```
/* ------------------------------------------------------------------------
/*  Simulate_Step() -- Perform one simulation step time advancement.
    Dequeue all events to be run at the current time and update the
    net values.  Then Enq_Sinks() of all nets which changed to generate
    the next round of events.  */
/* ------------------------------------------------------------------------
void Simulator::Simulate_Step() {
  //  make these static so they don't get re-allocated every call!;
  static Event_Class *e;
  static Net_Class *net;
  static Basis *tempobj;
  static Queue Changed_Nets;

  int e_count=0;  //  number of events that were executed at this time step

  /* dequeue events, update net values,  and enqueue changed nets */
  while (EventQ.Not_Empty() && (EventQ.Get_Next_Key() <= Get_Sim_Time())) {
    e = (Event_Class *) EventQ.Deq();
    net = (Net_Class *) e->Get_Net();
    net->Set_Pending_Event(net->Get_Pending_Event()-1);
    net->Set_Value(e->Get_New_Value());
    Changed_Nets.Enq_Head((Any_Type) net);

    delete e; /* free up memory space */
    e_count ++;
  }

  //  inc. the total overall event count
  event_count += e_count;

  fprintf(history_st,"%d %d\n",Get_Sim_Time(),e_count);

  //  For all nets which have changed value, put their sinks onto;
  //  the PostQ for consideration for generating new events.  Objects
  //  are only added to the PostQ IF they are not already there, thus
  //  eliminating unnecessary event generation.

  while (Changed_Nets.Not_Empty()) {
    net = (Net_Class *) Changed_Nets.Deq_Tail();

    /* enqueue all objects connected to the changed wire */
    net->Enq_Sinks();

  }

  //  All objects on the PostQ can now process their inputs and enqueue
  //  new events onto the event queue.

  while (PostQ.Not_Empty()) {
    tempobj = (Basis *) PostQ.Deq_Head();
    tempobj->Process_Input();
  }
}
```

```
/* ----------------------------------------------------------------------------
/*  set_value_help() -- Set the value of a net.  This routine is tri-state
    smart.  It relies on the bit encoding of the net values, and also
    assumes that all objects connected have a Get_Outvalue() function by
    doing a wired-and function.   */
/* ----------------------------------------------------------------------------
static Any_Type set_value_help(Any_Type d) {
  Basis *obj;
  obj = (Basis *) d;
  int val;

  val = obj->Get_Outvalue(local_ID);
  if ((val==XXXXXXX) || (retval==XXXXXXX)) retval = XXXXXXX;

  //  Relies on the fact that z=3, x=2, so z&z = z, z&1=1, z&0=0
  else retval = retval & val;

}

/* ----------------------------------------------------------------------------
/* Set_Value() -- Set the value of a net, and rise/fall times.
   This is tri-state aware.  If the net is tristate, it will look at all
   the sources and do a wired-and function on them.  */
/* ----------------------------------------------------------------------------
void Net_Class::Set_Value(int val) {

  if (type==SIMPLE_NET) {
    if ((un.simple.value.Get_Value() == 0) && (val==1))
      rise_time = Get_Sim_Time();
    if ((un.simple.value.Get_Value() == 1) && (val==0))
      fall_time = Get_Sim_Time();

    if (Get_Tristate()) {
      /* examine all the sources and set value to wired-and */
      retval = ZZZZZZZ;
      local_ID = (int) Get_ID_Addr();

      Sources.Iterate(set_value_help);

      un.simple.value.Set_Value(retval);
    }
    else un.simple.value.Set_Value(val);
  }
  else fprintf(error_st,"Net %s was not SIMPLE.  Couldn't set value\n",
        Get_Name());
}
```

Figure 30: Assigning Tri-state Values to Nets

5.4 Tri-state Net Updating

Tri-state nets are more complex to update than normal nets. When there is a change on a tri-state net, it is necessary to examine the outputs of all source objects connected to that net, and, by clever selection of the value representation of ZZZZZZZ and XXXXXXX, performing a wired-AND operation on those sources. *Figure 30* is the source code for updating the value of a net.

5.5 Simulation Loop

The main event-processing simulation loop is implemented in `Simulate_Step()` (see *Figure 5.5*). There are three separate phases in one simulation step. First, all events at the current simulation time are dequeued and the nets associated with those events are put on a queue of changed nets. The new values of the nets are also set in this loop. In the second phase, the sink objects of all of the changed nets are enqueued onto a `Post_Queue`. The reason for putting items onto this queue is so that only unique net changes will be enqueued. Finally, in the last phase, all of the items on the `Post_Queue` perform a `Process_Input()`, which will place the next group of events onto the `EventQ`.

5.6 Comments

Simulation time is very fast for designs with .eqn objects, but gate-level objects need to be faster. This is a by-product of the large overhead induced by managing the event queue.

The current user interface to the simulator is a non-interactive command script. It would be nice if an interactive X-windows interface could be built around the simulator.

The simulator only supports unit-delay simulations, however, in the future, variable-delay simulations might be supported. This would, however, cause the simulator to grow slower, unless improved event-processing algorithms and profiling are used.

```
/* Constructor */
INV::INV(char *n, Net_Class *out,  Net_Class *i1):Basis(_INV,1,n) {
      input1 = i1;
      output = out;
};

/* Process_Input() -- update internal value */
void INV::Process_Input() {
      value = !(input1->Get_Value());
      EventQ.Post_Event(output, value, Get_Delay());
};

/* Print_Info() -- Output information about the inverter */
void INV::Print_Info(FILE *out) {
   fprintf(out,"INV: %s in: %d out: %d\n",Get_Name(),input1->Get_Value(),
            output->Get_Value());
};
```

Figure 28: Code Implementing INV Class

```
class Driver : public Simulator {
 private:
  assoc_list *driver_ports;

 public:
  Driver(assoc_list *p) : Simulator() { driver_ports = p; }

  #include "run_driver.cc"

};
```

Figure 29: Code Implementing Driver Class

```
class Net_Class: public Basis {
 private:
  int pending_event=0;    /* is an event scheduled to run? */
  int Tristate=0;          /* is the net connected to a tbuf? */
  int rise_time= -1;       /* last time the net changed to high */
  int fall_time= -1;       /* last time the net changed to low */
  int type= SIMPLE_NET;   /* SIMPLE_NET or BUS_NET */
  union {
    struct {      /* SIMPLE_NET */
      Sig_Class value;
    }simple;
    struct {      /* BUS_NET */
      Net_Class **net_array;
      int size, offset;
    }bus;
  }un;

 public:
  Queue Sources, Sinks;  /* queues of objects that drive or receive data */

  /* constructors */
  Net_Class(char *n, int id);
  Net_Class(char *n, int id, int size_in, int offset_in);
  /* destructor- deallocates all sinks, sources, and the net_class itself */
  ~Net_Class();

  void Add_Source(Basis *obj);
  void Add_Sink(Basis *obj);

  /*  Get/Set private variables,  procedures excluded here for brevity */

  Basis * Get_Sink(Basis *obj);
  Basis * Get_Source(Basis *obj);
  void Post_Bus_Value(int bits, int num);
  void Post_Bus_Uniform_Value(int num);
  void Post_Uncond_Bus_Value(int bits, int num, int etime);
  void Post_Uncond_Bus_Uniform_Value(int num, int etime);
  void Enq_Sinks();
  void Print_Info(FILE *out);
  void Print_Rnl(FILE *out);
  void Print_Sources();
  void Print_Sinks();
  void Process_Input();
  Basis * Remove_Source(Basis *src);
  Basis * Remove_Sink(Basis *sink);
};
```

Figure 27: Definition of Net_Class

The ID number of the object should be unique. A global procedure named Generate_ID() returns a unique, monotonically increasing integer for that purpose.

5.3 Derived Objects

Figure 26 and *Figure 27* show how objects are derived from the Basis class. *Figure 28* shows the source code that implements an inverter. Each simulation object has its own internal input/output/tri-state pointers to *Net_Class*es, and its own private internal value. The `Process_Input()` procedure performs the object function and enqueues a new event if necessary.

The `Net_Class` uses a union to store either simple (one-bit) nets or buses (arbitrary size). A bus is represented as an array of pointers to simple nets. The `Net_Class` structure contains queues of source objects and sink objects. In the simulator, when an object is declared, it must connect itself to input nets (the object is acting as a sink), or connect itself to output nets (source object). This information is then used by the simulator to determine successive propagations of signal changes. It is necessary to store a net's last rise time and fall time so that clocked objects can detect rising or falling edges. The other private variables of the class should be self-explanatory.

```
class INV : public Basis {
 private:
  Net_Class *input1, *output;
  int value=0;

 public:
  /* Constructor */
  INV(char *n, Net_Class *out,  Net_Class *i1);

  void Process_Input(); /* update internal value */
  void Print_Info(ostream *out);

};
```

Figure 26: Definition of INV Class

Almost all simulation objects are defined similarly. In the case of gate definitions, instead of specifying a different simulation object for every kind of gate, two general-purpose gates, GATE2 (2-input gate) and GATE5(5-input gate) were defined. These objects were initialized with the appropriate function pointer. 3- and 4-input gates were then implemented as 5-input gates with the extra inputs assigned (HIGH or LOW) in a manner that would leave the behavior of the gate unaffected.

Driver objects were defined differently. Since the driver object must have control of the simulation, it is derived from the Simulator class, as shown in *Figure 29*. It was necessary to #include the main `void run()` procedure, which is defined in the file *run_driver.cc* so that *xnfwirec* would compile it at the appropriate time. There is an association list named `driver_ports` containing net name/net pointer pairs. There is no distinction made between inputs and outputs, so it is up to the programmer to use the nets in the appropriate manner.

5.2 CLASS Basis

It was necessary to establish a common class for all objects in the simulator so that objects may be handled in a uniform manner. All objects and nets in XS are derived from the `Basis` class. This class, shown in *Figure 25*, contains information that is inherited by all simulation objects and nets. In the Basis class, there

```
class Basis {
 private:
  char * name;  /* String Identifier */
  int type;      /* Object type */
  int ID;         /* Unique identification number */
  int delay;     /* Signal propogation delay through this object */

 public:
  /* Constructors */
  Basis();
  Basis(int t, int d, char *n);

  /* Destructor */
  virtual ~Basis();

  int Get_Delay();
  int Get_ID();
  int * Get_ID_Addr();
  char * Get_Name();
  virtual int Get_Outvalue(int dummy);
  int Get_Type();

  virtual void Print_Info(ostream *out);

  /* Process_Input() -- Update internal data based on new input */
  virtual void Process_Input();

  /* Reset() -- response to a reset signal */
  virtual void Reset();

  void Set_Delay(int del);
  void Set_ID(int id);
  void Set_Name(char *s);
  void Set_Type(int t);
  void Set_Params(char *n, int t, int d);
};
```

Figure 25: Definition of class Basis

are several `virtual` functions *Print_Info(), Print_Rnl(), Process_Input(), Reset(), Get_Outvalue()*. These functions allow each derived object to take its own specific action when the function is called. *Print_Info()* sends object information to an output stream. *Print_Rnl()* sends RNL-compatible data to the RNL file, which contains the simulation trace. *Process_Input()* performs allows the object or net to respond to a change in the inputs. Finally, *Get_Outvalue()* is used to resolve the proper value of tri-state nets.

```
//  Pseudocode showing event generation and processing.

//  in simulator.cc, Simulator::Run()
//  ---------------------------------
while (Simulation_Not_Done) {

  //  in simulator.cc, Simulator::Cycle()
  //  -----------------------------------
  while (Current_Cycle_Not_Done) {

    //  in simulator.cc, Simulator::Simulate_Step()
    //  -------------------------------------------
    while (EventQ.Not_Empty && (EventQ.Next_Time() <= Get_Sim_Time())) {
      ev = EventQ.Deq();
      ev->Get_Net()->Set_Value(ev->Get_Value()); //  update net value
      Changed_Nets.Enq_Head(ev);
    }

    while (Changed_Nets.Not_Empty())  {
      net = Changed_Nets.Deq_Tail();

      //  in net_list.cc, Net_Class::Enq_Sinks()
      //  --------------------------------------
      for (obj=net->Get_First_Sink(); obj=net->Get_Next_Sink();
           obj!=net->Get_Last_Sink()) {

        if (obj->Get_Type()==_NET) EventQ.Post_Event(net,value,delay);
        else {
          //  in xnfobjects.cc, Basis::Process_Input()
          //  ----------------------------------------
          //   -- function performed depends on the current object type
          value = object_function();
          EventQ.Post_Event(net,value,delay);
          }
      }
    }

  }
}
```

Figure 24: Event Generation and Processing

```
class Event_Class {
 private:
  Net_Class *net;   /* a net which will be changed */
  int time;         /* time at which the net will change value */
  int new_value;    /* value the net will  change to */

 public:
  /* Constructor */
  Event_Class(Net_Class *n, int nv, int t);

  Net_Class *Get_Net();
  int Get_New_Value();
  int Get_Time();

  void Set_Net(Net_Class *n);
  void Set_New_Value(int i);
  void Set_Time(int t);

  /* -------------------------------------------------------------------------
  /* Enqueue all events that will happen when net changes.  These events
    will be all wires that are connected to the "Sinks" of the current
    net.  This procedure call indirectly calls Process_Inputs, which will
    perform the actual event posting via Post_Event().  */
  /* -------------------------------------------------------------------------
  void Enq_Sinks(Net_Class *net);
};
```

Figure 23: Event_Class Definition

4 Performance

The performance of XS varies greatly depending on the primitives which are used. By far the most efficiently processed objects are FOREIGN .eqn devices. This is because each line in an .eqn file is processed as a whole, rather than as separate gates. This way, there is a substantial decrease in event-related runtime overhead. Gates are the next most efficient objects. Finally, CLBs are the least-efficiently processed, due to their complexity. It is expected that very few designs will use CLBs directly, and that FOREIGN .eqn files will be most common.

The table below shows the performance of the simulator on various benchmarks. It is worthwhile to note that the "eps" (events per second) rating is not a very accurate performance metric. This is because .eqn files generate less events, yet run significantly faster. In addition, XS does extensive filtering of unnecessary events, and so the eps rating may not be analogous to that of other simulators.

Benchmark	Comments	# of Nets	# of Objects	# of Events	EPS	RunTime(s)
eqnfa	half gate half .eqn	160	102	147490	2344	62.9
eqnfa	.eqn version	137	64	45563	3616	12.6
tetris	gate version	822	834	116364	689	168.8
tetris	.eqn version	717	653	93988	3494	26.9
traffic	.eqn version	28	19	5394	1860	2.9
treecomp	.eqn version	145	109	212485	4691	45.3

A number of things were done to enhance the speed of XS. First, it was found that file I/O was a major bottleneck. The C++ streams were replaced with the more efficient standard input/output C routines. Next, the watch commands were streamlined so that net information was only printed upon a change in the net, rather than at every simulation cycle. These changes roughly tripled the simulation speed. Another improvement was inlining various queueing functions. The final improvement was considering each line of an .eqn file as one large object. This resulted in another factor of four speed improvement (but only for designs which used .eqn files).

5 Design Details

This section of the paper reveals some of the algorithms and data structures used in the simulator. It is intended to give the reader some insight into how the simulator works, and should allow the reader to understand and modify the source code more readily.

5.1 Events

An event is considered to be a change in a net value at a given time. When such a change occurs, the event is placed onto an event queue. This queue is a time-ordered doubly linked list. Events may be generated by a FORCE command in *test.script*, by an object responding to an input change, or by a Driver. The data structure for the event class is given in *Figure 23*. The `Enq_Sinks()` procedure is the key procedure used by the event-processing loop. It causes all of the sink objects of the given net to `Process_Inputs()`, which in turn will post new events for the next simulation timestep. The main simulation loop, described in Section /refsimulator controls the event processing procedure, however, many of the details are hidden within procedure calls. *Figure 24* shows the finer details of how events are generated and processed.

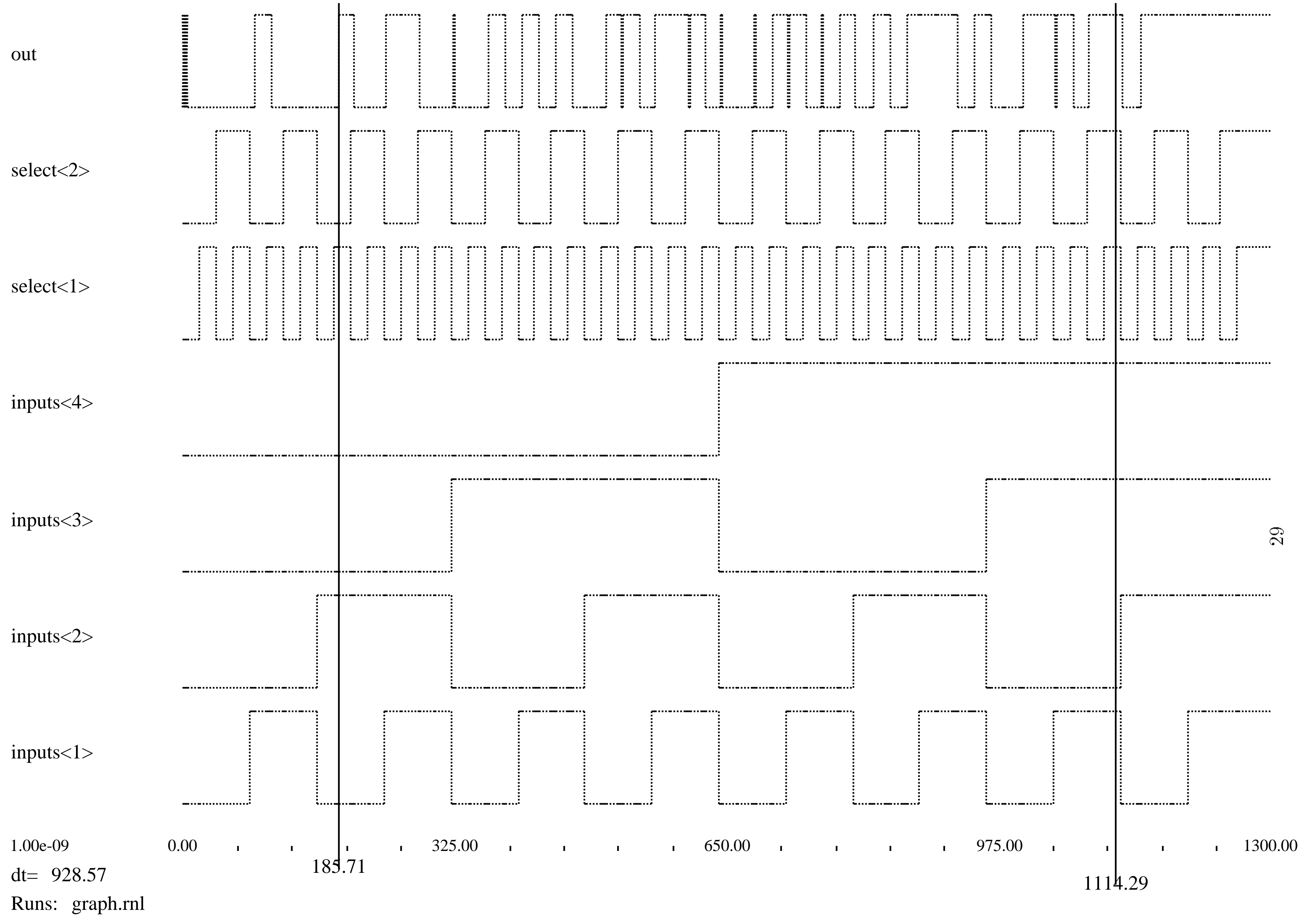

Figure 22: N_MUX: Simulation Results

```
void run() {

    int notdone=1, data, select, time=0;

    fprintf(output_st,"RUNNING DRIVER!!!!\n");

    // Put the events on the event queue.
    for (data=0; data<16; data++) {

      Change_Bus_Output_Abs(4,data,"data",time);

      for (select=0; select<4; select++) {

      Change_Bus_Output_Abs(2,select,"select",time);

      time+=20;
      }
    }
    // Simulate, 20 cycles at a time!
    while (notdone) {

      // Now let the simulator run for a while & settle
      notdone=Cycle(20);

    }
  }
```

Figure 21: N_MUX: Driver Implementation

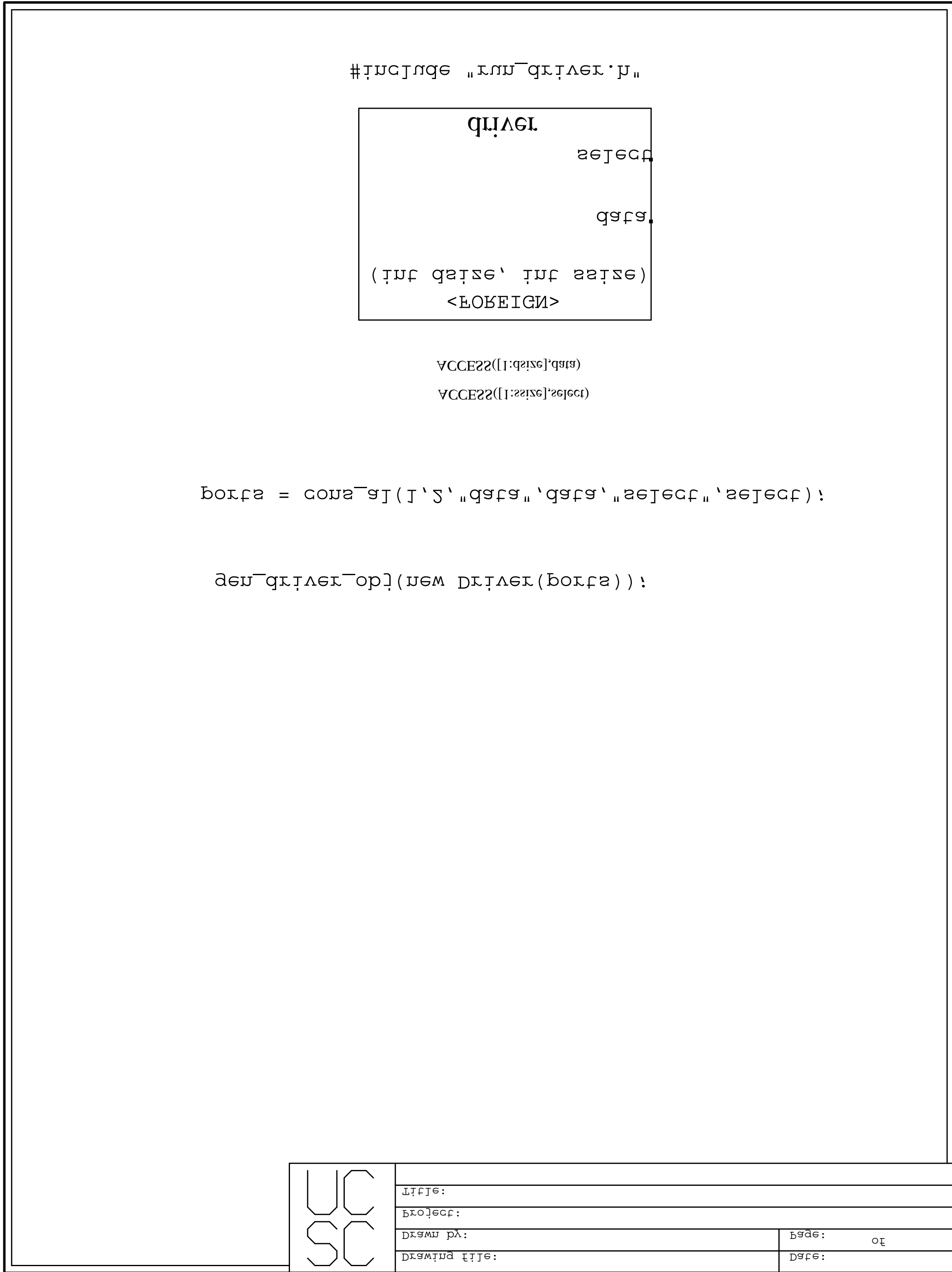

Figure 20: N_MUX: Driver Definition

Figure 19: N_MUX: Bottom Level

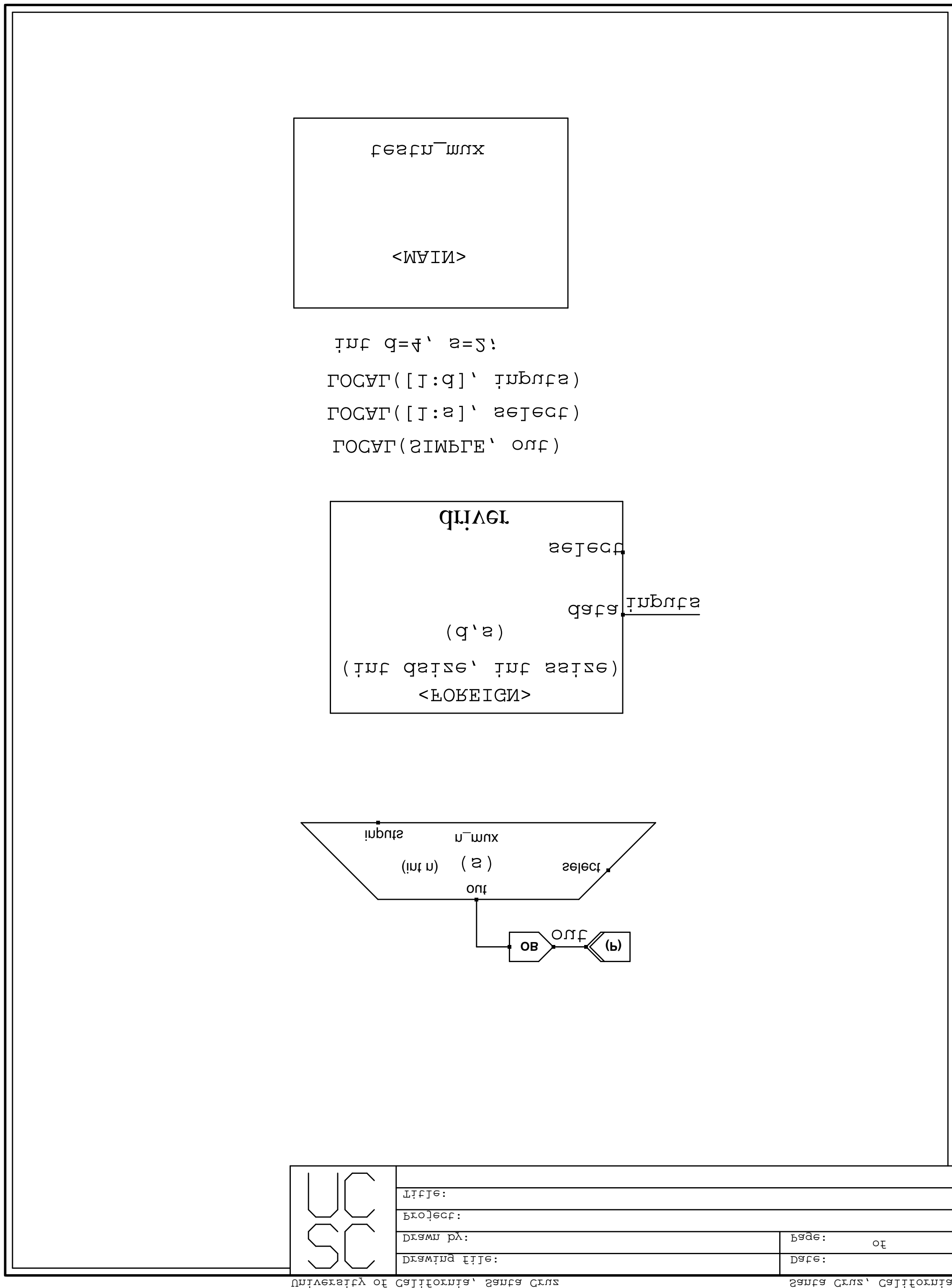

Figure 18: N_MUX: Top Level

3.3 nmux: A Driver Example

The final example demonstrates the use of a Driver. Here, the Driver is used to generate a complete set of test vectors for a four to one multiplexor. *Figure 18* shows the top level of the design. A Driver object should be at the top level of the design. The design of the mux is shown in *Figure 19*. It has been recursively defined so that any n-to-1 mux (where n is a power of 2) may be created. *Figure 21* is the C++ code for the Driver. It first places all of the necessary testing events onto the event queue, and then it runs the simulation until there are no events left. One VERY IMPORTANT thing to know is that the only way to place multiple events associated with the same net onto the event queue is to use the CHANGE_xxxx_ABS() commands. These commands put a command on the queue at the given time (relative to time zero, rather than relative to the current time) whether or not an event for the net already exists. The CHANGE_xxxx() commands will not enqueue multiple events on the queue, in order to improve the event-processing efficiency of the simulator.

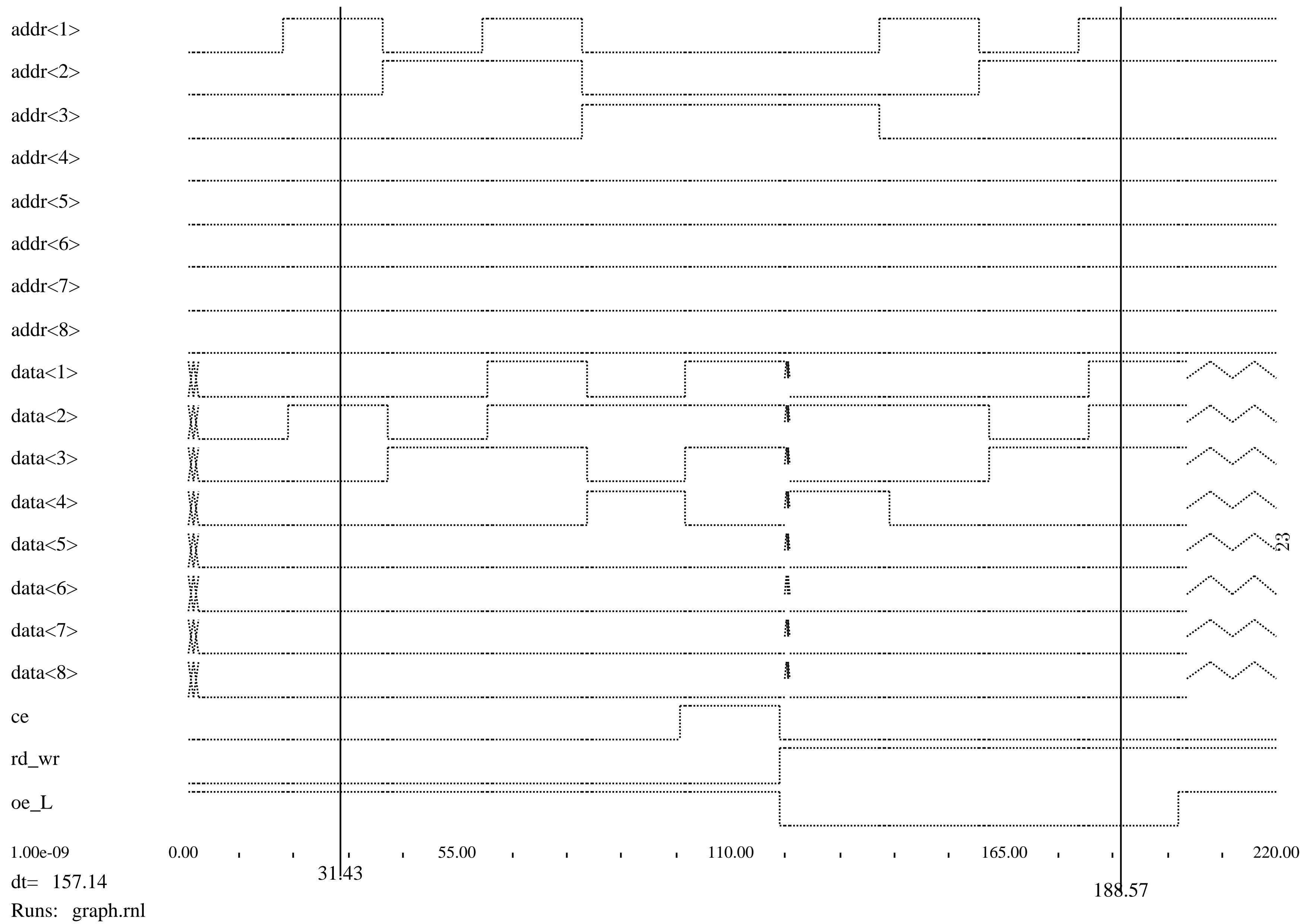

Figure 17: Memory: Simulation Results

```
def_bus 3 cerwen ce rd_wr oe_L
def_bus 8 din din<1> din<2> din<3> din<4> din<5> din<6> din<7> din<8>
def_bus 8 data data<1> data<2> data<3> data<4> data<5> data<6> data<7> data<8>
def_bus 8 addr addr<1> addr<2> addr<3> addr<4> addr<5> addr<6> addr<7> addr<8>

watch cerwen 0 220
watch data  0 220
watch addr 0 220

// Write mem[0]=0
force din 0 1
force addr 0 1
force ce 0 1
force rd_wr 0 1
force oe_L 1 1

// Write mem[1]=2
force din 2 20
force addr 1 20

// Write mem[2]=4
force din 4 40
force addr 2 40

// Write mem[3]=7
force din 7 60
force addr 3 60

// Write mem[4]=10
force din 10 80
force addr 4 80

// Turn off chip enable, Will NOT write 7 into addr[4]
force din 7 100
force ce 1 100
force rd_wr 0 100

// Read back what's in mem[4] (should be 10)
force ce 0 120
force rd_wr 1 120
force oe_L 0 120

// Read back  mem[1] (should be 1)
force addr 1 140

// Read back  mem[2] (should be 2)
force addr 2 160

// Read back  mem[3] (should be 4)
force addr 3 180

// Turn off output enable, to get ZZZZZZZ
force oe_L 1 200
```


Figure 16: Memory: test.script

```
//  mem_proc.cc -- a c-procedure which pretends to be a memory chip
#include <stdio.h>

//  cc_obj_help.h -- This contains useful interface routines for
//  handling net value accesses.
#include "cc_obj_help.h"

//  These input parameters are standardized!!!
int mem_proc(assoc_list *outs, assoc_list *ins, assoc_list *constants)
{
  char *name;
  int dinval, doutval, ain, i;
  static int initmem=0, words=0, bits=0, *memory;

  //  initialize the memory - Put i into memory[i];
  if (!initmem) {
    initmem=1;
    words = (1<<(Access_Constant("a_bits")));
    bits = Access_Constant("d_bits");

    if (bits>32) {
      fprintf(output_st,"ERROR: Unable to make memory with words>32 bits!\n");
      exit(-1);
    }

    //  we have 1<<'a_bits-1' words of memory
    memory = (int *) malloc(words*sizeof(int));

    for (i=0; i<words; i++) { memory[i] = i; }
    Bus2Value("data_out",ZZZZZZZ);
  }

  //  IF chip is active (ce_L==0) then, we can continue;
  if (Access_Value("ce_L")==0) {

    dinval = Access_Bus_Value("data_in");
    ain = Access_Bus_Value("addr");

    if (Access_Value("readnotwr")==0) {
      //  *** WRITE mode, D=input ***;
      memory[ain] = dinval;
      Bus2Value("data_out",ZZZZZZZ);
    }
    else {
      //  *** READ mode, output=ZZZZZZZ ***;
      doutval = memory[ain];
      Change_Bus_Output(bits,doutval,"data_out");
    }
  }
}
```

Figure 15: Memory: mem_proc.cc – C_code Implementation

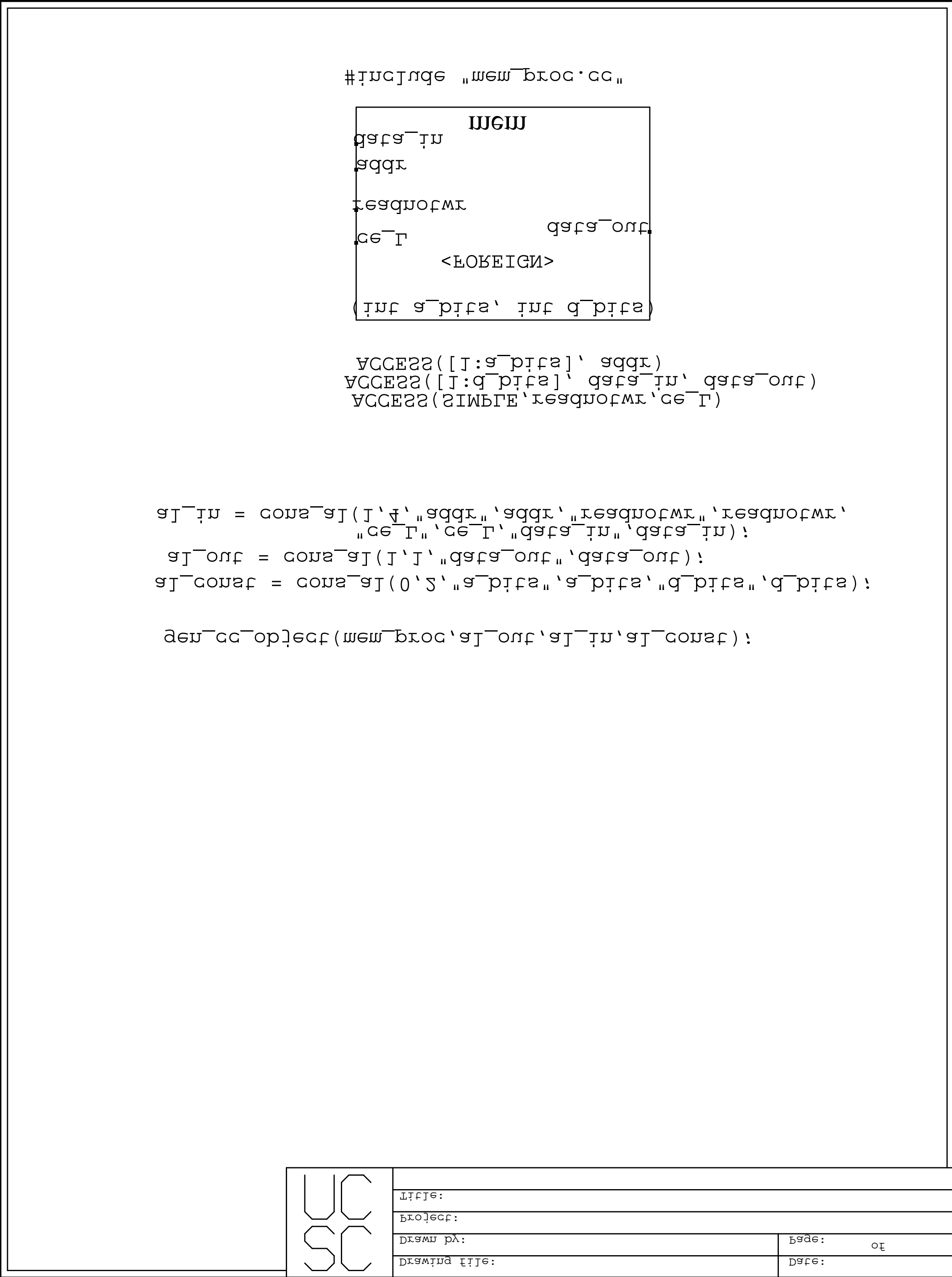

Figure 14: Memory: Device Lower Level

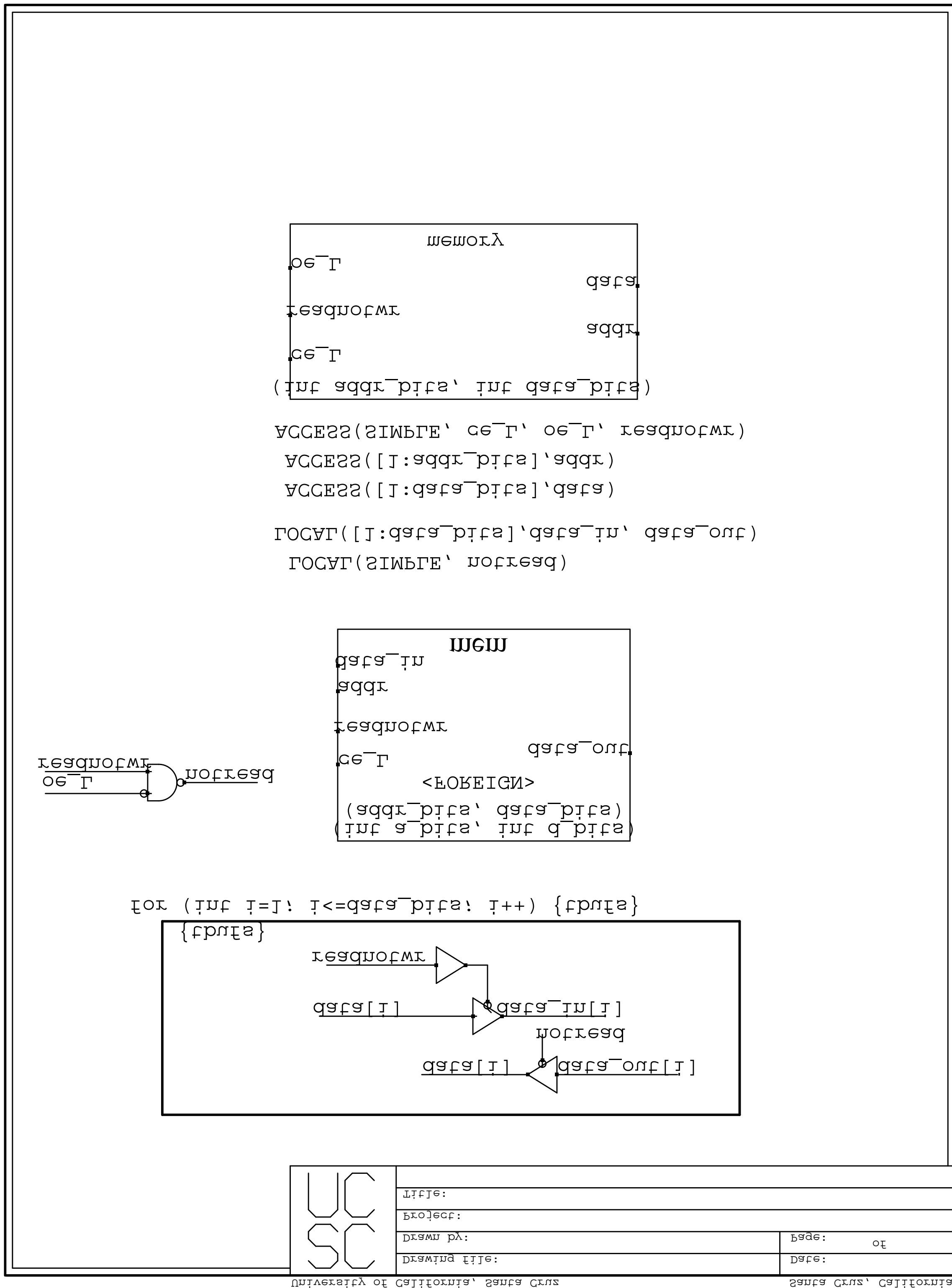

Figure 13: Memory: Device Top Level

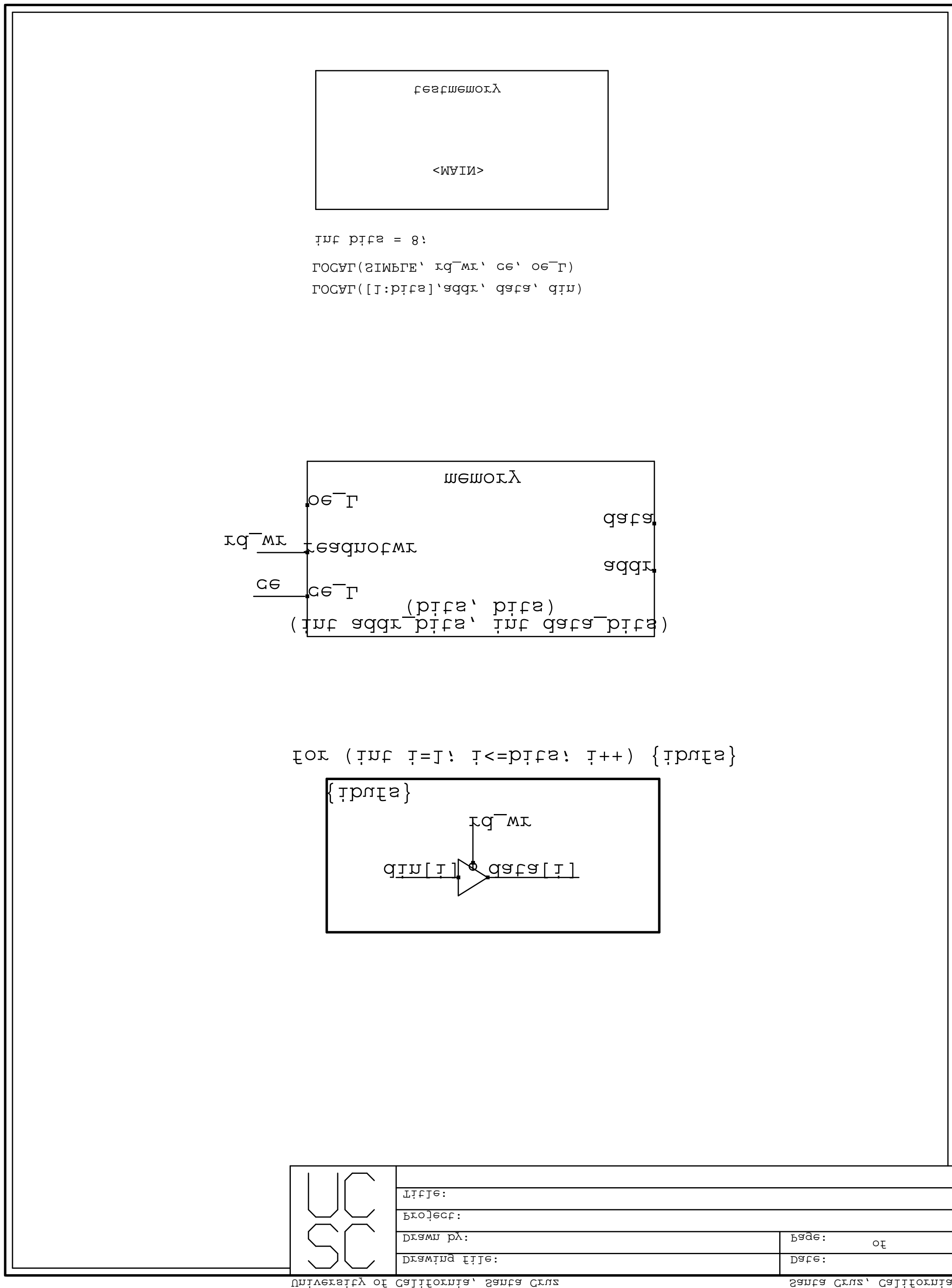

Figure 12: Memory: Top Testing Level

3.2 A C_code Example: RAM Chip

The next example uses a C_code object to implement a random access memory chip (*Figure 12*, *Figure 13*, and *Figure 14*). C_code objects cannot use tri-state nets, however, that limitation is easily avoided by creating an extra level in the hierarchy where tri-state buffers are used at the input/output interface of the C_code object (see *Figure 13*). When it is necessary to FORCE the value of a tri-state net, it is important to make sure that the net being FORCEd is on the input of a TBUF. In *Figure 12*, if one were to try to FORCE data[i], the simulation would be incorrect. This is because when a tri-state net value is updated, it checks all of the object sources to the net to see what each source is generating. A dangling wire or a source without a valid value would cause incorrect values to appear.

Figure 15 is the complete description of the behavior of the memory chip. *Figure 16* is the simulation control script. Note that once a value is FORCEd, it remains at that value unless another FORCE command overrides it. FORCE should only be used on input nets.

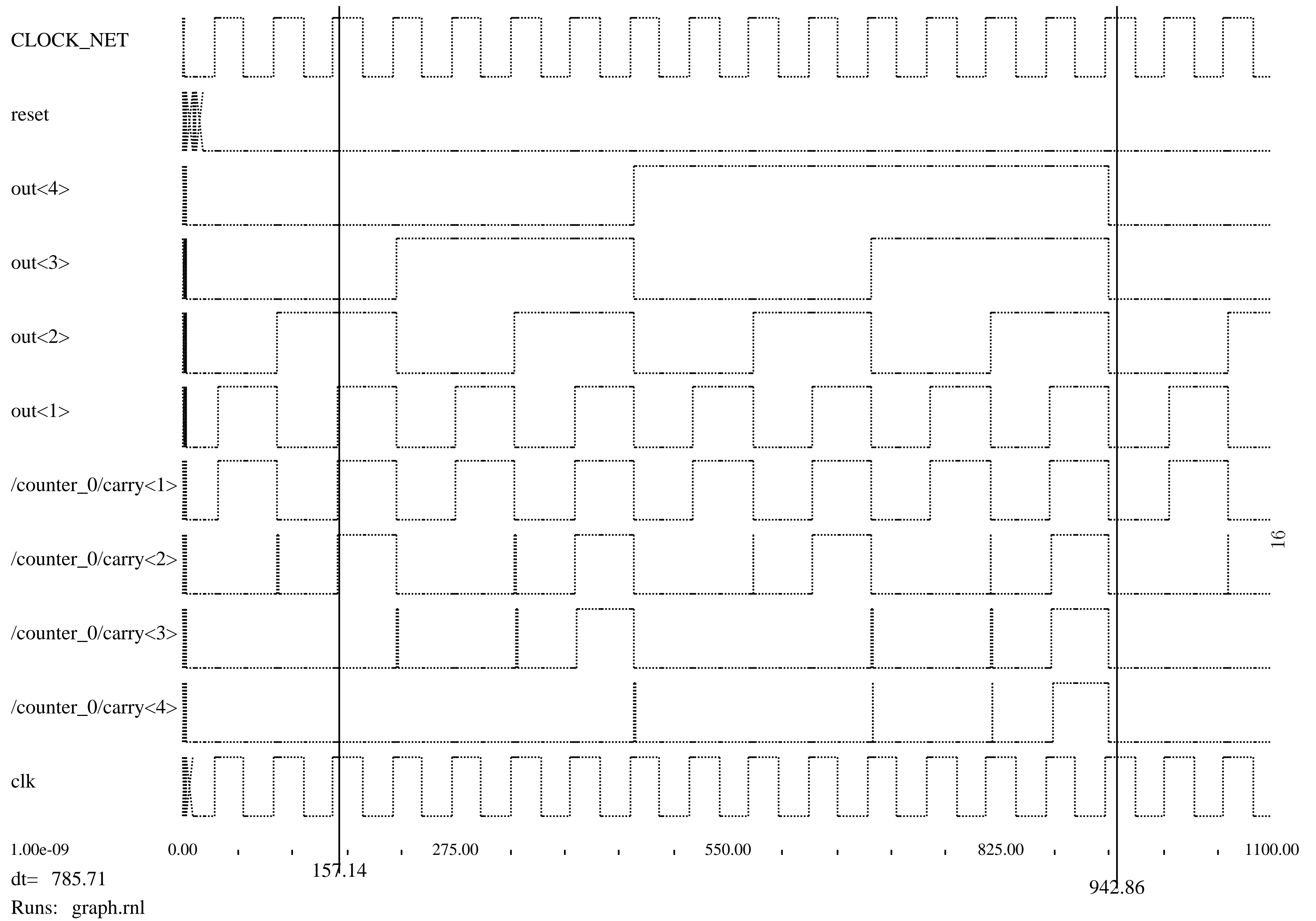

Figure 11: Counter: Simulation Results

```
start_clock 0
stop_clock 1100
connect_net CLOCK_NET, clk, 10;

#  reset is active high
force reset 0 20

def_bus 4 out out<4> out<3> out<2> out<1>

watch clk,0,1100

# note that separators need not be whitespaces...
#  MUST use the hierarchical net name!
watch /counter_0/carry<4> 0 1100
watch /counter_0/carry<3>,0,1100
watch /counter_0/carry<2>; 0; 1100
watch /counter_0/carry<1> 0 1100;

watch out 0 1100


w reset 0 1100
w CLOCK_NET 0 1100
```

Figure 10: *test.script* File for Counter

Figure 9: Counter: Lower Level

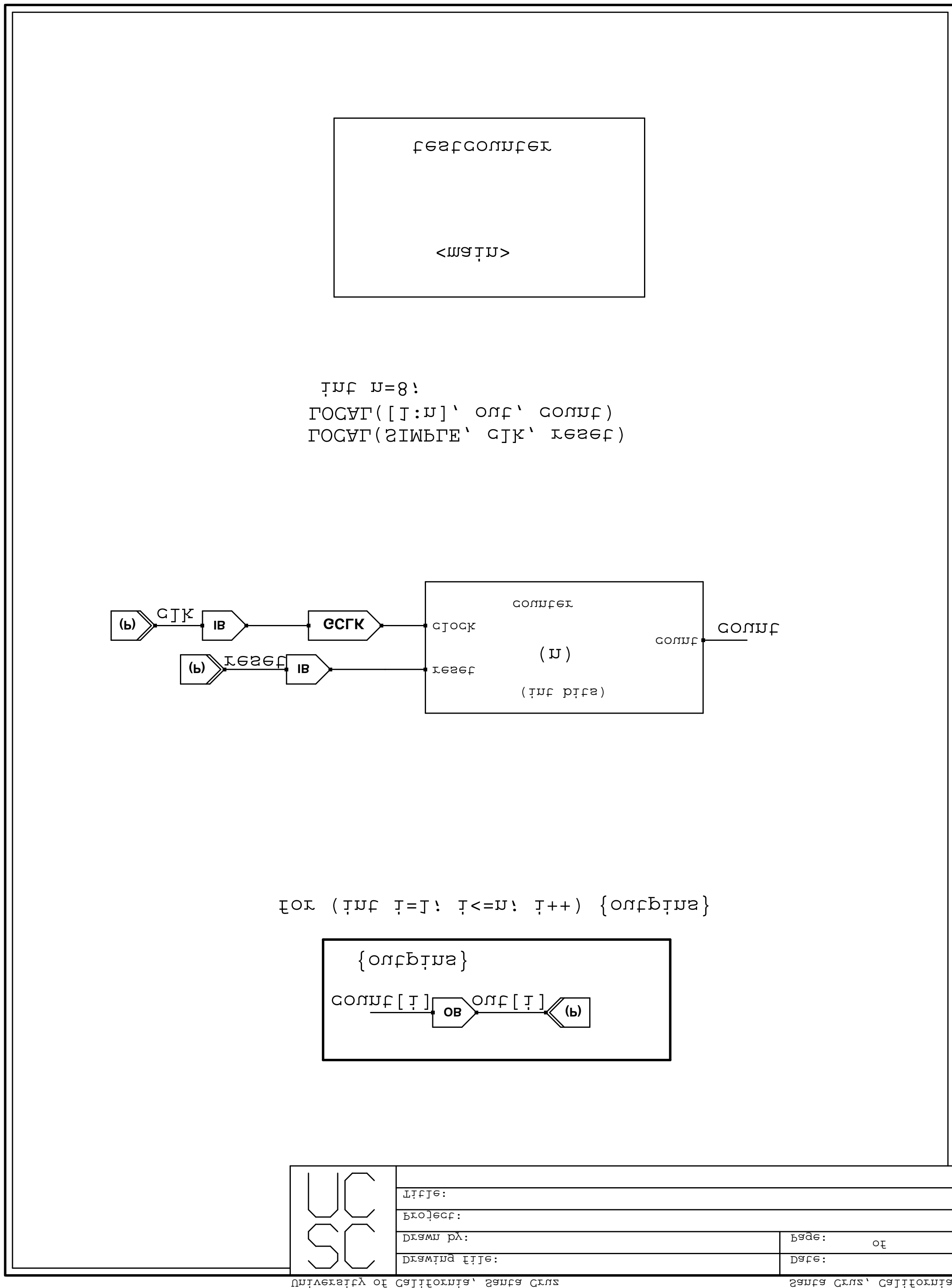

Figure 8: Counter: Top Level

2.9 Miscellaneous

There are several pre-defined objects and nets which the simulator uses internally. The user should avoid defining any objects or nets of the same name in their schematics. These include: CLOCK_OBJ (a clock), CLOCK_NET (the net containing the clock output signal), HIGH_NET (a net that always has a high logic value), and LOW_NET (a net that always has a low logic value). These objects are for internal use only, and should never be used by the user, except for CLOCK_NET, which should only be used in *test.script* to connect the clock signal to the user's circuit.

The FORCE command in *test.script* should only be used on nets which have no source objects, or which are connected to input pin sources. On nets which have source objects, after the FORCE has taken place, the net value might be changed by the source object. A FORCE command should never be used on a dangling tri-state net. Instead, one should FORCE the value of a normal net which is the input to a TBUF (tri-state buffer).

One additional file which the simulator produces is *netnames.txt*. This file contains a list of all of the nets used in the system. One should consult this file to determine the exact hierarchical name of any nets that are to be WATCHed.

3 Examples

This section contains several examples showing how the simulator is used. All of the *XDP* files and simulation files are included here.

3.1 A Clocked Example: Counter

The first example is a simple binary counter. The counter drawing is shown in *Figure 8* and *Figure 9*. Notice that the lower level counter definition is a not dependent on any XILINX-specific parts, or tied to any external devices (such as input/output buffers, IOBs). This allows the lower level counter to be imported as a generic part in many designs, without need for editing the description. *Figure 10* shows the simulation control file. For clocked designs, it is necessary to specify the clock starting and ending times, and also to specify the net to connect the clock to. This is shown in the first three lines of *test.script*. Then various nets are watched and forced. Every script command is read in and placed in a command queue before simulation begins, so it is not important what order the commands occur in the script. However, for WATCH commands, it is nice to group buses together or in sequential order so that they appear together.

COMMAND (synonym)	*Argument 0*	*Argument 1*	*Argument 2*
connect_net (cn)	src. net name	dest. net	time to start
connect_obj (co)	src. net name	dest. obj	time to start
def_bus (db)	bus size	bus name	arg2...argn: net name
disconnect_net (dn)	src. net name	dest. net	time start
disconnect_obj (do)	src. net name	dest. obj	time start
force (f)	net name	value	time to start
set_clock_period (sp)	time to set	new period	
start_clock (sc+)	time to start		
stop_clock (sc-)	time to stop		
watch (w)	net name	time to start	time to end
// (/*)(#)	Comment	- ignore current line	

Each command must be on a new line, and each argument must be separated by either a space, tab, or one of the following characters: `, ; : ( )`

The test script must contain two carriage returns (or two blank lines) at the end of the file. Commands are not case-sensitive.

2.7 The *.xsinit* Initialization File

If the file *.xsinit* is present in the working directory, the simulator will read this file and set various global parameters. Commands in this file consist of a variable name and a new value to assign that variable. These variables are (with their default values):

output_stream stdout

rnl_stream graph.rnl

error_stream stderr

history_stream /dev/null

silent_mode 1

The stream `output_stream` is used for printing out general information by the simulator, such as the current simulation time. `rnl_stream` is the stream which outputs a file compatible with *sigview* that contains the simulation data for all of the nets tagged by the WATCH command in *test.script*. It should be given a filename which ends in `.rnl` so that *sigview* will know which file format it uses. `history_stream` outputs a list of the number of events which occurred at each simulation cycle. The history stream will produce a *sigview*-compatible *SPICE* file, so if it is to be directed to a file, one should use a `.spice` extension.

2.8 Foreign Devices (.eqn Files)

A Foreign Device is a boolean equation (.eqn) file. Each line in the file is converted into a function tree which is considered to be one macro-gate object. This tree is given a unit-delay evaluation time, since it has been experimentally determined that each line in an .eqn file roughly corresponds to a CLB. The function tree supports AND, OR, XOR, XNOR, INV, and BUF functions. Evaluation is done recursively. At any given level, the parent first evaluates all of its children and then does its own function evaluation, returning the result. Since many potential evaluations are done all at once, avoiding queueing overhead, this significantly increases the performance of the simulator.

```
//  run_driver.cc -- A minimal driver... This does nothing...

//  *Always* must use the procedure name: void run()
void run() {

  //  Declare your variables here
  int notdone=1;

  //  Simulate, 1 simulation cycle at a time!
  while (notdone) {

    //  Now let the simulator run for a simulation cycle
    notdone=Cycle(1);

  }
}
```

Figure 7: An Example of a Driver Procedure

int Cycle(int i) – Perform i simulation steps, update the simulation time, execute items in the command queue, and return a 1 if the simulation is not done, and a 0 if the simulation is done.

int Compute_Next_Time() – Compute the next time that a simulation event should occur and return that time. If nothing is on the event and command queues, return a -1. Use only with Simulate_Step().

void end_globals() – Signal the end of a simulation, by flushing the output streams and deallocating all simulation data structures.

void Read_Commands() – Read all commands from file "test.script" and enqueue them onto the command queue. Use only with Simulate_Step().

void Simulate_Step() – Perform one simulation step time advancement. Dequeue all events to be run at the current time and update the net values. Then Enq_Sinks() of all nets which changed to generate the next round of events. *Not* recommended for general use. Use Cycle() instead.

***output_st** – A stream used for general output.

***error_st** – A stream used for error and warning output.

Currently, the bus manipulation routines are limited to 32-bit buses.

2.6 Controlling the simulation with *test.script*

There is a special file named *test.script*, which is used to control the simulator. This batch file is a sequence of commands which tells the simulator what to do. This file is read in before the start of simulation, and all of the commands are placed on a command queue. Each command is executed by the simulator at the specified time. The following is a list of all of the currently supported commands:

```
#include "run_driver.h"

driver
select
data
(int dsize, int ssize)
<FOREIGN>

ACCESS([1:dsize],data)
ACCESS([1:ssize],select)

ports = cons_al(1,2,"data",data,"select",select);

gen_driver_obj(new Driver(ports));
```

UC SC

Title:
Project:
Drawn by:
Page: of
Drawing file:
Date:

University of California, Santa Cruz
Santa Cruz, California

Figure 6: An Example of a Driver Object

```
//  Filename:  my_ccode_proc.cc

#define UNITDELAY 1

#include <stdio.h>

//  cc_obj_help.h -- This contains useful interface routines for
//  handling net value accesses.

#include "cc_obj_help.h"


//  These input parameters are standardized!!!
int my_ccode_procedure_name(assoc_list *outs, assoc_list *ins,
                            assoc_list *constants)
{
  //  Declare variables here, use static if they are to be persistent

  int constantvalue, input1value, input2value;
  int output1value;

  //  Your C++ Code goes here...
  //  MUST be reentrant code!!!

  //  Accessing a constant
  constantvalue = Access_Constant("const2");

  //  Accessing a bus net
  input1value = Access_Bus_Value("input1");

  //  Accessing a simple net
  input2value = Access_Value("input2");

  output1value = (input1value > input2value)

  //  Changing an output net
  Change_Output("output1",output1value,UNITDELAY)
}
```

Figure 5: An Example of a C_code Procedure

```
#include "my_ccode_proc.cc"

my_ccode_obj
input1
input2
output1
<FOREIGN>
(int const1, int const2)

ACCESS([1:const1], input1)
ACCESS(SIMPLE, input2, output1)

al_in = cons_al(1,2,"input1",input1,"input2",input2);
al_out = cons_al(1,1,"output1",output1);
al_const = cons_al(0,2,"const1",const1,"const2",const2;

gen_cc_object(my_ccode_procedure_name,al_out,al_in,al_const);
```

UC SC
Title:
Project:
Drawn by: Page: of
Drawing file: Date:
University of California, Santa Cruz Santa Cruz, California

Figure 4: An Example of a C_code Object

```
(Assoc_List *) cons_al(int isanet, int numberofassociations, ...)
```

creates the specified association list. The first parameter, *isanet*, is needed to distinguish between input/output nets and constant values. This allows the simulator to add the nets to the netlist. Nets may be either SIMPLE or BUSes. The second parameter, *numberofassociations*, is the number of signal name/signal pointer pairs that are specified. There are NO RESTRICTIONS on the number of inputs/outputs/constants allowed. After the association lists are constructed, `gen_cc_object()` must be called to instantiate the C_code object. Note that there are no constraints on the names of the C++ procedure name (except that the C++ procedure name must match the first parameter in the call to `gen_cc_object()`) , the input/output/constant names, or the object block name.

The C_code procedure file must contain the include file declaration `#include ''cc_obj_help.cc''` in order to gain access to the netlist interface commands.

In Section 3.2, an example C_code object is implemented.

2.5 Driver Objects

A Driver object is a mechanism for allowing some piece of C++ code to take over the control of the simulation engine. This is useful in representing some external object which communicates with the circuit being simulated. Since the Driver takes over control of the simulator, it may even provide its own output display mechanism. The procedure for creating a Driver object is very similar to that of a C_code object. *Figure 6* shows a generic Driver object, and *Figure 7* shows a generic Driver procedure.

In order to make a driver object, one must copy the file *run_driver.h* into the current directory. This provides the interface routines between the simulator and the C++ driver code. Then, a file named *run_driver.cc* must be created, containing the controlling procedure named `void run()`. This procedure functions as the main() procedure. The following are the interface routines which may be used by the driver:

int Access_Value(name) – Get the value of a given simple net.

int Access_Bus_Value(busname) – Convert an input bus value into an integer and return that integer.

int Access_Sub_Bus_Value(name,i) – Get the value of a given bus sub-element

void Bus2Value(busname,value) – Set the specified bus to a certain single value (ie all elements of the bus are assigned 'value', which must be one of: 0, 1, XXXXXXX, ZZZZZZZ). Using Relative time. XXXXXXX (unknown value) and ZZZZZZZ (high impedance) are predefined constants.

void Bus2Value_Abs(busname,value) – Set the specified bus to a certain single value (ie all elements of the bus are assigned 'value', which must be one of: 0, 1, XXXXXXX, ZZZZZZZ). Using Absolute time.

void Change_Output(name,value,delay) – Post the event signaling that a SIMPLE output net has changed.

void Change_Sub_Bus_Output(name,value,delay) – Post the event signaling that a BUS sub-net output has changed.

void Change_Bus_Output_Abs(bits, num, busname,etime) – Convert an int to a bus value and post the change to the simulator. 'bits' specifies the number of valid lower bits of the integer. Using Absolute time.

Figure 3: Primitive XNF Devices

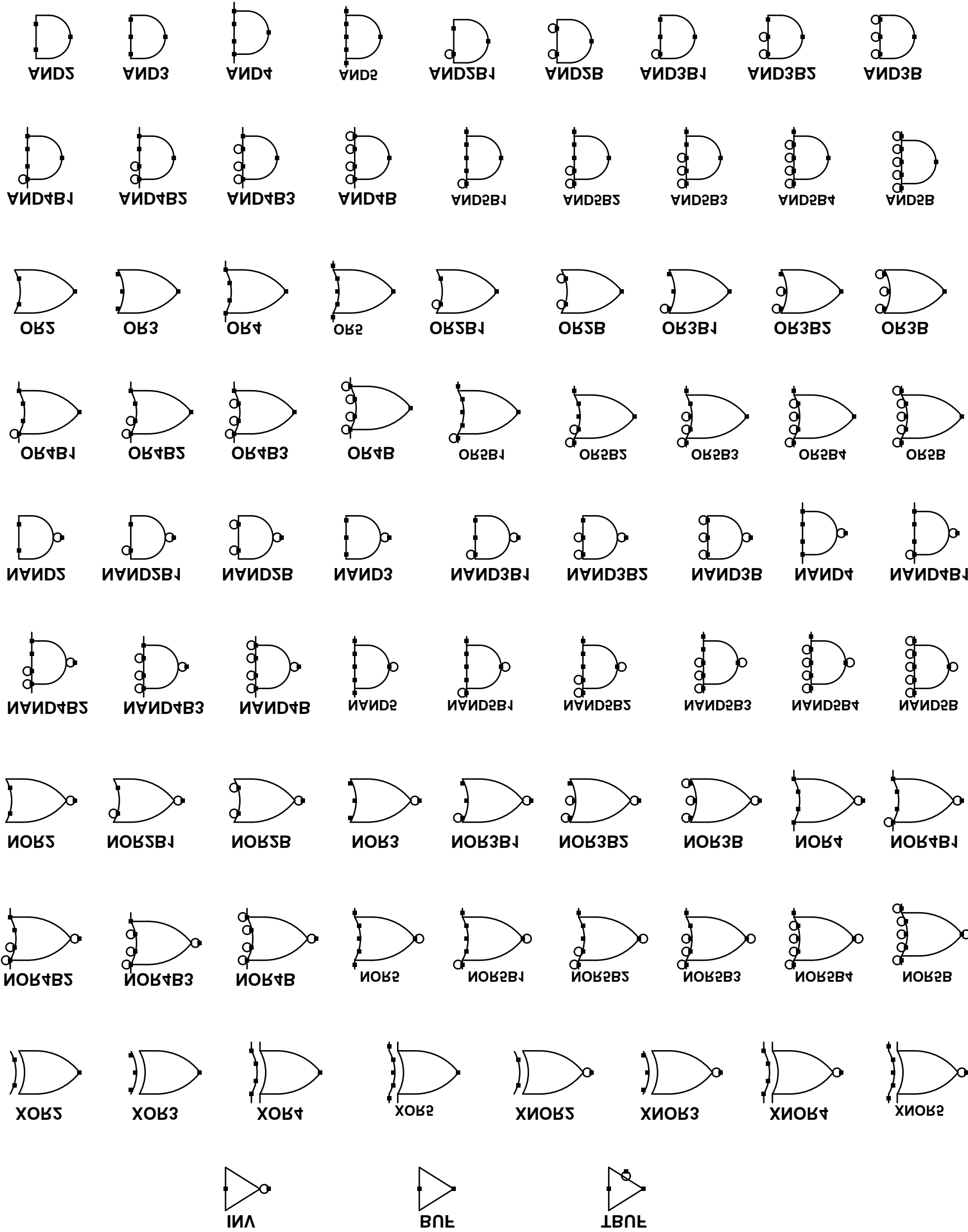

Figure 2: Primitive XNF Gates

- ACLK, GCLK are turned into buffers

In addition to the basic library of objects, the simulator also supports *C_code* objects and a *Driver* object. These will be discussed in Sections 2.4 and 2.5.

All of the gates, buffers, flip-flops, latches, and CLBs incur a unit propagation delay. C_code objects also have a unit delay, however, Driver objects may post events at any delay interval. Each line of a Foreign ".eqn" file is compiled into a function tree. The evaluation of an entire tree is given a unit delay. CLBMAPs and IOBMAPs maintain their gate-level descriptions for simulation purposes, and are not transformed into CLBs or IOBs by XS.

2.4 C_code Objects

XS has a provision for allowing a piece of C++ source code to function as a simulation object. This code must be reentrant (always starts execution at the top of the code) and may access connected wires through a special set of commands. If the code needs persistent data, then those data items must be declared `static`. C_code objects are useful in situations where one wishes to test a design which has not been fully implemented at the gate level. The C_code is used to mimic the desired object's function. Another use for C_code objects is to emulate some external device, for example, a memory chip.

The C_code interface procedures are:

int Access_Value(name) – Get the value of a given simple net.

int Access_Bus_Value(busname) – Convert an input bus value into an integer and return that integer.

int Access_Sub_Bus_Value(name,i) – Get the value of a given bus sub-element.

int Access_Constant(name) – Get the value of a given constant

void Change_Output(name,value,delay) – Post the event signaling that a SIMPLE output net has changed.

void Change_Sub_Bus_Output(name,value,delay) – Post the event signaling that a BUS sub-net output has changed.

void Change_Bus_Output(bits, num, busname) – Convert an integer to a bus value and post the change to the simulator. 'bits' specifies the number of valid lower bits of the integer.

void Bus2Value(busname,value) – Set the specified bus to a certain single value (i.e. all elements of the bus are assigned 'value', which must be one of: 0, 1, XXXXXXX, ZZZZZZZ). XXXXXXX (unknown value) and ZZZZZZZ (high impedance) are predefined constants.

Currently, the bus manipulation routines are limited to 32-bit buses.

A C_code object must be drawn in *XDP* in a very specific manner in order to provide XS with all of the necessary information for simulation. *Figure 4* is an example of a generic C_code object with two inputs, one output, and two constants. The associated C++ code is shown in *Figure 5*. There are several things to note. First, note that the object is declared as <FOREIGN>. Also, it is necessary to have the `#include ''my_ccode_proc.cc''` line ABOVE the object block.

There are three predefined association lists (al_in, al_out, al_tri) which are used to group together a signal name with its signal pointer. The predefined procedure

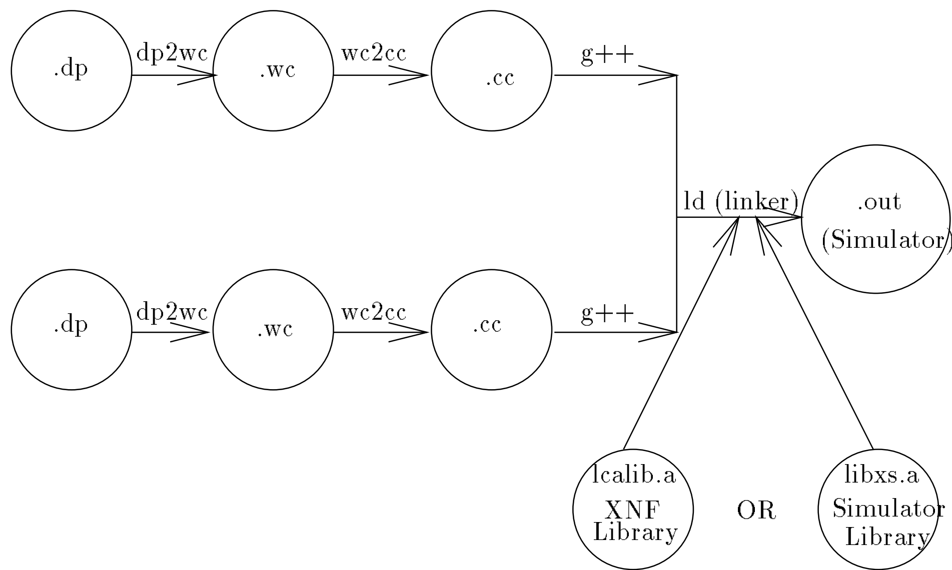

Figure 1: *wirec* Translation Process

2.2 Setup

It is assumed that the user is familiar with *XDP*, and the translation program *xnfwirec*. In order to run the simulator, it is necessary to set the environment variables WCINCLUDE and WCLIB to point to the directories containing the XS include and library files. Then, to run the simulator on an *XDP* file, an alias should be set up to call *wirec* with the `libxs` library:

```
alias xs ``wirec -S -L libxs''.
```

The simulator may then be invoked by the command

```
xs mycircuitname
```

After the simulator has been executed, the command:

```
sigview graph.rnl.
```

may be used to view the results of the simulation.

2.3 Primitives

Figure 2 and *Figure 3* are the libraries which contain all XILINX XNF primitives supported by *xnfwirec*. XS supports all of those primitives except the following:

- IOB
- OSC
- tags (Flag-S, Flag-C, Flag-X, Flag-N, Flag-L)
- Pads (IPAD, OPAD, BPAD are ignored)

XS - XILINX 2000/3000 FPGA Simulator

Jason Zien, Jackson Kong, Pak K. Chan, Martine Schlag

October 17, 1991

1 Introduction

With the growing complexity of field programmable gate arrays (FPGA), there is the growing need for sophisticated design tools to provide higher level abstractions for managing large designs. However, it is not enough to be able to create large designs. It is also necessary to test and debug them. Debugging FPGA, designs on the circuit board is an awkward task, since the designer can only access the input/output pins of the chip. XS (pronounced as "excess") provides the designer with the ability to simulate and debug circuit designs quickly, and with access to all internal nets. XS is a unit-delay, event-driven simulator written in gnu g++ v.1.39. It was designed with an object-oriented methodology, and should be easily adaptable and extensible to any discrete-time simulations.

XS works in conjunction with several other programs to provide an environment for developing FPGA circuits. *XDP* is an interactive schematic capture program created by Carl Ebeling which supports hierarchical objects, and recursive and repetitive object definitions. *xnfwirec* is a dependency-checking program which compiles an *XDP* .dp drawing file into a XILINX Netlist File (XNF). *xnfwirec* first converts the drawing into a *wirec* .wc file with the program *dp2wc*. Next, the conversion program *wc2cc* converts the .wc file into a C++ .cc file. Finally, this file is linked with a library that will allow the resulting compiled C++ program to output the .xnf file. The exact same process is used for generating a simulation output file *graph.rnl*, except that a different library is used in the final linking step. The output of the simulator, *graph.rnl*, is a complete simulation run which can be viewed with the program *sigview*.

Section 2 describes all of the available commands and features of the simulator. *Section 3* is series of examples demonstrating the use of the simulator. Some runtime performance figures are given in Section 4. The specific implementation details of the simulator are described in *Section 5*.

2 Reference

2.1 Overview

Figure 1 shows the process in which a schematic is converted into an executable simulation. The entire process is encapsulated by a dependency-checking program named *wirec* [1]. By linking with the XS simulation library *libxs* instead of the XNF (Xilinx Netlist File Format) library *lcalib*, simulation code is produced. The simulator reads command input from a file named *test.script*, which controls the execution of the simulation. An optional file, *.xsinit*, can change various default simulation parameters (such as the file to direct output to). When a simulation is run, the results are placed in a file named *graph.rnl*. The program *sigview* may be used to view the results of the simulation.

[1] Refer to the *Xnf-Wirec Tutorial* for more information