## CHAPTER 1



verilog TBgen

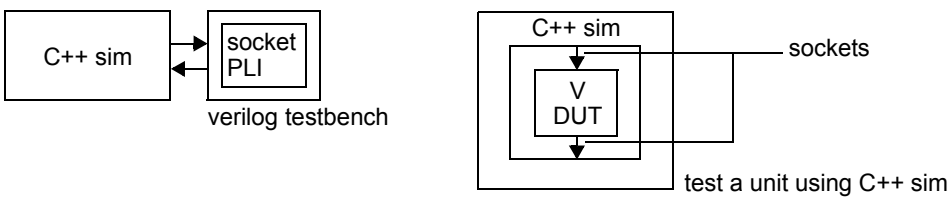
C++ SimGen

stim/exp vec reader/writers/clocks
mem state vec reader/writers/clocks
reg state reader/writers/clocks

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]

Perl 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 #s to each functional unit block (FUB)
assign names to each FUB vector<“str“,object pointer>

**FIGURE 1.1** Verilog PLI interface in tesbench communicates directly with C++ simulator using sockets



SG
C++ sim_gen
-builds the class hierarchy using the hier file or extracted hierarchy
-classes inherit from a base class which contains the variables & functions to support sim_cntl.

```
class sim_cntl_hier base : public sim_cntl
  private:
    UINT unit_id;
    name template name;
    name instance name;
    hierpath hp; // string containing hier_path to unit

  public:
    get/set {unit_id, template_name};
    typedef vector<mem*> mem_vec;
    typedef vector<reg*> reg_vec;
    global sym vars

  private:
    mem_vec m_v; // vector of memories
    reg_vec r_v;
    mem_hash mh; // vector<obj_id, mem*>
    reg_hash ;
    mem_hash ;
    unit vec ;
    unit hash ;
```

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

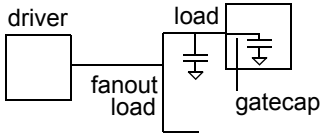
Simulation
inst (first cycle) or sim_cntl(after 1st cycle)
clk eval
state elements
combo logic
communication with outside

scheduler can reorder combination operations in the C++ sim so that combo fan in cones are executed correctly
can find
- component loops
- combinational loops

SG
Verilog code generation
structural simulation
structural gate level

The gate level view instantiates cells with different drive strengths.
there should be an algorithm in the verilog code gen section that is able to determine load and the required drive strength of the cells.

**FIGURE 1.2**

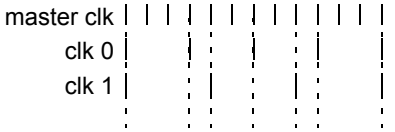



use logical Effort

$$\sum_{i=0}^{fanout} U_i$$

SG
Overview simgen
- 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

**FIGURE 1.3**



Sim Gcen

**FIGURE 1.4** interconnect objs

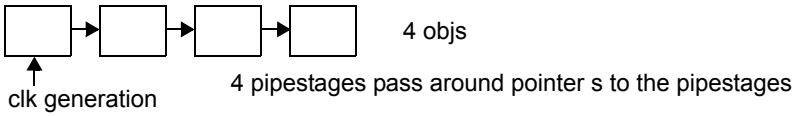
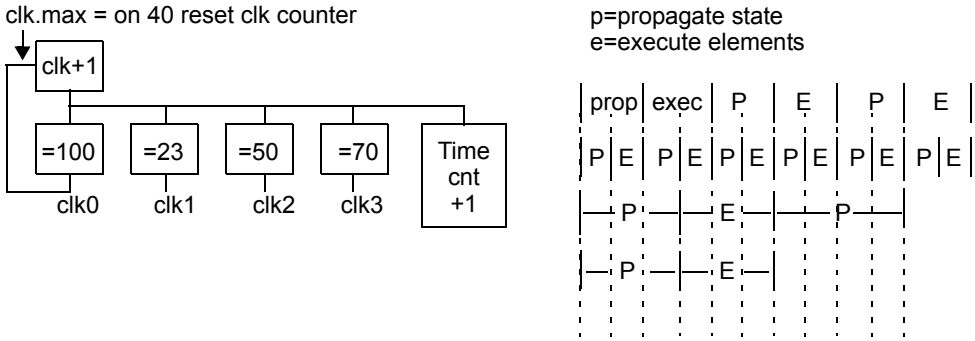


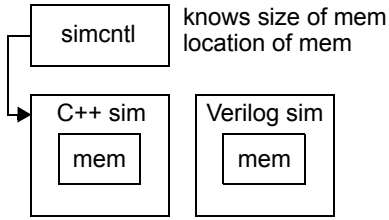

**FIGURE 1.5**




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
Threading the sim

SimGen

**FIGURE 1.6**

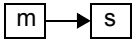



Sim Cntl has options to load/dump the mems to/from files:

- hex (ascii)
- binary

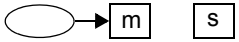
Sim cntl
Evaluation order
1) calc clks
2) prop state elements
master -> slave

**FIGURE 1.7**




3) calc combinational logic cones

**FIGURE 1.8**




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

**<this text is also found in 2006_02_26_csim2. Duplicate entry ?>**

**C++ Cycle Based Simulation With Self Registering Objects (FUB's and state elements) and Multiple Clocks**

**-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**
**-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>;
}
```

**-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] propagate**
**-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**
**-each state element has a master and a slave data element (data[2])**
**data[0] is the master**
**data[1] is the slave**
**The master is written with a write() function.**
**The slave is read with a read() function**
**-There are blocks called functional unit blocks (FUB) which contain FUB's, combinational logic and other state elements**
**-The simulation engine consists of propDQ() and execute() virtual functions**
**-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**

**o 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 (fub_list) { f->execute();
    foreach s (state_element_list) { s->propDQ();
}
```

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

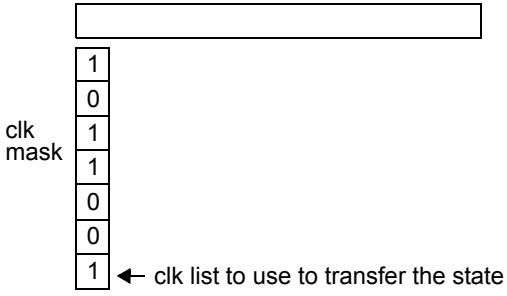
- using a FIFO/QUEUE between two units to capture all events coming from one unit (works even if the downstream unit is blocked)
- using interprocessor send and receive instructions which wait for a valid bit effectively blocking the execution of the unit executing the instruction

-During executeFUB's reading the output of another FUB are allowed to read the following from another FUB

```
<state_element>.read()
<FUB>.<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 FUB by the downstream FUB to evaluate the fain in the upstream FUB 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
</ end of duplicate entry ?>

**FIGURE 1.9** To eval multiple clk domains

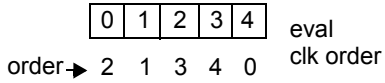



```
Reg
  clkmask clkm;

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)

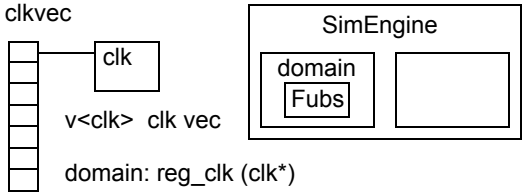
The order of eval for the clks should $\underline{\wedge}$ (try?) every sim cycle based on the current relationship of the clock edge with respect each other everycycle so the propDQ needs a to know order of clkc to eval every sim cycle. the order of clk eval may $\underline{\wedge}$ .There should be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

**FIGURE 1.10** 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!

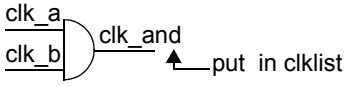
**FIGURE 1.11**




each class clk has (???) clk edge

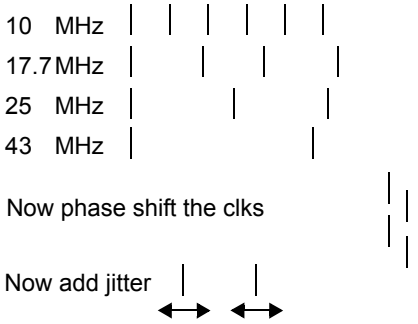
```
ClkEdgenext clk;
vector <obj*>; // pointers to all objects that this clks connected to
               // use to call propDQ
```

**FIGURE 1.12**




```
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 1.13**



```
    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 fub // use to enable vcd dump
```

[...]

/test

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

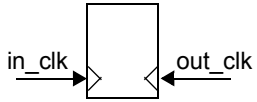
```
SimEngineInst::init() {
  vcd -> init();
}
SimEngineInst::terminate() { // call whenever sim exits
  vcd -> terminate();
}
```

File Format
header
alias
initial values
/\ values
footer

```
class vcd_event {
}
operator <<

class vcd_cache {

  private:
    ofstream vcd_file;
    vcd_event* vcd_ev_a[VCD_EVENT_ARRAY_SIZE];
    int cnt;

  public:
}
void spill_cache(void) {
  if (cnt == ) {
    for (i=0; i<=VCD_EVENT_ARRAY_SIZE; i++) {
      vcd_file << vcd_ev_a[i];
      delete vcd_ev_a[i];
      }
    cnt=0;
    }
}
void add_vcd_event(vcd_event* v_e) {
  vcd_event[cnt++]=v_e;
  spill_cache();
}
```

```
#define VCD_DUMP

vcd_dump::init -> new vcd_cache object
                  open vcdfile
<state element>::init() -> create vcd aliases (signal_name <->
sig_alias) i.e. "*$("
                           sig initial value
<state element>::propDQ() -> if(newvalue != d[i]) then
                            vcd_dump(new_value alias); // need to store
                                                       // alias in obj
only implement in the state element level which has NO enables
add alias, inst name/name, clk to state elements
add clk obj (name, cycle, period) to class hierarchy
<state element>::vcd_dump(new_val, clk_cycle)
             cycle = clk->get_cycle(); //only one clk per state element
                                       //what about a (??) clks
```

**FIGURE 1.14**




```
class clk
  get_cycle() //returns the absolute time reflected in the ps/ns
              //timestep
```

VCD
Every class needs a VCD dump routine

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

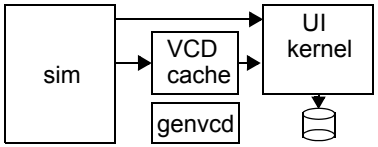
Every class needs a verilog type Register=reg
Need R(??) Inst name
Each level needs instname
$var reg 32 *x net32
/|\ /|\
| |
type width alias verilog name

IEEE Verilog Page 218

#500 calctime() {return 2;}

pag218 VCD

**FIGURE 1.15**

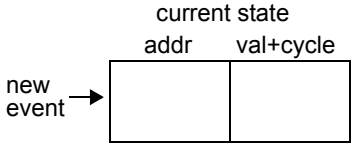



```
propDQ(d)
  if(d[i]!=d) //new event
    vcd_cache.push_back(d,t+1,cycle,time)

    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 1.16**




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