CHAPTER 1 chapter title

FIGURE 1.1 Peer to peer

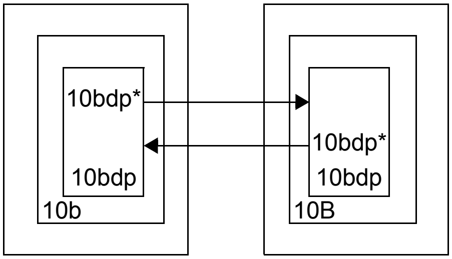
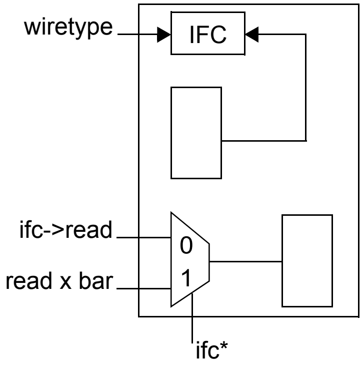


FIGURE 1.2 Caller ID phone (?), Bike (?)



always listens or listen, if instantia(?)

FIGURE 1.3

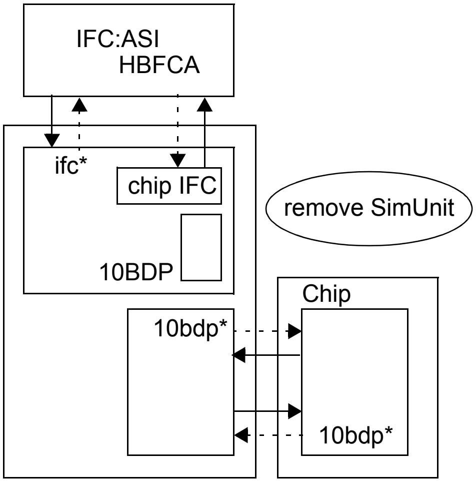
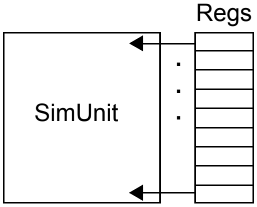

IFC:ASI
HBFCA
ifc*
chip IFC
remove SimUnit
10BDP
Chip
10bdp*
10bdp*

FIGURE 1.4



outside of each chip constructor

```
pSE->RV=new regVec();
c=newChip
  .pushback(Reg)
loop on RV
r->setGcRc()
```

FIGURE 1.5

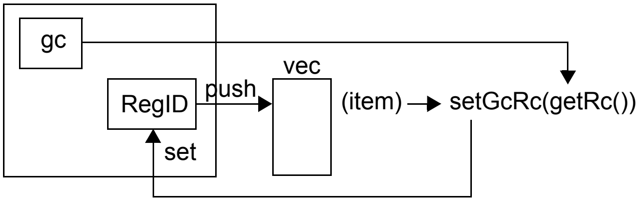
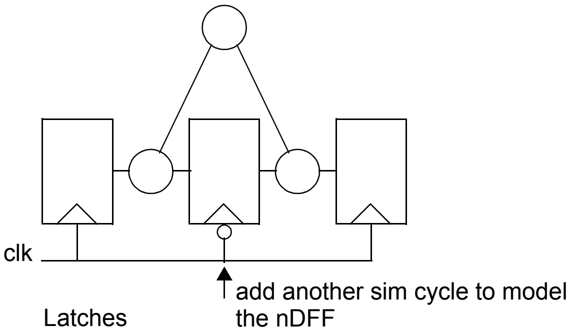


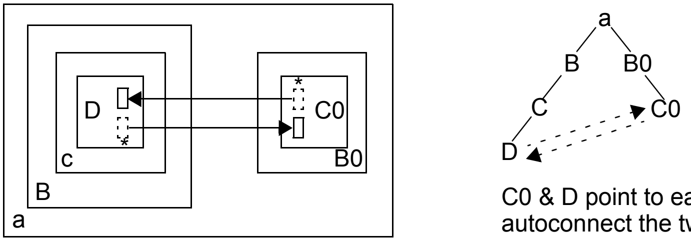
FIGURE 1.6 C++Sim State Elements



add another sim cycle to model the nDFF

Generate verilog interconnect using C++

FIGURE 1.7



C0 & D point to each others' regs
autoconnect the two

pointers to regs/wires are interconnect
the hierarchy is created by calling FUB constructors inside of other FUB constructors HID's are created by passing strings & num to FUB constructors inside of other FUB constructors and concatenating(?) the name of the instance to the num

```
bar:bar(nm){
  foo(nm+string(".foo"));
}
```

Each Fub has a child Fub vec v<FUB *> and a parent FUB*
A tree is constructed by traversing the child_fub_vec. FUB::traverse is a virtual function which iterates through the child_fub_vec, visiting each child FUB*

```
class wire
  int or LLU data;
  int width;
  WIRETYPE wt; // ?? for wire types
```

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 vars 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(?)).

FIGURE 1.8

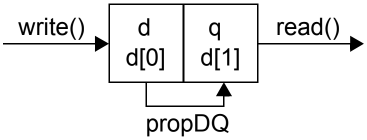
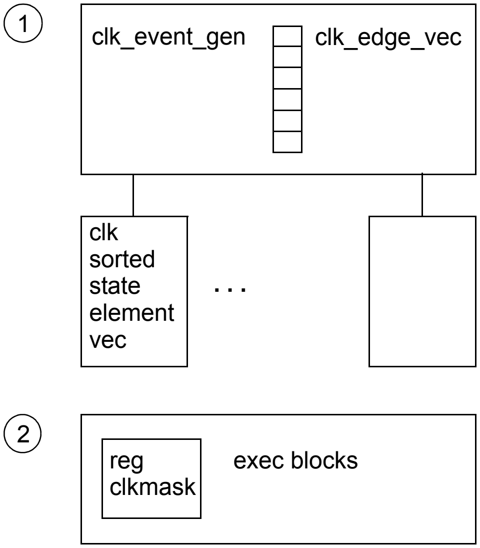


FIGURE 1.9



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

What is turn around time? It is the amount of time it takes to recompile a module and resimulate the module. Figure9. “dynamic vs static routing and compile times (turn around time)”, on page 16.

FIGURE 1.10 dynamic vs static routing and compile times(turn around time)

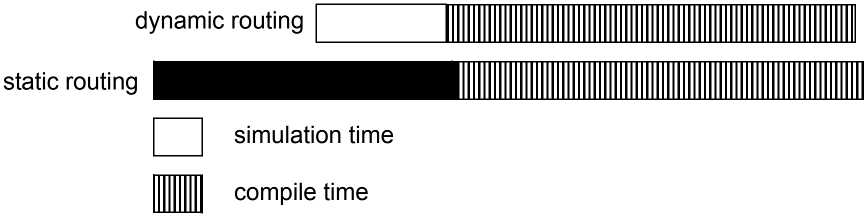
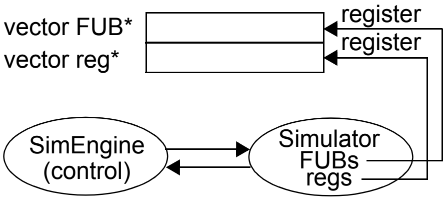


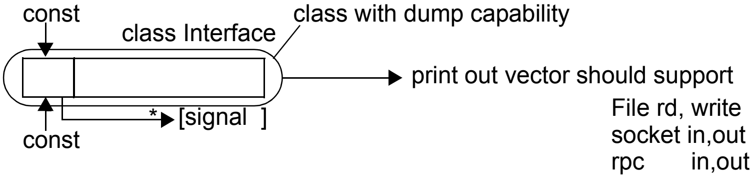
FIGURE 1.11 Array



```
breakpoint
program load ascii binary
debugger FUB*ctFUB(<"name">) // returns pointer to FUB
watch pts // regs/mems to break on help
print FUBs //prints out list of accesibile FUBs

load (ascii or binary filename) <- contain multiple program files
scope
print scope // prints regs in scope
execute {
  foreach(r regvec){
    r.exec();
    }
  }
prop {
  foreach f in fubvec
    f.prop();
  }
```

FIGURE 1.12 Interconnect between modules using vectors & constants by both modules



SimEngine

```
vector<*FUB>
vector<*memory>
vector<*reg>  // one reg class polymorphic functions
vector <*breakpoint>
class reg <<type>> {
  <type> master
  <type> slave
  unsigned char en
  write(<type>in){master=in;}
  exec (void) {slave=master;}
     en=0;
typedef unsigned char UCHAR;

class memory
  dump every n cycles(int n)
```

1) Need a string class. Need a name buffer class -> get out of 1st chapter of austern's book
2) Need a dependency calculator -> get out of largescale C++ cut to verilog env & gen makefiles
3) Need verilog tools gen tb   gen ifc
4) verification tools gen ran vecs

```
#ifndef REG_H
#define REG_H
#include <stdio.h>
#include <assert.h>
#include <vector>
#include <iostream>

#include “smacro.h”

class SimArch;
class BaseReg;

//typedef vector<BaseReg*>::iterator BRPI;

class BaseReg {
private:

public:

  BaseReg(){};
  ~BaseReg(){};

  virtual void prop()=0;
  virtual void print()=0;

};

//The next value pointer is a pointer to the previous element which is
//the driver
//If the default constructor is called then the Reg Class new’s a new T
//and uses
//There are three different kinds of constructors that are used
//1. Default -- will be initialized to a constant
//2. scalar type constructor for interfacing to combinatorial logic
//3. Reg<T> type constructot for interfacing to a register from the
//previous pipe (used for linking pipelines)


template<typename T>
class Reg : public Base {
private:

  // T, n;
  T* n;
  T p;

public:
```

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)

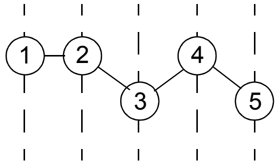
-State elements may be enabled with local signals

-There is a hierarchy of state elements which is used to implement the virtual functions.

Find next clk edge

drives sim cycle

FIGURE 1.13 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
```

Simulator Arch
read reg
en reg

FIGURE 1.14

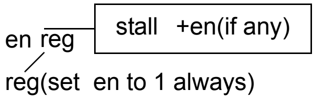
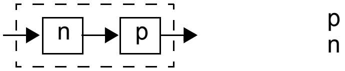


FIGURE 1.15



reg constructor registers reg(this) with SimArch object

```
static pSA

class SimArch {
  private:
    vector<Reg*> vReg; //pointers to regs
  public:
    void propRegs() {
    foreach r(vReg)
      r->prop();
    }
    void assignRegs()
};

template<T>;
class Reg {
  private:
    T *n;
    T *p;
  public:
    Reg() { pSA->reg Reg(this); }
    void prop() { p=n; }
    void assign(T pT) { n=pT; }
```