file specified must be the root of the network to simulate. All other arguments are optional.

The options with their default values are as follows:

user-model.0
No default. Link with one or more user defined models. The models must first be compiled with **mkmod.** See n&mod(l). Note that gensim will automatically include models specific to the given network. However, this option allows user libraries to also be included.

- x When this flag is present, the simulator executable will be built, but not run.

- i When this flag is present, the simulator will run in interactive mode, to speed the debugging process.

- f When this flag is present, the simulator will run in compiled mode instead of event-driven mode. This is a faster way to simulate circuits at the higher abstraction levels.

-t #time-steps
This option allows the number of time steps simulated to be set. The default is 1000.

Cio is run automatically by **gensim. Cio** is the compiler that reads the network description files and produces a detailed description of the network for the simulator. Although **cio** can be run by itself, the reasons to do so are being eliminated.

The models that were specified in the network are linked with the actual simulator code to produce the executable simulator. The simulator reads the files produced by **cio** to get the network connectivity, output delays, etc. The simulator can also be executed by **root** concatenated with .exe with possible options of simulation time, specified with the **-t** option. The simulator sends its output (which is the output from the monitor models) to stdout. Error messages go to stderr.

FILES

root.exe	simulator executable
root.out	output from simulation run
root.elm	element information, from cio
root.con	network connectivity, from cio
root.sys	list of models used, from cio
root.c	model function address table, from cio

SEE ALSO

THOR(1) mkmod(1) cio(1) simview(1)

AUTHORS

Modified bv Sun Young Hwang and Robert Alverson

Written by Henry and Beverly Velandi
University of Colorado, Boulder

NAME

intTHOR - user interface to THOR simulator

DESCRIPTION

THOR is a behavioral simulator using models written in the CHDL modeling language and network descriptions written in the CSL language. With THOR, simulation can be done in interactive mode, batch mode, or both combined. There are a variety of commands available to control the execution of the simulator and they are described here.

After entering the interactive mode of simulation, the system writes the prompt on the terminal. The prompt includes the current directory path and current simulation time maintained by the simulator. Network hierarchy is considered similar to the file system hierarchy. A file can be considered as an element model, or a node, while a directory can be considered as a **subnet.** To confine the range of the scope to a particular **subnet,** the **change** command can be used. The user can respond to the prompt by entering a command followed by option fields. Unambiguous abbreviations are accepted. For example, **'st'** and 'ste' are acceptable as the **step** command. The wild-card characters * and ? are allowed in arguments.

To enter the interactive mode, one can apply the following option when running the simulator:

-i [filename] [-q] [-1] [-c]

If **filename** is given, the file is read as a command file and executed. If **-q** is given, quiet mode is set and there is nothing displayed while the command file is executed. Otherwise, all the command lines and their results are displayed as if the commands were given from keyboard. If **-1** is given, a log file with an extension **.log** is generated. If **-c** is given, a command file with an extension .cmd is generated.

example:

```
        multiplier.exe -i test.cmd -1 -c

                ... Initially, the command file 'test.cmd' is read and
                    executed. Then, the following commands entered
                    interactively are written out to 'multiplier.cmd'.
                    All the outputs displayed on the screen are logged into
                    'multiplier.log'.
```

Control commands for simulation are described as follows:

alias [word line]

If there are arguments **word** and **line, the line** gets **word** as an alias. **The line** is a sting enclosed in a pair of double quotes. If no argument is given, it lists all the aliases.

example:

```
        > alias x "examine in1 in2 out"
        ... The word x is replaced internally with
            the line, examine in1 in2 out.
```

Use the command unalias to remove an alias.

break [node(s)] [(node value)(s)] [element.(s)]

This command sets break points if there are one or more arguments given. If not, it displays the list of the break points currently set. If **node(s)** are specified in the option field, break points are set for changes of node values. If **(node value)(s) are** specified, break points are set for the specified node value pairs. If **element.(s) are** specified, break points are set for the element's model entrance.. (NOTE: element specifications always end with a period) Any combination of the above can be given as arguments to a command line.

> break or1.in1 ... break on change of value of node or1 .in1
> break (orl.inl 1) . . . break when node orl.inl gets value 1
> break orl. ... break on activation of element or1

Break points are reset with **the clear** command.

change directorypath

This command changes the working directory in hierarchical network to the directory specified by the argument. Root directory and parent directory are designated by **'$'** and '^', respectively.

example:

> change ^.adder
... Changes the working directory to the sub-network 'adder'
in the parent directory.
> change $.adder.xor
... Changes the working directory to the sub-network 'xor',
two levels down from the root directory. Root directory
means a topmost circuit description in a hierarchical network.

clear [number(s) or *]

This command clears break points currently set, With the argument ***,** it clears all the break points. Otherwise, it clears break point(s) **specified by the number(s).** Each break point is given with **a** unique number so that it can be cleared easily by specifying the number. The number can be obtained with **the break** command without arguments.

example:

> clear 2

connectivity [node or element.]

This command is used to examine the circuit connectivity. If **node** is given as an argument, it displays all the elements connected to the node with their pin numbers and pin names. If **element** is given, it displays all the nodes connected to the pins of the element. (NOTE: element specifications end with a period)

example:

> connectivity adder.xor2.
> connectivity adder.xor2.in 1

deposit (node value)(s)

This command temporarily sets the values of the specified nodes to the specified values. As the simulation proceeds, the values may change.

example:

> deposit (adder.xor2.in1 0)

disable [number(s) or *]

This command disables the dump of node values. If * is given, it disables all the nodes. Otherwise, it disables the dump only for **the** nodes specified by the **number(s).** The numbers can be obtained with the **enable** command without arguments.

example:

> disable 2

dump This command dumps the values of enabled nodes.

This command is used to examine nodes of elements every time the simulation stops or the **dump** command is issued If a node is specified as an argument, it enables the dump of the node values. If an element is specified as an argument, it enables the dump of the state and pin values of the element. Without any argument, it displays the list of the enabled nodes and elements.

example:

```
        > enable adder.xor1 .in 1
```

Enabled nodes or elements are disabled with the **disable** command.

examine [node(s)] [element.(s)]

If a node is specified, it displays the current value of the specified node. If an element is specified, it displays the state and pin values of the specified element.

example:

```
        > examine muxl .and2.*
                produces the output like:
                mux1.and2.in1 = 0
                mux1.and2.in2 = 1
                mux1.and2.out = 0
```

exit This command is for exiting the simulator. It is equivalent to **the quit** command.

freeze [node(s)] [(node value)(s)]

This command freezes nodes if there are one or more arguments given. Otherwise, it displays the list of the nodes currently frozen. If **node(s)** are specified, the nodes are frozen to current values. If **(node value)(s)** are specified, the nodes are frozen to the specified values.

example:

```
        > freeze adder.xor 1 .in 1
        > freeze (adder.xorl.inl 1)
```

go [[+]time]

This command starts simulation and continues until ***max_time*** is reached, if there are no arguments. If + **is** omitted, the simulation continues till absolute **time** is reached. + means relative time unit and the simulation stops **time** steps after current simulation time.

example:

```
        > go +10 . . . simulates for 10 time units.
        > go loo . . . simulates to absolute time 100.
```

help This command displays help information about each command with examples.

mark This command marks the current time so that later the simulation can be restarted from this time.

quit This command is for exiting the simulator. It is equivalent to the **exit** command.

restart This command restarts simulation from the time marked by the **mark** command. If a time has not been marked, it restarts from time 0.

run filename

This command reads command lines from the command file specified by **filename** and executes them. The command file can be created by the user with an editor or by the simulator for simulation rerun. The latter can be done by setting the variable cmd_file to 1 (see set command).

set (variable [value])(s)

This command **sets variables** to **values.** Allowed Variables are **max_time, timeunit, log,**

cmd-file, and **quiet. max-time** is the maximum time units that are simulated. The default value **is** 1000. **timeunit** is time units per step. The default value is 1. If **log is** set to 1, a log **file(filename.log)** is created. The default value is 0. If **cmd-file** is set to 1, all the commands executed during simulation are written to **a** file(filename.cmd) such that one can rerun the same commands. The default value is 0. If **quiet** is set to 1, the commands read from command file are executed in quiet mode. Otherwise, all the command lines and their results are displayed on the CRT screen while the simulator runs. The default value is 0.

example:

```
> set (max_time 500)
```

... allows simulator to run up to 500 absolute time units.

show variable(s)

This command displays the values of variables. Allowed variables are **max-time, timeunit, log, cmd-file,** and **quiet.**

example:

```
> show max_time
```

step This command simulates for one time unit

trace [node(s)] [(node value)(s)] [element.(s)]

This command sets traces if there are one or more arguments given. If not, it displays the list of the nodes and elements set for traces. If **node(s) are** specified, the nodes are set for traces of changes of node values. If **(node value)(s)** are specified, the nodes are set for traces of specified node values. If **element.(s)** are specified, traces are set for model entrances of the specified elements. That is, it traces the evaluation of the elements.

example:

```
> trace adder.xor1.in1
> trace (adder.xorl .inl 1)
> trace adder.xorl.
```

unalias [number(s) or *]

This command removes aliases. If * is given, it removes all **the** aliases. If **number(s) are** given, it removes the specified **alias(es).** The numbers can be obtained with the **alias** command without arguments.

unfreeze [number(s) or *]

This command unfreezes frozen nodes. With ***,** it unfreezes all the frozen nodes. Otherwise, it unfreezes nodes specified by the **number(s).** The numbers can be obtained with the **freeze** command without arguments.

example:

```
> unfreeze 2
```

untrace [number(s) or *]

This command resets traces currently set. If * is given, it resets all the traces. Otherwise, it resets the trace settings specified by **the number(s).** The numbers can be obtained with the **trace** command without arguments.

example:

```
> untrace 2
```

!string The string is ignored. It is accepted as a comment line.

SEE ALSO

THOR(1) gensim(1)

AUTHORS

Kiyoung Choi
Stanford University

BUGS

Please report any bugs to choi@sonoma.stanford.edu

NAME

mkmod, epp - description of r&mod and epp, and introduction to THOR models

SYNOPSIS

mkmod [-p] [-m] file [cc options]
epp [-d [n]] file

DESCRIPTION

Mkmod is the THOR model compiler. It runs epp over the model to convert the THOR constructs into C constructs, and then invokes the C compiler.

Epp is the THOR model preprocessor, which is used to convert signal and group declarations into the appropriate C code, and convert implicit uses of groups into the necessary function calls. The user should never need to call this directly; mkmod **-p** will generate just the preprocessed output.

USAGE

For **mkmod,** the file name must not have a suffix; **.c** is added to find the source. Any options given after the flle name are passed on to the C compiler.

-m Compile as a main program. This will compile and attempt to **link** your file; without this option, the model is only compiled into an object file.

-p Only run through epp; no compilation is attempted.

For epp, the file name must be given in full; output is written to standard output. Signal and group declarations are changed to C code; implicit group usage is changed to function calls (for instance, a[] = b[] is changed to funpack(a, 0, n, fpack(b, 0, n)).

-d Turn on debug mode; this is optionally followed by a single digit which can be used to set the debug level higher.

Both mkmod and epp use the THOR environment variables; see THOR(l) for information about these.

INPUT FORMAT

The input to **mkmod** is a THOR model, essentially written in C. This model consists of the following sections:

Model header

Input, output, biput, and state signals and groups

Initialization section (optional)

Body

The model header consists of the keyword MODEL, with the name of the model in parenthesis, followed by an open brace. This name will be given in the (f=...) section of the CSL.

Following this are the input, output, biput, and state signal and group declarations. These start with IN-LIST, OUT-LIST, BI_LIST, and ST-LIST respectively, and end with ENDLIST. Inside each section are declarations for signals and groups. The order within each group must match the order of the signals and groups within the CSL; the names can be different. The declarations possible are:

SIG(name) Declares a single wire called name.

GRP(name, size [, {BIGENDIAN|LITTLEENDIAN}]) Declares a group (or bus) of size wires. If a third argument is given, it must be one of BIGENDIAN or LITTLEENDIAN, and it specifies the default bit ordering of the group when it is used as an integer; BIGENDIAN is the format used by

Motorola, for instance, with the most significant bit being the one with the highest number, and LITTLEENDIAN is used in IBM mainframes, with the most significant bit being bit 0. If the third argument is omitted, BIGENDIAN is assumed.

VGRP(name) Declares a group of unknown size. There can only be one of these for each signal section (input, output, biput, and state) and it must be the last declaration. This is useful for creating parameterized models.

TSIG(name) Declares a temporary (internal) signal.

TGRP(name, size [, {BIGENDIAN|LITTLEENDIAN}]) Declares a temporary (internal) group.

BUS and REG are synonyms for GRP; likewise, VBUS and VREG for VGRP.

The model body can include any standard C statements or functions. Signals are integer variables which can take on four values: ONE, ZERO, FLOAT, and UNDEF. Groups are arrays of signals; these can be accessed as individual signals by g[3] to get the third signal of group g, for instance. In addition, entire groups can be packed into an integer value or integer values put onto a group through the fpack and funpack procedures described below. They may also be referred to by following them by expressions in brackets, as:

g[] means the entire group interpreted as an integer; it takes on values from 0 to 2^n-1, where there are n bits. A negative number indicates that one of the signals in the group is floating or undefined.

g[n] means the nth signal of group g; it can take on values ONE, ZERO, FLOAT, or UNDEF.

g[n:m] means signals n through m of group g interpreted as an integer; takes on values from 0 to 2^(m-n+1)-1. Again, a negative number indicates that one of the signals in the group is floating or undefined. The first number (n) is used as the most significant bit; the second is used as the least significant.

Any of these three formats can be used as a value or as the left hand side of an assignment statement.

The initialization section (if one exists) must begin with INIT and end with ENDINIT. Any statements can be executed within the initialization section, such as initializing some state variables or setting up an internal array.

The body of the model is executed whenever the values of one of the input or biput signals or groups changes, It must be exited with the EXITMOD(value) statement; a non-zero value indicates an error. Any number of EXITMOD(value) statements can be used.

EXAMPLE

Here is a small example of a model and an explanation of its function.

```
/*
 *  This is an example model for a
 *  74138-like 3 to 8 demultiplexor.
 *  Three enable inputs, two active
 *  low, and the three inputs and
 * eight outputs. The name this
 *  model should be referred to in
 * the csl is 'mux'.
 */
```

```
MODEL(mux) {
/*
 *   Next come the input, output,
 *   biput, and state signal declarations.
 *   Note that ',BIGENDIAN' below is
 *   unnecessary as it is the default.
 */
IN-LIST
        SIG(en_a);
        SIG(en_b_bar);
        SIG(en_c_bar);
        GRP(mux_in,3,BIGENDIAN);
ENDLIST;

OUT-LIST
        GRP(mux_out_bar,8,BIGENDIAN);
ENDLIST;
/*
 *   The following two sections are
 *   entirely optional when there
 *  are no signals.
 */
BI_LIST
        /* none */
ENDLIST;

ST-LIST
        /* none */
ENDLIST;
/*
 *   Next comes the main code section,
 *  which consists of C program
 *  statements, with the group
 *   shorthand described above.
 *
 *   We initialize the output to
 *   nothing enabled; then, if the
 *   enable signals match, we hit
 *  that output.
 */
          mux_out_bar[] = 255 ;
          if (en-a == ONE && en_b bar == ZERO
                        && en_c bar == ZERO)
                mux_out_bar[mux_in[]] = ZERO ;
          EXITMOD(0) ;
 }
```

BUGS

Most errors, even in the THOR specific sections such as signal declarations, will be reported in an obtuse way by the C compiler.

SEE ALSO

cio(**1)**
cslim(1)
THOR(1)

AUTHORS

Written by Henry and Beverly Velandi
University of Colorado, Boulder

Minor modifications to mkmod and
rewrite of epp by Tomas Rokicki
Stanford University

NAME

simview – prints simulator output in tabular format

SYNOPSIS

simview [-s] [input-file-name]

DESCRIPTION

Simview **takes the input-file-name,** a file in THOR format, and prints the results in tabular format. **If** no input-file-name is specified, ***simview*** will read from standard input. **The rows** represent the time and the columns represent the signals being monitored. The results are printed on the standard output. The **–s** option prints a shortened output file where output is only printed when the signal changes.

The simulator output file format is:

1) The first line in the file is the maximum time reached during simulation

2) The format of subsequent lines is:

   time monitor_name value

   where, time is the time the monitor was called, monitor-name is the name of the monitor, and value is the value of the monitor input at this time. Value is an arbitrary string of any length, but this length must remain constant for a given monitor or the columns will not line up in the output from ***simview.***

3) Note, all monitors are called by the simulator at time zero.

FILES

SEE ALSO

BUGS

AUTHOR

Peter Moceyunas
University of Colorado, Boulder

NAME

THOR - introduction to THOR commands and applications programs

DESCRIPTION

THOR is a behavioral simulator using models written in the CHDL modeling language (a hardware description language based on the "C" programming language) and network descriptions written in the CSL language. There are a variety of commands available to control the execution of THOR ***(gensim)*** and to develop models that are simulated using THOR. The simulation can be done in interactive mode, batch mode or both combined. The simulator output can be printed in tabular format using ***simview*** or processed and displayed ***in*** waveform using ***analyzer.*** A detailed description and a user tutorial may be found in ***the THOR tutorial*** *(THORtutor(1)).*

The basic entity used in formulating THOR networks is the ***model*** which represents a hardware device. These ***models*** are linked together to form a network using the CSL network definition language. ***Cio*** reads this network description, compiles it, and produce a detailed description for the simulator. Models are written using the CHDL modeling language and compiled by *mkmod* to produce object codes for the simulator. There are useful ***functions*** which may be used in formulating models. ***Functions*** are logical or behavioral units that are called by a model just as a function is called in "C". ***These functions*** return a value to the ***model*** on return after they are finished executing. As an aid for the model synthesis, ***cslim*** generates an optimized PLA from a finite state machine model written in **CHDL.**

SEE ALSO

General

ana(1)	- pipes banalyzer monitor output into graphic analyzer
aview(1)	- prints banalyzer monitor output in tabular format
cio(1)	- compiles network description files written in CSL
cslim(1)	- generates PLA from a finite state machine model
intTHOR(1)	- user interface to THOR simulator
mkmod(1)	- compiles a model written in CHDL
gensim(1)	- generates executable simulator, and controls execution of THOR
simview(1)	- prints simulator output in tabular format
THORtutor(1)	- THOR tutorial
simint(5)	- interface to Rsim, chip tester, etc.

General purpose models

analyzer(3)	- a monitor which displays its inputs graphically
banalyzer(3)	- batch analyzer monitor
gen(3)	- describes the signal generator models available
generic(3)	- describes generic gates, (AND,NOR,etc)
mon(3)	- describes the signal monitor models available
ttl(3)	- describes the ttl models available

General purpose functions used in models

current-time(3)	- a copy of the simulator's absolute time variable
FDUMP(3)	- macros used when debugging is desired in a function
fadd(3)	- adds two n-bit groups with carry-in/out
faddc(3)	- adds n-bit group to a constant

fand(3)	- logically ***ands*** two n-bit groups
fandc(3)	- logically ***and*** a signal group and a constant
fcat(3)	- concatenate two signal groups
fcatac(3)	- concatenate a siganl group and a constant
fcatca(3)	- concatenate a constant and a signal group
fckbin(3)	- checks for valid values {ZERO,ONE} in a group
fckmsize(3)	- check maximum size of lsb - msb range
fckpty(3)	- compute parity type of signal group
fckrange(3)	- compare two ranges of msb - lsb
fckrange3(3)	- compare three ranges of msb - lsb
fckvalue(3)	- checks for invalid signal values
fcopy(3)	- copy a signal group
fcopyinv(3)	- copy and invert a signal group
fdecr(3)	- decrement the value of a signal group
ferr(3)	- prints the type of error requested
fgetval(3)	- initialize a signal group
fincr(3)	- increment the value of a signal group
finitmem(3)	- initialize memory from a file
finv(3)	- logical inverts a group
fior(3)	- inclusively ***ors*** two n-bit groups
fiorc(3)	- inclusively ***ors*** a constant and signal group
fpack(3)	- convert signal group to an integer
fprval(3)	- prints value of a signal group
fprvec(3)	- prints the value of a signal group with a label
frorl(3)	- circular rotates a group left (LSB to MSB)
frorr(3)	- circular rotates a group right (MSB to LSB)
fsckbin(3)	- check for valid values {ZERO,ONE} in a signal
fsetword(3)	- set n-bit group to a logical value
fshftl(3)	- shift a group left (LSB to MSB)
fshftr(3)	- shift a group right (MSB to LSB)
fshftr0(3)	- shift a group right with 0 shifted in
fsub(3)	- subtracts two n-bit groups with borrow-in/out
fsubc(3)	- subtract a constant from a signal group
fswap(3)	- exchange bits in signal groups
funpack(3)	- convert a constant to a signal group
fxnor(3)	- logically exclusively ***nors*** two n-bit groups
fxnorc(3)	- exclusive nor of a siganl group and a constant
fxor(3)	- logically exclusively ***ors*** two n-bit groups
fxorc(3)	- inclusive or of signal group and a constant
mname(3)	- returns a pointer to the user defined name
self-sched(3)	- schedules a model for evaluation

Vector functions used in models

vectors(3)	- general description of vector functions
vand(3)	- logical ***and***
vfall(3)	- check for 1 -> 0 transition
vinv(3)	- logical inversion
vmap(3)	- maps legal values
vnand(3)	- logical ***nand***
vnor(3)	- logical ***nor***

vor(3)	- logical ***or***
vrise(3)	- check for 0 -> 1 transition
vtbuf(3)	- single bit **tri-state** buffer
vtbufi(3)	- single bit inverting **tri-state** buffer
vtgate(3)	- CMOS transmission gate
vxnor(3)	- logical exclusive ***nor***
vxor(3)	- logical ***exculsive or***

AUTHORS

Modified by members of CAD group
Stanford University

Written by Henry and Beverly **Velandi**
University of Colorado, Boulder

NAME

THORtutor - THOR tutorial