CHAPTER 1 CSL C/C++ Simulator

TABLE 1.1 Chapter Overview

1.1 Definitions
1.2 CSL C/C++ Simulator Overview
1.3 CSL C/C++ Simulator Concepts
1.4 CSL C/C++ Simulator Commands summary
1.6 CSL C/C++ Simulator Commands
1.7 CSL C/C++ Simulator Examples
1.8 CSL C/C++ Simulator Checker

1.1 Definitions

1.2 CSL C/C++ Simulator Overview

The job of a C/C++ simulator in a chip design project is to provide an alternate model which generates stimulus and expect vectors and architectural state which is used to verify the RTL model. The vectors and state are written to files which are then loaded into memories in the RTL testbench. The stimulus memory(s) are then used to "drive" the design under test and the expected memory(s) are used to verify that the design under test is functioning correctly. The problem that the verification (tests), C++ simulation, and RTL design teams face is ensuring that the stimulus and expected vectors/architectural state are consistent. By consistent we mean that the different models and tests use the same signals in the same order in the vectors, that the architectural state is the correct width, that the transactions are in the same order, that the address spaces are the same,

The C++ simulator is a cycle based simulator. The simulator uses registers to capture the state of the simulation at the end of each clock cycle. More than one clock may be used in the simulation cycle. Each register is registered in a clock domain vector coorsponding tothe clock domain that the flip flop belongs to. the simulation cycle consists of the following steps: MAKE THIS INTO A TABLE
initialization: initialize the simulator
reset: issue a reset to the synchronous devices in the simulation

propagate: propogate the values from the element zero to element one in the registers
generate: evaluate the combinational logic in each unit and write the element zero elements in the registers
control: out of band control operations (command shell interrupts from the user, print out debug values,program termination,

The clock edges are generated from a master clock. Each clock contains a counter which is incremented each time the master clock isssues a "tick". Each clock is defined in terms of a number of "ticks" which is called the clock cycle count. When the clock's counter has reached the clock cycle count a rising clock edge is triggered and the counter is reset to 0. The simulator can be configured to simulate falling edge devices. If the simulation contains falling edge devices then each clock is 2x the clock cycle count. When the clock counter reaches clock cycle count devided by two then the falling clock edge is triggered. The rising and falling clock edge are used to control registers which capture the simulation state. When either the rising or the falling clock edges are triggered then the simulator will propogate the values from the element zero to the element one elements in all of the registers in the clock domain.

The CSL C++ simulator libraries define the cycle based simulation infrastructure. The libraries contain a set of classes which define the basic elements of the simulator. The elements in the library coorespond to the elements in the generated RTL. For example, a C++ unit class is defined in the library. It contains the virtual propogate and generate functions. The propogate function is overridden in the inherited class and contains the functional logic for the unit. The functional logic in the propogate function writes the values to the input of each register in the unit.

Each unit registers itself with the simulator's unit vector. The unit vector is used by the simulator to evaluate (the generate phase) the units after each clock cycle. Units are only evaluated when a clock associated with the unit has issued a rising or falling edge and the propogate phase has taken place. The generate function transfers the values from element zero to the element one in each register in the unit.

The generate phase controls the units and the propogate phase controls the flip flops.
CSL uses a high level specfication language to generate the C++ simulator inheritance hierarchy

Generate verilog interconnect using C++
Cycle based C++ structural RTL simulation Limit
Find cycles in combinational code
Find races - where a variable is written independently in two different blocks.
Find conditions where a signal does **not** have a default
prior to a conditional & the var is not written in all conditional branches.

Verify in SM's using case/if-ifelse that the variables that are not assigned in all branches are assigned a default value prior to the conditional.

Verify that the inputs to execute blocks.
pull values directly from reg's, rf's, mem's (state elements - SE) (or use a function to pull from an SE thru a logic cone.

1.3 CSL C/C++ Simulator Concepts

Automatically ordering the C++ simulator signals to match the signal order of the vector/architectural writer, vector/architectural state file, testbench memory, design under test interface, and testbench comparator.

The C++ simultor is cycle based. The C++ simultor contains a simulation engine which ahs several phases.

```
    unit_name.generate_cpp_unit(cpp_unit_name);
```

C++ class cpp_unit_name is the name of the cpp unit which cooresponds to the RTL unit named unit_name. cpp_unit_name is a C++ class which is derived from the C++ class unit which is a C++ simulator library class.C++ class cpp_unit_name contains private/protected variables for each interface signal. The C++ class cpp_unit_name contains public functions to set and get the value of each interface signal. The CSL verification components can add vector and architectural state writers to the C++ class cpp_unit_name which are connnected to the C++ class cpp_unit_name's interface signals. The writers capture the values of the C++ class cpp_unit_name's interface signals and write them out to a file. The file can then be loaded into the testbench.

In addition other CSL objects can add objects to the C++ class cpp_unit_name. Examples include registers, register files, fifos,... Moreover, the address space of the design is specified in CSL and is converted into C++ and Verilog code. The generated C++ address constants are used to create the correct address structure in each C++ unit.

The generated C++ class unit_name inherits C++ class unit which is in the CSL C++ simulator library. The C++ class unit declares virtual functions which include propogate and generate. These functions are defined by the user in the class unit_name_user. The simulation team writes a C++ class unit_name_user which defines the virtual functions in the C++ class unit.

Additional functions such as get_unit are defined in the C++ class unit. get_unit() returns the reference to the unit and is used by the command shell debug interface to access the individual objects in the C++ class unit, unit_name, and unit_name_user.All other CSL chapters which contain components which have C++ equivalents generate C++ code which is added to the C++ class unit.

TABLE 1.2

	Description
class unit	defined in the C++ simulator library
class unit_name	generated by the csl from the CSL unit specfication
class unit	written by the user

The automatic generation of the unit_name class which contains the
interface variables and the vector writer classes automatically ties
the C++ simulator unit interface to the testbench memory which is
connected to the design under test or the testbench DUT output comparator.
The DUT stimulus vectors are used to "drive" the DUT and the DUT
expected vectors are comapred against the DUT outputs.

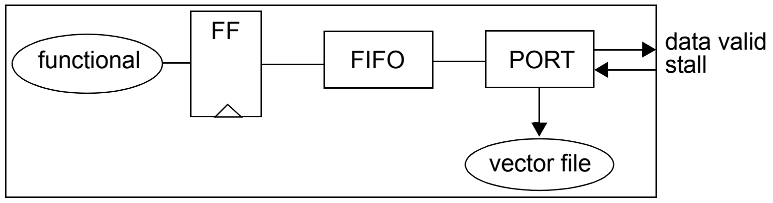
CSIM
Address object for C++ simulators
Addresses can be byte aligned, double byte aligned, 32-bit word aligned, 64-bit word aligned etc.

Constants and variables which are used to hold addresses values are aligned to 8, 16, 32 or 64 bits. Variables which do not have address alignment information are difficult to work with. A class which contains the address and the different methods to extract the different aligned values for the addresses reduces the amount of debugging by the programmer.

1.3.1 Addresses

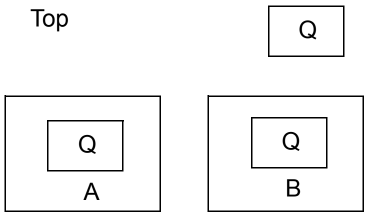
Typical C++ Simulator module

FIGURE 1.1 C++ Simulator module interface logic



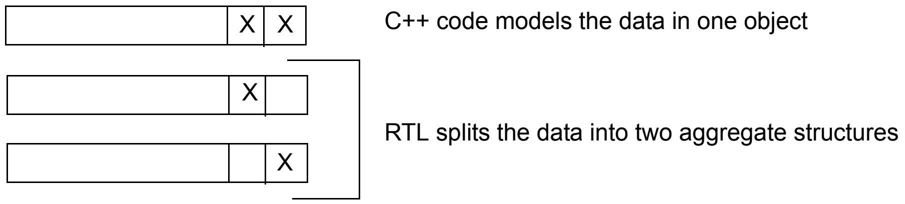
Address scheme for addressing UNITs on any level of hierarchy
Q=QSPU = querry status performance unit
AU= address unit

FIGURE 1.2 Querry status performance uni



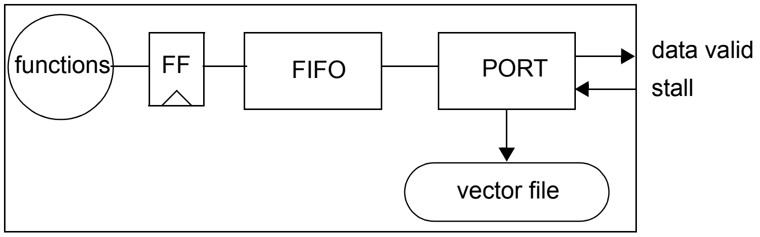
1.3.2 Csim/RTL modeling

FIGURE 1.3 Csim/RTL modeling difference


X X
C++ code models the data in one object
X
X
RTL splits the data into two aggregate structures

1.3.3 CSim vector generation

FIGURE 1.4 Vector generation



1.3.4 Dynamic Objects

Dynamically load shared objects

The Csim contains a list of module names.
A function will call dl_open() on each module name.
A path will be searched.
If the <mn>.so is not found then the dl_error() message will be printed out.

1.3.5 Command line arguments

The csim - vm </m> argument tells the csim to create and connect PLI's for each of the Verilog DUT's interfaces.

sim_cntl_gen master_cntl
ls, cd, pwd   break
simulation control commands
start, stop, v(??), step, cont, load_mem, read_mem, load_mem_wd, read_mem_wd, load_reg, rd_reg, get_obj<addr> or <"obj_name">, dump_obj, compare_obj_state <state_obj>

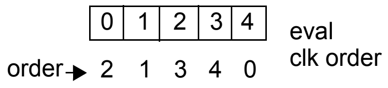
<file_obj;>[record# infile]

1.3.6 CSim Eval

1.3.6.1 Csim Eval overview

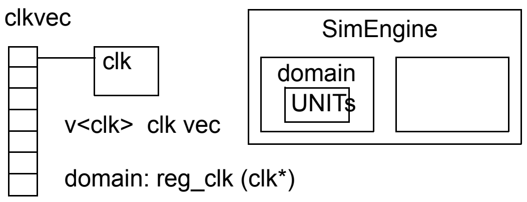
The order of eval for the clks should /\ (try?) every sim cycle based on the current relationship of the clock edge with respect each other every cycle so the propDQ needs a to know order of clkc to eval every sim cycle. the order of clk eval may /\ .There should be a global clk eval array which determines the order to eval the clks in a state element propDQ

FIGURE 1.5 global clk order list for sim cycle



some clks may clk a device more than once during a sim cycle - not allowed
sim cycle should be based on the fastest clk!

FIGURE 1.6 Sim clock

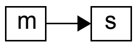


each class clk has (???) clk edge

```
ClkEdgenext clk;
vector <obj*>; // pointers to all objects that this clks connected to
               // use to call propDQ
```

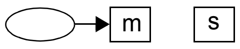
Sim cntl
Evaluation order
1) calc clks
2) prop state elements
   element zero -> element one

FIGURE 1.7 State elements



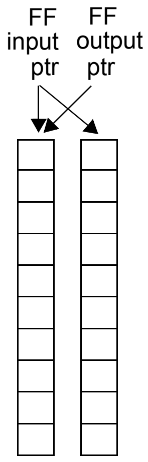
3) calc combinational logic cones

FIGURE 1.8 Combinational logic cones



4) PLI’s after 2s
5) vectors dump after 2s

FIGURE 1.9 List of flip-flop input and output connections



Fast simulation state element
element zero element one update
ping pong between the two arrays to do the state update
for one clock domain
put all state variables in an array
each FF contains an index into the array
Don’t store the state in the FF class. Store the state in a clock domain array

FIGURE 1.10 Global Reg list

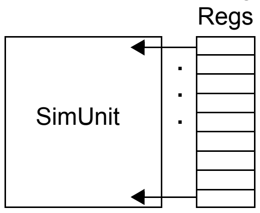
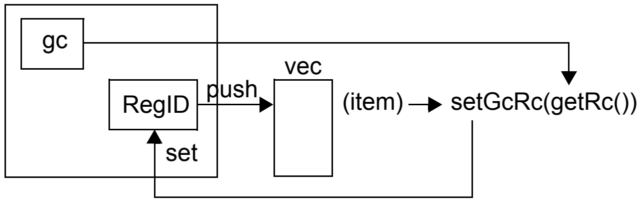


FIGURE 1.11 Vector output



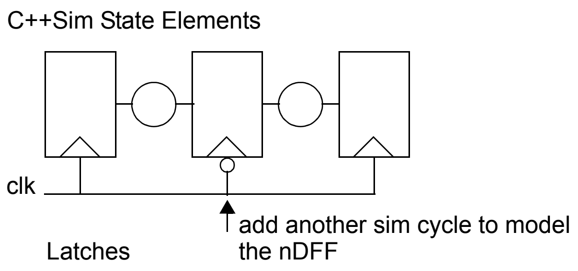
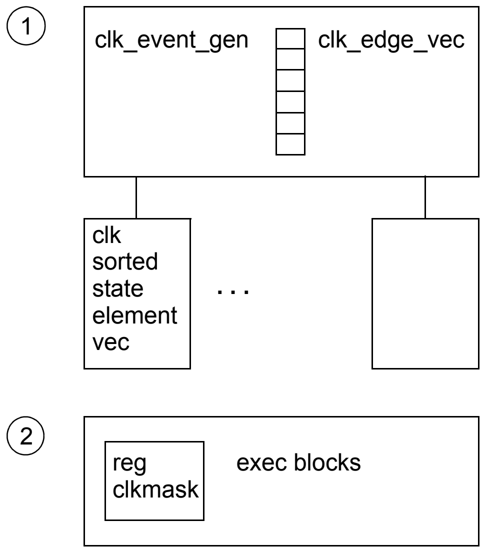


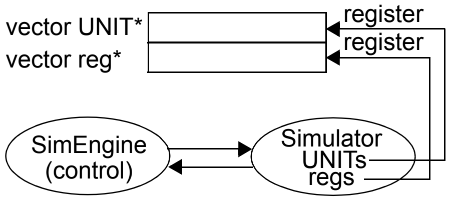
FIGURE 1.12

FIGURE 1.13



the clk mask in the reg's determines which sorted state element lists the " are added to(??)

FIGURE 1.14 Array



1.3.6.2 Sim evaluation order

1.inst (first cycle) or sim_cntl(after 1st cycle)

2.clk eval

3.state elements

4.combinational logic

5.communication with socket or files

6.command shell

7.scheduler can reorder combination operations in the C++ sim so that combinational fan in cones are executed correctly

8. component loops

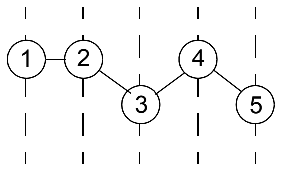
9. combinational loops

1.3.6.3 Clock generation

-There is a clock generator which has the following properties
-clock edge events are calculated using the following formula:

```
if(simcycle) {
  <specific_clk_sum> = <fastest_clk_period> + <specific_clk_sum>;
    <specific_clk_edge_event> = <specific_clk_sum> >
<specific_clk_period>; // clk_sum greater than clk_period
    if (<specific_clk_edge_event>) { <specific_clk_sum> =
<specific_clk_sum> % <specific_clk_period>;
}
```

FIGURE 1.15 next clk edge Determining when to eval



-currenttime newtime=0

```
compute all checksums
foreach clk (clklist) {
  t_clk = clk.clk_period + clk.clk_sum
  if(clk > current_time){
  if(t_clk>newtime){
    newtime=t_clk;
    newclk=clk
    }
  }
}
currenttime = newtime
eval
  static LLU currenttime
  alias ct

class clk {
  clk_period
  clk_sum
}

if
```

1.3.6.4 Register evaluation

-State elements have virtual propDQ function which uses the list state elements list of clocks and global enables to enable the data[0] to data[1] propagation
-skew relative to other clocks may be introduced using a flag to the simulator
-jitter can be introduced to the simulation
This would mean that the simulation would have to have a simulation cycle based on the fastest clk*2 so that the <clk_specific_sums> could have their sums modified (move the edge back or forwards in time)
-global signals such as reset are also enables for the state elements

1.3.6.5 Master controller

need static properties which check if there is more than one value in the cycle being written to the data[1]
How do you guarantee that two ore more clocks won't collide with each other (both be on at the same time) which could cause different values in the state element to overwrite the data[1].

-There is a master controller which executes the following algorithm to simulate the design

```
init(); //setup the clk frequencies
sim_cycle=get_fastest_clk();
while(not_done){
  calc_clk();
  foreach f (UNIT_list) { f->execute();
  foreach s (state_element_list) { s->propDQ();
}
```

1.3.6.6 File operation

Sim Cntl has options to load/dump the mems to/from files:

- hex (ascii)
- binary

File Format
  header
  alias
  initial values
  /\ values
  footer

1.3.6.7 UNIT

All UNIT's are registered in a global array
-UNIT has a global execute function
- interfaces may be decoupled from the state elements using one of the following mechanisms

- using a FIFO/QUEUE between two units to capture all events coming from one unit (works even if the downstream unit is blocked)
- using interprocessor send and receive instructions which wait for a valid bit effectively blocking the execution of the unit executing the instruction

-During executeUNIT's reading the output of another UNIT are allowed to read the following from another UNIT

```
<state_element>.read()
<UNIT>.<func>()
```

where <func>() is a function which follow the following rule
-race conditions occur if the output of a block is combinational and only the output variable is read. instead a function should be called in the upstream UNIT by the downstream UNIT to evaluate the fain in the upstream UNIT back to the state elements.
-cycles are not allowed in combinational logic
-Each state element has a list of pointers to global clocks and enables (reset)
-State elements may be enabled with local signals
-There is a hierarchy of state elements which is used to implement the virtual functions.
Find next clk edge
drives sim cycle

1.3.6.7.1 Unit access

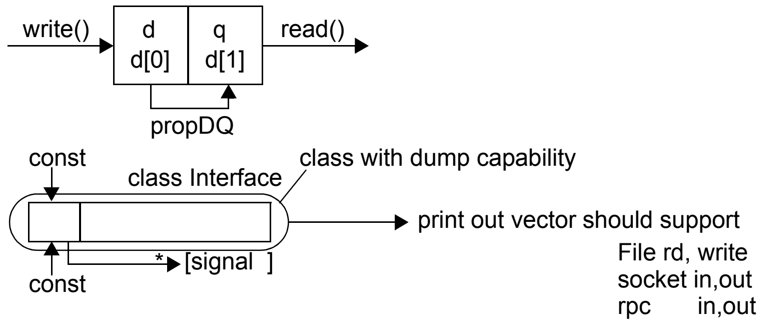
Interface to C++ simulator/ verilog
access symbols in C++ or v.sim using $<hierarchical_path>_symbolname OR locally $<symbol_name>
Access objects different ways
assign unit numbers to each functional unit block (UNIT)

1.3.6.7.2 Unit IDs

assign names to each UNIT vector<“str“,object pointer>

1.3.7 CSim interface design

FIGURE 1.16 Interconnect between modules using vectors & constants by both modules



C++ Simulation Interface Design

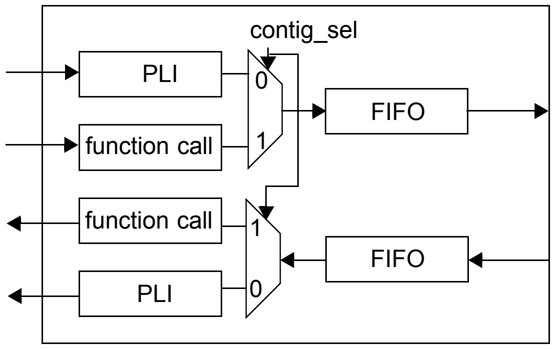
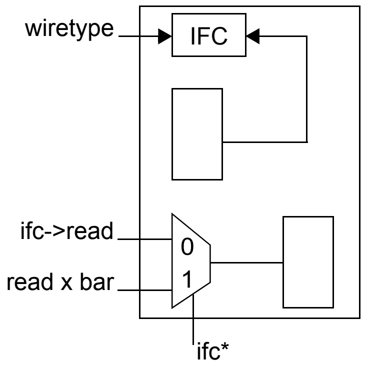


TABLE 1.3 Verilog & Csim

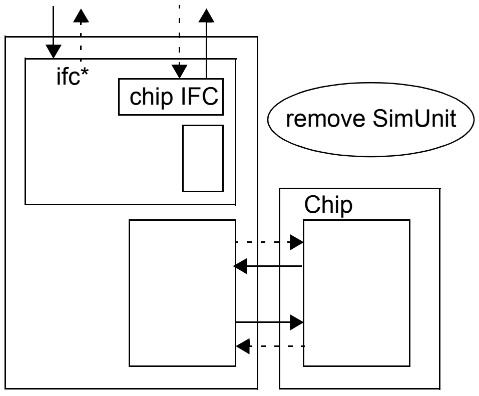
generating	Verilog interface	verilog registers / memories
Maintaining	Csim interface	Csim registers/ memories
	TB instantiations	Driver register/memories

FIGURE 1.17 Interface



always listens or listen, if instantiated

FIGURE 1.18 Host Bus Model



1.3.8 CSim Framework

- cycle based C++ simulation
- event driven simulation

A core checker is needed to verify if the C++ logic is cycle based. This contains a loop checker and uses totpological sort.
//Describe C++ API and sim mechanism

1.3.9 Debug

C simulation
The C simulator registers and state elements should have a print function to print out their state. There will be a mechanism to arrange the print out of the elements.

We will instrument the C pipelines in order to facilitate the debug of the machine.
The instrumentation will print out debug messages as defined by the programmer.
The debug messages will be conditioned by an "if(prn_msg.level())".
prn_msg is an object which is instantiated in each class. It corresponds to a command line argument with the name <module_name>_prn_msg. The object <module_name>_prn_msg will register itself with the simulation command line argument control object which will parse the command line and look for the strings which match the <module_name>_prn_msg variables. when the command line contains a <module_name>_prn_msg=<level> the command line control object will set the level in the <module_name>_prn_msg object with a call to the <module_name>_prn_msg->level(<level>) function.
To reiterate, each module contains an object called <module_name>_prn_msg. The <module_name>_prn_msg object registers itself with the command line control object. The string in the command line control object is

The messages will print out when the command line flag is passed into the program which controls the debug message.

Methodology
C++ and Verilog equivalence checker
compare interfaces
compare clk tunes
compare lib cells
compare clk wave forms
compare reset logic
compare reset wave forms
compare state elements (SE)
compare SE fanin cones bidirectional logic

1.3.10 Memory map

The memory map for each module should be converted into a diagnostic test to write and read each register and compare the value written against the value read.

The offset method of writing the registers generates code which compares the address offset argument to each of the addresses in the memory space. If there is a match then the data is either written to the memory location or the data is read from the memory location.

In the case of memories which are addressed and written using index(offset)/data pairs a write to the data register triggers a write to the memory in the next cycle.

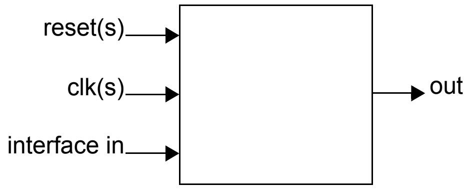
Registers are assumed to be 32 bits.
We could have a return type of UINT* where UINT * points to an array of UINT's which hold the data for a register.
If there are variable sized registers in the memory space then the reader needs to know the size of the register.
They need to call the <memory_element_name>.size() function to determine the size of the memory element which is measured in 32-bit words.

The same operation can be done on the write side.

csim
Address object for C++ simulators
Addresses can be byte aligned
double byte aligned
32-bit word aligned
64-bit word aligned
constants and variables which are used to hold address values which are aligned to 8,16,32 or 64 bits address variables which do not have address alignment information are different methods to extract the different aligned values for the addresses values the amount of debugging by the programmer

1.3.11 CSim clock

FIGURE 1.19 Testbench



The C++ class needs to declare the module interfaces, resets, and the clks. The clks and resets are connected to a testbench driver or a global or local clock module.

1.3.11.1 Clock generation

FIGURE 1.20 Clock phase

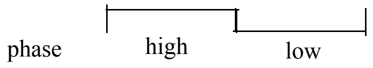


FIGURE 1.21 Clock signals

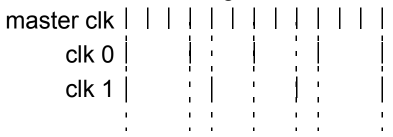
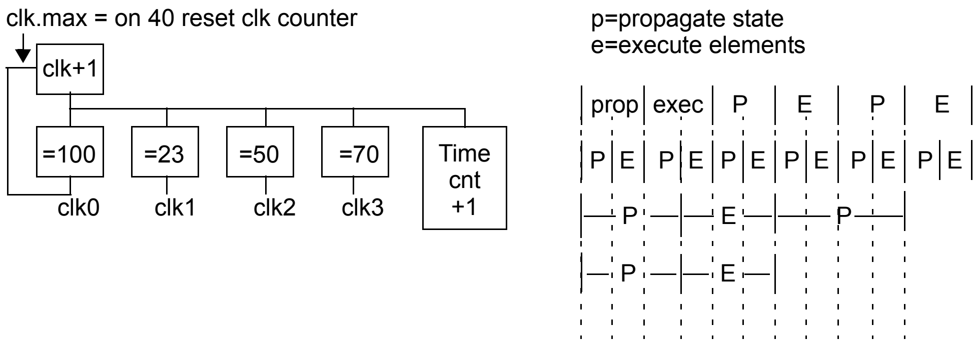
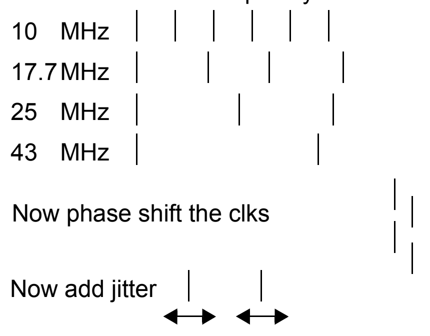


FIGURE 1.22 Clock generator



```
cnt = 0;
nextclk = 0;
nextedge = clk_vec[0].nextedge;
for(i=1; i<clk_vec.size();i++)  {
  if(clk_vec[i].nextedge < nextedge)
    next_clk = cnt;
    cnt++;
  }
return next_clk;
}
```

FIGURE 1.23 Clock frequency

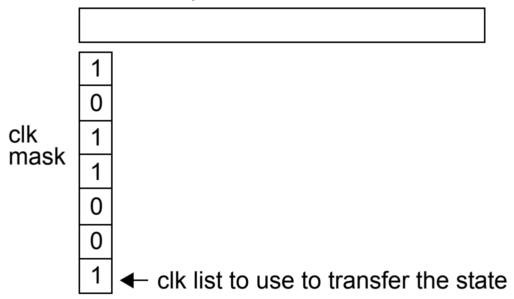


1.3.11.2 Clock synchronization

cycle based c++ analysis tool
combinational loop check
component loop check
crossing clk domains
check for syncronisation between clk domains FF_{clk0} to FF_{clk1} to FF_{clk1}
need to check for clk edge violations

1.3.11.3 Clock Domain

FIGURE 1.24 Multiple clk domains



```
Reg
  clkmask clkm;


Reg(...,clkm);
```

C++ parser should analyse the reg to reg connections and print warnings if there is a clk domain crossing
add a wrapper function clkDomainCross(Reg* WrReg, Reg* RdReg)

1.3.11.4 Gated Clocks

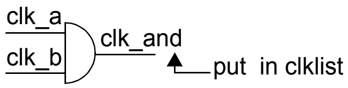


FIGURE 1.25 Anded clock

1.3.12 Waves

/test

```
vcd_enable(){
  foreach f(fv){
    f.vdc_enable();
  }
  foreach c(child UNITvec) {
    c.vcd_enable();
  }
}
```

VCD

Every class needs a VCD dump routine

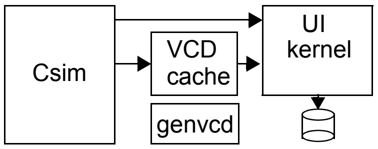
```
Object
  virtual void dumpVCD() {}
```

Each level needs instname

```
$var reg 32      *x      net32
       /|\             /|\
        |               |
  type width    alias   verilog name
```

#500 calctime() {return 2;}

FIGURE 1.26 VCD



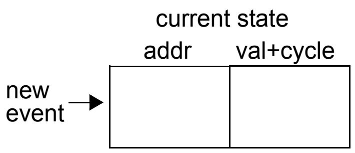
```
propDQ(d)
  if(d[i]!=d) //new event
    vcd_cache.push_back(d,t+1,cycle,time)

    write VCDevent(Reg*, newVal)
```

```
        vcd_cache.push(VCDevent)

        wire operator=


    while() {
      clk
      prop
      exec
      process vcd_cache
    }
```

VCD_DB
tree \
      | of all vcd nodes
list   /

VCDEevent: name, address, cycle, val

FIGURE 1.27



```
    if(cs[newevent.addr]->val != newevent.val)
       cs[newevent.addr]->val == newevent.val
    & generate VCDEevnt(newevent.addr)
```

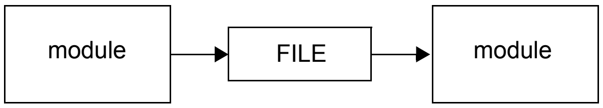
1.3.13 CSim Init

CSIM initialization
initialize memory and variables
process id (PIN)
var dom (pid?)

1.3.14 CSim Levels

must be able to run the Csim at the system, chip, cluster, module levels.
Host bus randomization

FIGURE 1.28 Data flow bus



1.3.15 CSim socket

1.3.16 Module connection

Csim interconnect between module: FIFO

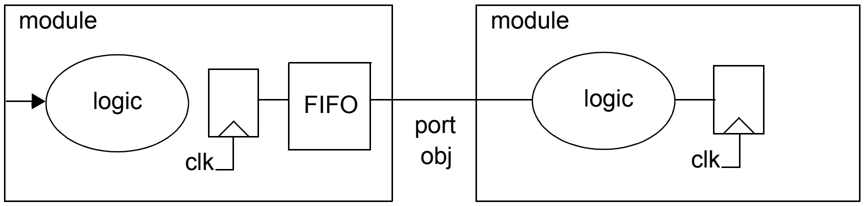
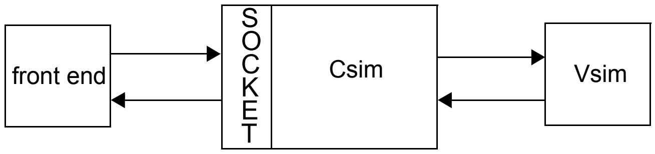


FIGURE 1.29 Socket connection



CSIM module

FIGURE 1.30 clock/reset

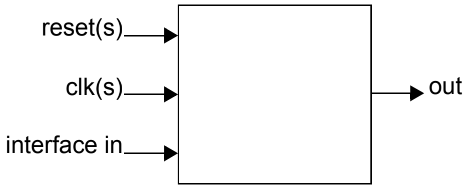


FIGURE 1.31 Socket connection

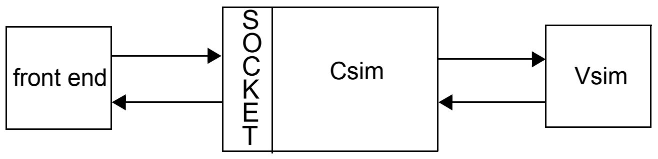


FIGURE 1.32

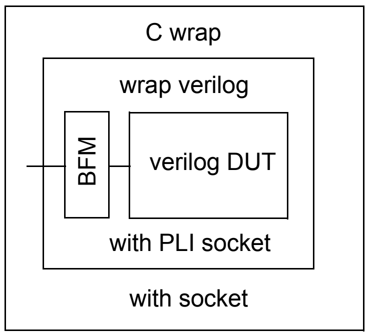


FIGURE 1.33 Peer to peer connection

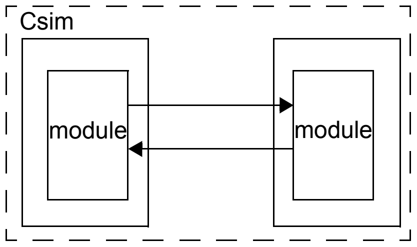
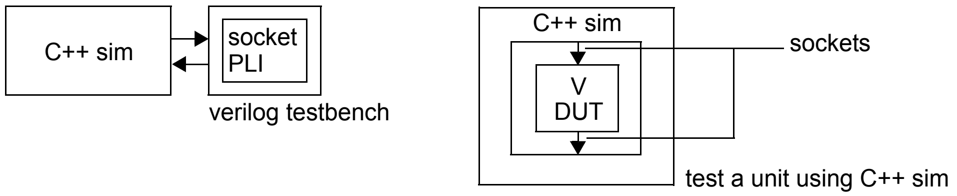


FIGURE 1.34 Verilog PLI interface in testbench communicates directly with C++ simulator using sockets



1.3.17 Csim register

-Each clock is an enable for state element propagation
-The fastest clock is the simulation cycle trigger for each state element

-Register all state elements in a global array
-Iterate through global array and eval each enable for each state element
-each state element has a element zero and a element one data element (data[2])
 data[0] is element zero
 data[1] is element one
 The element zero is written with a write() function.
 The element one is read with a read() function
-There are blocks called functional unit blocks (UNIT) which contain UNIT's, combinational logic and other state elements
-The simulation engine consists of propDQ() and execute() virtual functions

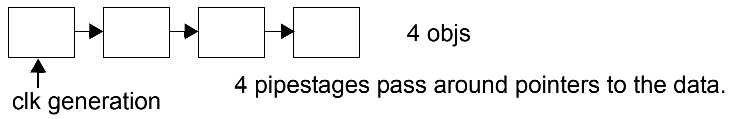
1.3.17.1 CSim register proprieties

-propDQ() is in effect a non-blocking assignment
-propDQ() checks each clk and enable in the glocbal list and propagates the values if the one or more of the enables is on.
-need to check for logic collisions if more than one clock is on

1.3.18 Pipeline

```
SIGNAL HANDLER
EXECUTION LOOP
//syncronisation barrier
while(1) {
   //USAGE process
   interrupt_clock(signal handler)
   clk_calc; // one phase per loop
   state element
   - propagate
   -generate
   port process;
}
```

FIGURE 1.35 interconnect objs



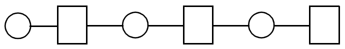
Three different types of pipelines are used in the C++ simulator to model digital pipelines.

- cycle accurate
- transaction accurate
- latency accurate

1.3.18.1 Cycle Accurate Pipeline

A cycle accurate pipestage models the state of the pipeline at the outputs of each pipeline's state elements. The exact latency through the RTL design pipeline is modeled and the logic operations are performed in the same pipestage in the C++ simulator as in the RTL design.

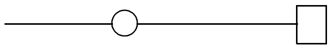
FIGURE 1.36 Cycle accurate pipeline



1.3.18.2 Transaction Accurate Pipeline

A transaction accurate pipeline models the state of the pipeline at the module's output state elements. The latency through the RTL design pipeline is not modeled. However, the logic operations are not necessarily performed in the same pipestage in the C++ simulator as in the RTL design.

FIGURE 1.37 Transaction accurate pipeline



1.3.18.3 Latency Accurate Pipeline

A latency accurate pipeline models the state of the pipeline at the state element outputs of the design. The exact latency through the pipeline is modeled. However, the logic operations can be performed in any pipestage in the C++ simulator.

FIGURE 1.38 Latency accurate pipeline

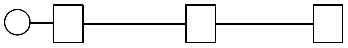
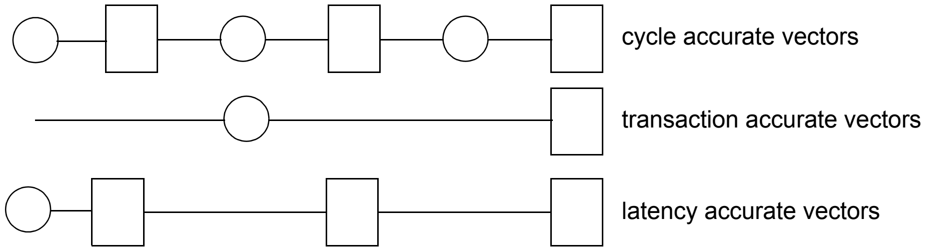


FIGURE 1.39 Three vector generation models

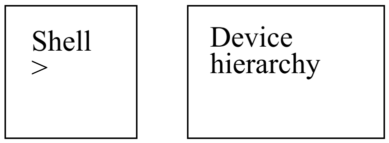


1.4 CSL C/C++ Simulator Commands summary

pwd

ls
cd <dir>
load_mem <file> <mem>
run
run_to
break <time>
break line
stop
start
clock commands

FIGURE 1.40



1.5 dump_waves -format *file name*

```
csl_sim sim_name;
sim_name.compare_level(unit_name);
sim_name.vector(vector_name);
  sim_name.add_valid_bit();
  sim_name.generate_header_comment(header_options);
```

1.6 CSL C/C++ Simulator Commands

`csl_sim` *`sim_name`*`;`

DESCRIPTION :

EXAMPLE :

CSL CODE

```
   //csl code goes here
```

sim_name.**compare_level**(*unit_name*);

DESCRIPTION :

EXAMPLE :

CSL CODE

```
   //csl code goes here
```

sim_name.**vector**(*vector_name*);

DESCRIPTION :

EXAMPLE :

CSL CODE

```
   //csl code goes here
```

sim_name.**add_valid_bit();**

DESCRIPTION :

CSim adds a valid bit to each line of the generated arch_stateor vector file to mark the last arch_state when the last arch state vector set was read. When the valid bit equals 1 then the last vector is detected.
Csim does this (adds a bit) when generating the archstate file

EXAMPLE :

FIGURE 1.41 CSim

```
valid/vector
0
0
0
.
.
.

1 ( last arch_state )
```

CSL CODE

```
//CL
csl_testbench tb;
csl_csim sim_name;
sim_name.add_valid_bit();
csl_arch_state as;
as.set_expected_filename(“expected.as”);
as.set_single_line();
as.set_dut_mem(arch_state_memory);
as.set_dut_mem_init(0);
as.set_dut_arch_state_radix(bin);
tb.add_arch_state_instance(as, as0);
csl_unit unit_a;
tb.add_dut_instance(unit_a, DUT);
```

NO VERILOG

sim_name.**generate_header_comment(***header_options***);**

DESCRIPTION :

Add options to control what the comment header looks like. When the CSim generates the arch_state file it adds comments in the header. Options are:

TABLE 1.4 Header Comment Options

Option	Description
ASName	arch state name
SimVer	simulator version used to generate arch states
SimArgs	simulator arguments used
Date	date of the simulation
Time	time of the simulation
User	user
MachineArch	machine architecture

EXAMPLE :

CSL CODE

```
//CL
csl_arch_state as;
csl_csim sim_name;
as.set_connection_type(file);
as.set_expected_filename(“expected.as“);
sim_name.generate_header_comment(ASName, SimArgs, Date, Time);
```

VERILOG CODE

```
//N=no. of arch_states
reg[WIDTH-1:0]as[0:DEPTH-1];
reg[WIDTH-1:0]as_mem[0:N*DEPTH-1];
initial begin
   $readmemb(“expected.as“,as_mem);
   as=as_mem[0];
end
```

1.7 CSL C/C++ Simulator Examples

1.8 CSL C/C++ Simulator Checker

1.8.0.1 stim/exp vec reader/writers/clocks

1.8.0.2 mem state vec reader/writers/clocks

1.8.0.3 reg state reader/writers/clocks

SG
C++ sim_gen
-builds the class hierarchy using the hiearchyr file or extracted hierarchy
-classes inherit from a base class which contains the variables & functions to support sim_cntl.
sim cntl kernel
bash or tsch history
                    edit commands
support scripting at command line or reading in scripts
The sim snt kernel is embedded in the C++ simulator

SG
C++ support classes
-state elem lib cells (register themslves with clk domain SE vec)
-FF (different versions)
-combo elem (register themselves with combodomain c_vec)
-clk tree elem lib cells/classes register with clk tree vector
clk mux, and, or, latch_en
build clk trees with clk tree lib cells
- clk domain SE vector
   clk domain crossing elements
        syncronizers
        queues

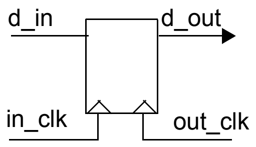
SG
Overviewcsl_sim spec
- builds the hierarchy
- sim cntl for C++ or Verilog
- load test
- checkpoint/restore state using vecs or pli
-random init of nodes using seed (repeatable) ()through PLI or vecs with verilog)
-supports multiple clks

Sim Gcen
Threading the sim

```
   class  // use to propDQ regs
```

```
      clk  // set up clk domains
    Vec;(?)
    RegVec regvec; // list of regs in clk domain that use clk

    regreg(Reg* r) { regvec.pushback(r); }
    class UNIT // use to enable vcd dump
```
[...]

FIGURE 1.42 Asynchronious unit



<MOVED FROM="CSL BUSES">

Modeling connections between units in a Csim sometimes requires abstracting the modules to that input buffered connection. They do not handshake in the same way as the hardware. This requires inserting FIFOs or queues between the modules to absorb the upstream data that cannot be processed by the downstream unit as fast as the data is produced. The intro of FIFOs which have no counterpart in the Csim does not affect the transaction modeling accuracy of the Csim.

FIGURE 1.43 Csim Producer Consumer

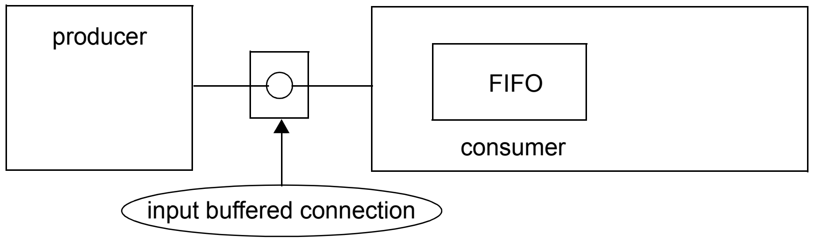
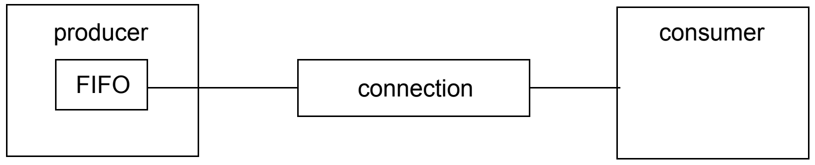


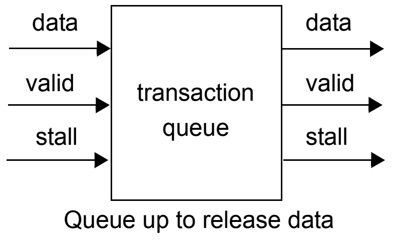
FIGURE 1.44 Producer Consumer Bus



output buffered

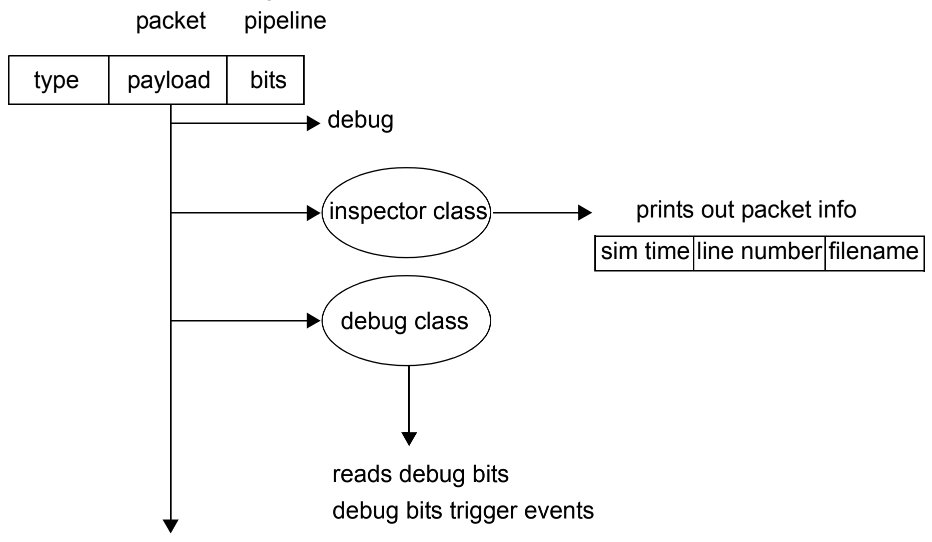
Transactions can be gathered and applied in burst mode or can be applied to the DUT randomly. The same mode controls can be applied to the consumer status.

FIGURE 1.45 C++ component



1.8.1 Packet debug mechanisms

FIGURE 1.46 Packet debug mechanisms



pag18

<MOVED>