System-on-Chip Design
RTL Modeling

Dr. Hao Zheng
Comp. Sci & Eng.
U of South Florida

Outline

- Data Types for logic circuits
- Modeling Combinational Logic
- Modeling Synchronous Logic

Data Types for Modeling Logic Designs

Logic and Bounded Integer Types

- Logic types

```
sc_bit var;     //Values: '0','1'
sc_logic var;   //Values: '0','1','X','Z'
sc_bv<length> var;  //bit vector
sc_lv<length> var;  //logic vector
```

- Bounded integer types

```
sc_int<length> var;   //signed int (<= 64bit)
sc_uint<length> var;  //unsigned int (<= 64bit)
sc_bigint<length> var;   //signed int
sc_biguint<length> var;  //unsigned int
```

sc_logic

'0', SC_LOGIC_0	false
'1', SC_LOGIC_1	true
'X', 'x', SC_LOGIC_X	unknown
'Z', 'z', SC_LOGIC_Z	high-impledance

Data Operations

Type **Operation**	sc_bit sc_bc sc_lv	sc_bc sc_lv	sc_int, sc_uint sc_bigint, sc_biguint
Bitwise	~ & ^ |	~ & ^ | << >>	~ & ^ | << >>
Arithmetic			+ - * / %
Logical			
Equality	== !=	== !=	== !=
Relational			> < <= >=
Assignment	= &= |= ^=	= &= |= ^=	= += -= *= /= %= &= |= ^=
Increment Decrement			++ --
Arithmetic if			
Concatenation	,	,	,
Bitselect		[x]	[x]
Partselect		range()	range()

Examples

```
sc_logic x;
x = '1';
x = 'Z';

sc_bit a;
sc_logic b, c;
a == b;
b != c;
b == '1';

sc_lv<4> val;
val = "110X";
```

```
sc_int<16> x, y, z;

z = x & y;  //bitwise-or of x and y
z = x >> 4;  //x right shift by 4
x = x + y;

sc_uint<8> ui1, ui2;
sc_int<16> i1, i2;
ui1 = i2;   // type conversion and
i1 = ui2;   // possible trunccation
```

Examples

```
sc_lv<8> ctrl_bus;
sc_lv<4> mult;

ctrl_bus[5] = '0';
ctrl_bus.range(0,3) = ctrl_bus.range(7,4);
mult = ctrl_bus.range(2,5);
```

Vector and Ranges

```
// Bit select or range() cannot be
// applied to a port or a signal

sc_in<sc_uint<4> > data;
sc_signal<sc_bv<6> > counter;
sc_uint<4> temp;
sc_uint<6> cnt_temp;
bool mode, preset;

mode = data[2]; // not allowed

temp = data.read();
mode = temp[2];
```

Arithmetic Operations

All fixed precision integer type calculations occur on a 64-bit representation and appropriate truncation occurs depending on the target result size

Arithmetic Operations

```
sc_uint<4> write_addr;
sc_int<5> read_addr;

read_addr = write_addr + read_addr;
```

- E.g.:
  - write_addr is zero-extended to 64-bit
  - read_addr is sign-extended to 64-bit
  - + is performed on 64-bit data
  - result is truncated to 5-bit result and assigned back to read_addr

Arithmetic Operations

```
// Arithmetic operations cannot be
// applied to bit/logic vectors.

sc_in<sc_bv<4> > pha1;
sc_signal<sc_bv<6> > pha2;
sc_uint<4> uint_pha1;
sc_uint<6> int_pha2;

uint_pha1 = pha1;
uint_pha2 = pha2 - uint_pha1;
pha2 = uint_pha2;
```

Modeling Combinational Logic

Combinational Circuit Modeling

- Logic is described in member functions.
- SC_METHOD is used to implement those functions.
- Every SC_METHOD processes must have a sensitivity list that includes all input signals that a combinational logic reads.
- Every signal/variable must be assigned in every branch of a if/switch statement.

Ports and Signals

- Declaration

```
sc_in<port_type> port_name;
sc_out<port_type> port_name;
sc_inout<port_type> port_name;
sc_signal<signal_type> signal_name;
```

- Operations

read()	retunrs value of signal or port
write()	assigns value to signal or port
event()	returns true or false if event on signal or port
default_event()	any change of value
value_changed_event()	any change of value
posedge()	returns true if 0 -> 1 transition
negedge()	returns true if 1 -> 0 transition

Specialized Ports and Signals

```
sc_in<T> port_name;
sc_out<T> port_name;
sc_inout<T> port_name;
sc_signal<T> signal_name;
```

where T is bool or sc_logic, there are additional member functions available.

posedge_event()	returns a sc_event
negedge_event()	returns a sc_event
posedge()	returns bool
negedge()	returns bool

Reading and Writing Ports and Signals

```
#include "systemc.h"

SC_MODULE(xor_gates) {
  sc_in<sc_uint<4> > a, b;
  sc_out<sc_uint<4> > c;

  void prc_xor_gates() { c = a ^ b; }

  SC_CTOR(xor_gates) {
    SC_METHOD(prc_xor_gates);
    sensitive << a << b;
  }
}
```

logic operators are not allowed for ports.

A Compiler error may occur! Arithmetic ops cannot be applied to ports/signals directly.

Reading and Writing Ports and Signals

```
#include “systemc.h”

SC_MODULE(xor_gates) {
  sc_in<sc_uint<4> > a, b;
  sc_out<sc_uint<4> > c;

  void prc_xor_gates() {
    c = a.read() ^ b.read();
  }

  SC_CTOR(xor_gates) {
    SC_METHOD(prc_xor_gates);
    sensitive << a << b;
  }};
```

Reading and Writing Ports and Signals

```
#include “systemc.h”

SC_MODULE(xor_gates) {
  sc_in<sc_uint<4> > a, b;
  sc_out<sc_uint<4> > c;

  void prc_xor_gates() {
    c.write(a.read() ^ b.read();
  }

  SC_CTOR(xor_gates) {
    SC_METHOD(prc_xor_gates);
    sensitive << a << b;
  }};
```

SystemC Half Adder

```
// File half_adder.cpp
#include “systemc.h”

SC_MODULE(half_adder) {
   sc_in<bool> a, b;
   sc_out<bool> sum, carry;

   void prc_half_adder() { … }

   SC_CTOR(half_adder) {
     SC_METHOD(prc_half_adder);
     sensitive << a << b;
   }
};
```

SystemC Decoder 2/4

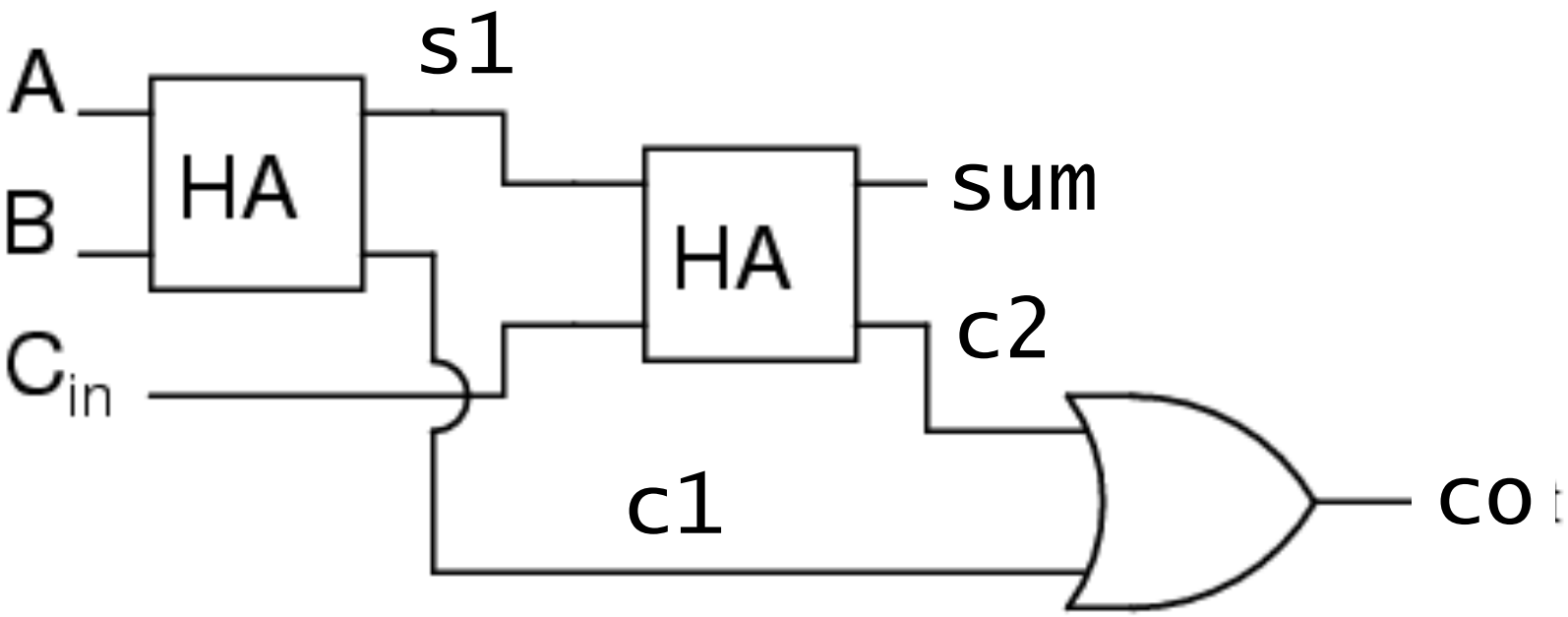
```
// decode.h
#include “systemc.h”

SC_MODULE(decoder2by4) {
   sc_in<bool> enable;
   sc_in<sc_uint<2> > select;
   sc_out<sc_uint<4> > z;

   void prc_decoder2by4() { … }

   SC_CTOR(decoder2by4) {
     SC_METHOD(prc_half_adder);
     sensitive(enable, select);
   }};
```

SystemC Decoder 2/4

```
// decode.cpp
#include “decode.h”
void decoder2by4::prc_decoder2by4()
{
  if (enable) {
    switch(select.read()) {
      case 0: z=0xE; break;
      case 1: z=0xD; break;
      case 2: z=0xB; break;
      case 3: z=0x7; break;
    }
  }
  else
    z=0xF;
}
```

Switch statements are synthesized to multiplexers.

Hierarchy: Building a full adder



Hierarchy: Building a full adder

```
#include “systemc.h”

SC_MODULE(full_adder) {
  sc_in<bool> a,b,cin;
  sc_out<bool> sum, co;
  sc_signal<bool> c1, s1, c2;
  half_adder *ha1_ptr, *ha2_ptr;

  void prc_or() { co = c1 | c2; }

  SC_CTOR(full_adder) { … }

  ~full_adder() { … }
};
```

Hierarchy: Building a full adder

```
SC_CTOR(full_adder) {
  ha1_ptr = new half_adder("ha1");
  ha1_ptr->a(a);
  ha1_ptr->b(b);
  ha1_ptr->sum(s1);
  ha1_ptr->carry(c1);

  ha2_ptr = new half_adder("ha2");
  (*ha2_ptr)(s1, cin, sum, co);

  SC_METHOD(prc_or);
  sensitive << c1 << c2;
}
```

A Behavioral Model of a Full Adder

```
// File: fa.h
#include “systemc.h”

SC_MODULE(fa) {
  sc_in<sc_int<4> > a, b;
  sc_out<sc_int<4> > sum;
  sc_out<bool> cout;

  void prc_fa();

  SC_CTOR(decoder2by4) {
    SC_METHOD(prc_fa);
    sensitive(a, b);
  }};
```

A Behavioral Model of a Full Adder

```
// File: fa.cpp
#include “fa.h”

void fa::prc_fa() {
   sc_in<sc_int<5> > temp;

   temp = a.read() + b.read();
   sum = temp.range(3,0);
   cout = temp[4];
}
```

The same circuit is synthesized if the input type is **`sc_uint`**.

Comparator Curcuit

```
SC_MODULE(neq) {
   sc_in<sc_int<4> > a, b;
   sc_out<bool> z;

   void prc_neq() { z = (a != b); }

   SC_CTOR(neq) {
      SC_METHOD(prc_neq);
      sensitive << a << b;
}
```

Different comparison circuits are synthesized for different relational operators used..

Loops for Synthesis

- C++ loops: for, do-while, while
- SystemC RTL supports only for loops
  - For loop iteration must be a compile time constant

```
// File: demux.h
#include “systemc.h”

SC_MODULE(demux) {
   sc_in<sc_uint<2> > a;
   sc_out<sc_uint<4> > z;

   void prc_demux();

   SC_CTOR(demux) {
     SC_METHOD(prc_demux);  sensitive << a;
   }};
```

Loops: An Example

```
// File: demux.cpp
#include “demux.h”

void demux::prc_demux(){
   sc_uint<3> j;
   sc_uint<4> temp;

   for(j=0; j<4; j++)
     if(a==j)    temp[j] = 1;
     else        temp[j] = 0;

   z = temp;
}
```

Loops are typically unrolled during synthesis.

Methods

```
// File: odd1s.h
#include “systemc.h”

SC_MODULE(odd1s) {
  sc_in<sc_uint<6> > din;
  sc_out<bool> is_odd;

  bool isOdd(sc_uint<SIZE> abus);
  void prc_odd1s();

  SC_CTOR(odd1s) {
    SC_METHOD(prc_odd1s); sensitive << data_in;
  }};
```

Methods other than SC_METHOD processes can also be used

Methods

```
// File: odd1s.cpp
#include “odd1s.h”

void odd1s::prc_odd1s(){
   is_odd = isOdd(din);
}
bool odd1s::isOdd(sc_uint<6> abus) {
   bool result;

   for(int i=0; i<6; i++)
     result = result ^ abus[i];

   return(result); }
```

During synthesis, the code in `prc_odd1s()` is inlined into `is_odd()`

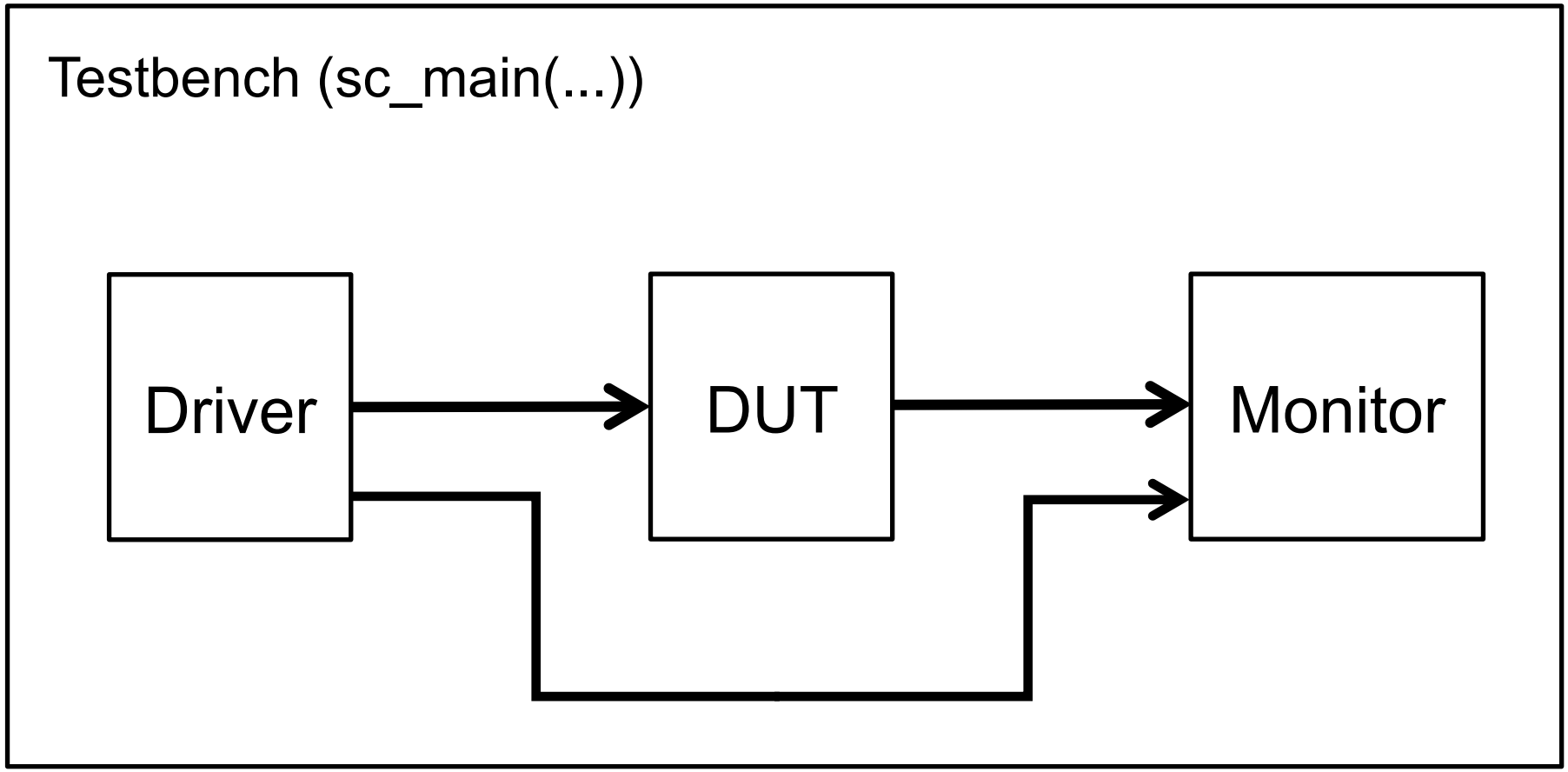
Multiple Processes

```
// File: mult_proc.h
#include "systemc.h"

SC_MODULE(mult_proc) {
   sc_in<bool> in;
   sc_out<bool> out;

   sc_signal<bool> c1, c2;

   void mult_proc1();
   void mult_proc2();
   void mult_proc3();

   SC_CTOR(mult_proc) {
     SC_METHOD(mult_proc1);  sensitive << in;
     SC_METHOD(mult_proc2);  sensitive << c1;
     SC_METHOD(mult_proc3);  sensitive << c2;
   }};
```

Combinational circuit can also be modeled with multiple processes.

Inter-process communications are done via signals.

Verification



Verifying the Functionality: Driver

```
// File driver.h
#include “systemc.h”

SC_MODULE(driver) {
   sc_out<bool> d_a, d_b, d_cin;

   // Generate input stimulus
   void prc_driver();

   SC_CTOR(driver) {
     SC_THREAD(prc_driver);
   }};
```

Verifying the Functionality: Driver

```
// File: driver.cpp
#include “driver.h”

void driver::prc_driver(){
   sc_uint<3> pattern;
   pattern=0;

   while(1) {
     d_a=pattern[0];
     d_b=pattern[1];
     d_cin=pattern[2];
     wait(5, SC_NS);
     pattern++;
   }}
```

pattern defines values for inputs of DUT

Verifying the Functionality: Monitor

```
// File monitor.h
#include “systemc.h”

SC_MODULE(monitor) {
   sc_in<bool> m_a, m_b, m_cin, m_sum, m_cout;

   void prc_monitor();

   SC_CTOR(monitor) {
     SC_THREAD(prc_monitor);
     sensitive << m_a,m_b, m_cin, m_sum, m_cout;

   }};
```

Verifying the Functionality: Monitor

```
// File: monitor.cpp
#include “monitor.h”

void monitor::prc_monitor(){
   cout << “At time “ <<
   sc_time_stamp() << “::”;
   cout <<“(a, b, carry_in): ”;
   cout << m_a << m_b << m_cin;
   cout << “(sum, carry_out): ”;
   cout << m_sum << m_cout << endl;
}
```

Observe inputs and outputs of DUT, and check if outputs are correct wrt inputs.

Verifying the Functionality: Main

```
#include “driver.h”
#include “monitor.h”
#include “full_adder.h”

int sc_main(int argc, char * argv[]) {
   sc_signal<bool> t_a, t_b, t_cin, t_sum, t_cout;

   full_adder f1(“FullAdderWithHalfAdders”);
   f1(t_a, t_b, t_cin, t_sum, t_cout);

   driver d1(“GenWaveforms”);
   d1.d_a(t_a);
   d1.d_b(t_b);
   d1.d_cin(t_cin);

   monitor m1(“MonitorWaveforms”);
   m1(t_a, t_b, t_cin, t_sum, t_cout);

   sc_start(100, SC_NS);
   return(0);  }
```

Modeling Sequential Logic

Edge Detection

- Positive or negative edges can be detected on signals or ports of `bool` type.
  - `signal_name.posedge()`: return true or false
  - `signal_name.posedge()`: return true or false
  - `signal_name.posedge_event()`
    - ✦return an 0 -> 1 event
  - `signal_name.negedge_event()`
    - ✦return an 1 -> 0 event

Modeling Synchronous Logic: Flip-flops

```
// File: dff.h
#include “systemc.h”

SC_MODULE(dff) {
   sc_in<bool> clk, reset, d;
   sc_out<bool> q;

   void prc_dff();

   SC_CTOR(dff) {
     SC_METHOD(prc_dff);
     sensitive_pos << clk; // edge sensitivity
     // sensitive_neg << clk;
   }};
```

Modeling Synchronous Logic: Flip-flops

```
// File: dff.h
#include “systemc.h”

SC_MODULE(dff) {
  sc_in<bool> clk, reset, d;
  sc_out<bool> q;

  void prc_dff();

  SC_CTOR(dff) {
    SC_METHOD(prc_dff);
    sensitive << clk.posedge_event();
     // sensitive << clk.negedge_event();
  }};
```

Output ports and signals can be synthesized to FFs.

Modeling Synchronous Logic: Flip-flops

```
// File: dff.cpp version 1
#include “dff.h”

void dff::prc_dff(){
   q = d;
}
```

```
// File: dff.cpp version 2
// FF w synchronous reset
#include “dff.h”

void dff::prc_dff(){
   if (!reset.read()) q = false;
   else               q = d;
}
```

Registers

```
#include “systemc.h”
const int WIDTH = 4;

SC_MODULE(reg) {
  sc_in<bool> clock;
  sc_in<sc_uint<WIDTH> > cstate;
  sc_out<sc_uint<WIDTH> > nstate;

  void prc_reg() { nstate = cstate; }

  SC_CTOR(dff) {
    SC_METHOD(prc_dff);
    sensitive_neg << clock;
  }};
```

Sequence Detector: "101"

```
// File: sdet.h
#include "systemc.h"

SC_MODULE(sdet) {
   sc_in<bool> clk, data;
   sc_out<bool> sfound;
   sc_signal<bool> first, second, third;

   void prc_sdet();
   void prc_out();

   SC_CTOR(sdet) {
     SC_METHOD(prc_sdet);
     sensitive_pos << clk;
     SC_METHOD(prc_out);
     sensitive << first << second << third;
   }};
```

A model contain multiple processes, some for combinational logic, some for sequential logic

Sequence Detector: “101”

```
// File: sdet.cpp
#include “sdet.h”

// A shift register
void sdet::prc_sdet(){
   first = data;
   second = first;
   third = second;
}

// match the shift reg to string pattern
void sdet::prc_out(){
   sfound = first & (!second) & third;
}
```

Counter: Up-down, Async Negative Clear

```
// File: cnt4.h
#include “systemc.h”
const int CSIZE = 4;

SC_MODULE(sdet) {
   sc_in<bool> mclk, clear, updown;
   sc_out<sc_uint<CSIZE> > dout;

   void prc_cnt4();

   SC_CTOR(cnt4) {
     SC_METHOD(prc_cnt4);
     sensitive_pos << mclk;
     sensitive_neg << clear;
}};
```

Counter: Up-down, Async Negative Clear

```
// File: cnt4.cpp
#include “cnt4.h”

void sdet::prc_cnt4(){
  if(!clear)
    data_out = 0;
  else
    if(updown)
      dout=dout.read()+1;
    else
      dout=dout.read()-1;
}
```

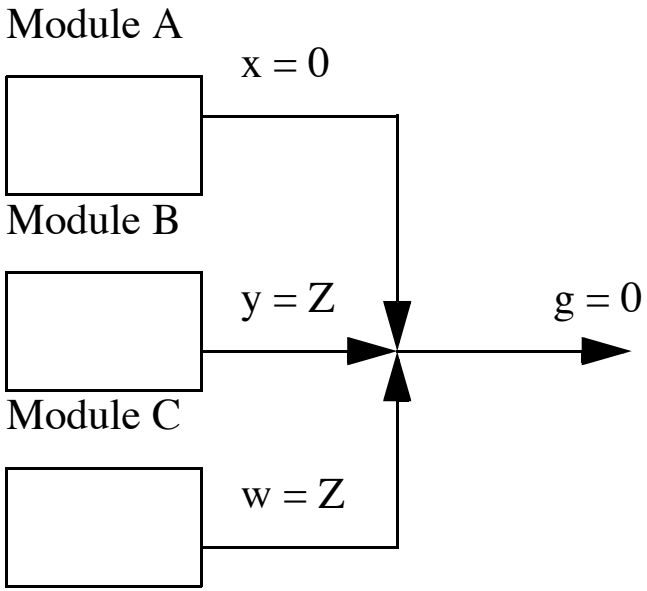
Template for Sequential Process with Asynchronous Reset

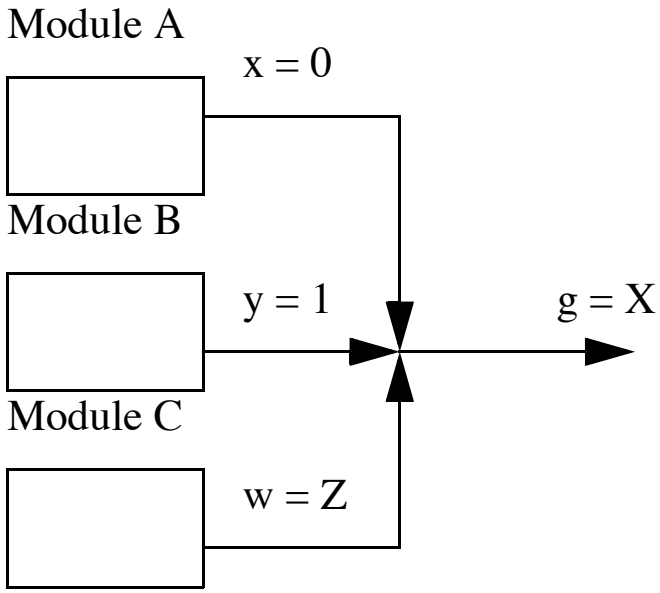
```
void my_module::my_process () {
  if (a)              // Positive value used, since positive
                      // edge specified.
    <asynchronous behavior>
  else if (b)
    <asynchronous behavior>
  else if (! d)       // Logical-not used, since negative
                      // edge specified.
    <asynchronous behavior>
  else if (! e)
    <asynchronous behavior>
  else if (! f)
    <asynchronous behavior>
  else                // Rising clock edge.
    <clocked behavior>
}
```

Multiple Clocks

```
#include “systemc.h”

SC_MODULE(multiclock) {
   sc_in<bool> clk1, clk2;
   sc_out<sc_uint<CSIZE> > ...;
   sc_signal<bool> ...;

   void prc_clk1() {}
   void prc_clk2() {}

   SC_CTOR(multiclock) {
     SC_METHOD(prc_clk1);
     sensitive_pos << clk1;
     SC_METHOD(prc_clk2);
     sensitive_neg << clk2;
   }};
```

Multiple Drivers and Resolved Logic





	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	Z	X
X	X	X	X	X

Multiple Drivers

```
#include “systemc.h”
const BUS_SIZE = 4;

SC_MODULE(bus) {
   sc_in<bool> a_rdy, b_rdy;
   sc_in<sc_uint<BUS_SIZE> > a_bus, b_bus;
   sc_out_rv<BUS_SIZE> z_bus; //resolved type

   void prc_a_bus();
   void prc_b_bus();

   SC_CTOR(bus) {
     SC_METHOD(prc_a_bus);
     sensitive << a_rdy << a_bus;
     SC_METHOD(prc_b_bus);
     sensitive << b_rdy << b_bus;
   }};
```

Multiple Drivers

```
void bus::prc_a_bus() {
   if (a_rdy)
     z_bus = a_bus.read();
   else
     z_bus = “ZZZZ”;
}

void bus::prc_b_bus() {
   if (b_rdy)
     z_bus = b_bus.read();
   else
     z_bus = “ZZZZ”;
}
```

`sc_signal_rv<n>`: resolved signal vector type

Variables or Signals

- Local variables declared within SC_METHOD processes do not have memory.
- Assignment to variables happens instantly.
- Assignment to signals happens after a *delta* delay.

Shift Register

```
#include “systemc.h”

SC_MODULE(shifter) {
   sc_in<bool> clk, din;
   sc_out<bool> dout;
   bool q; //may cause dout to carry wrong value

   void prc_reg1() { q = din; }
   void prc_reg2() { dout = q; }

   SC_CTOR(shifter) {
     q = 0;
     SC_METHOD(prc_reg1);
     sensitive_pos << clk;
     SC_METHOD(prc_reg2);
     sensitive_pos << clk;
   }};
```

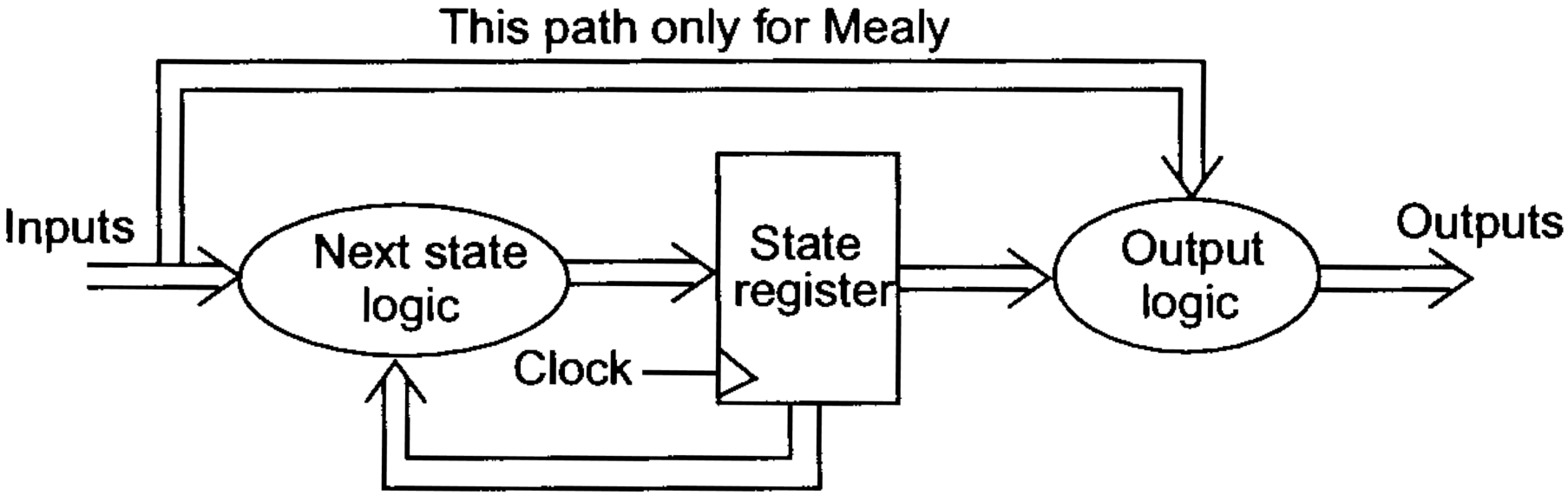
Shift Register (1)

```
#include “systemc.h”

SC_MODULE(shifter) {
   sc_in<bool> clk, din;
   sc_out<bool> dout;
   sc_signal<bool> q;

   void prc_reg1() { q = din; }
   void prc_reg2() { dout = q; }

   SC_CTOR(shifter) {
     q = 0;
     SC_METHOD(prc_reg1);
     sensitive_pos << clk;
     SC_METHOD(prc_reg2);
     sensitive_pos << clk;
   }};
```

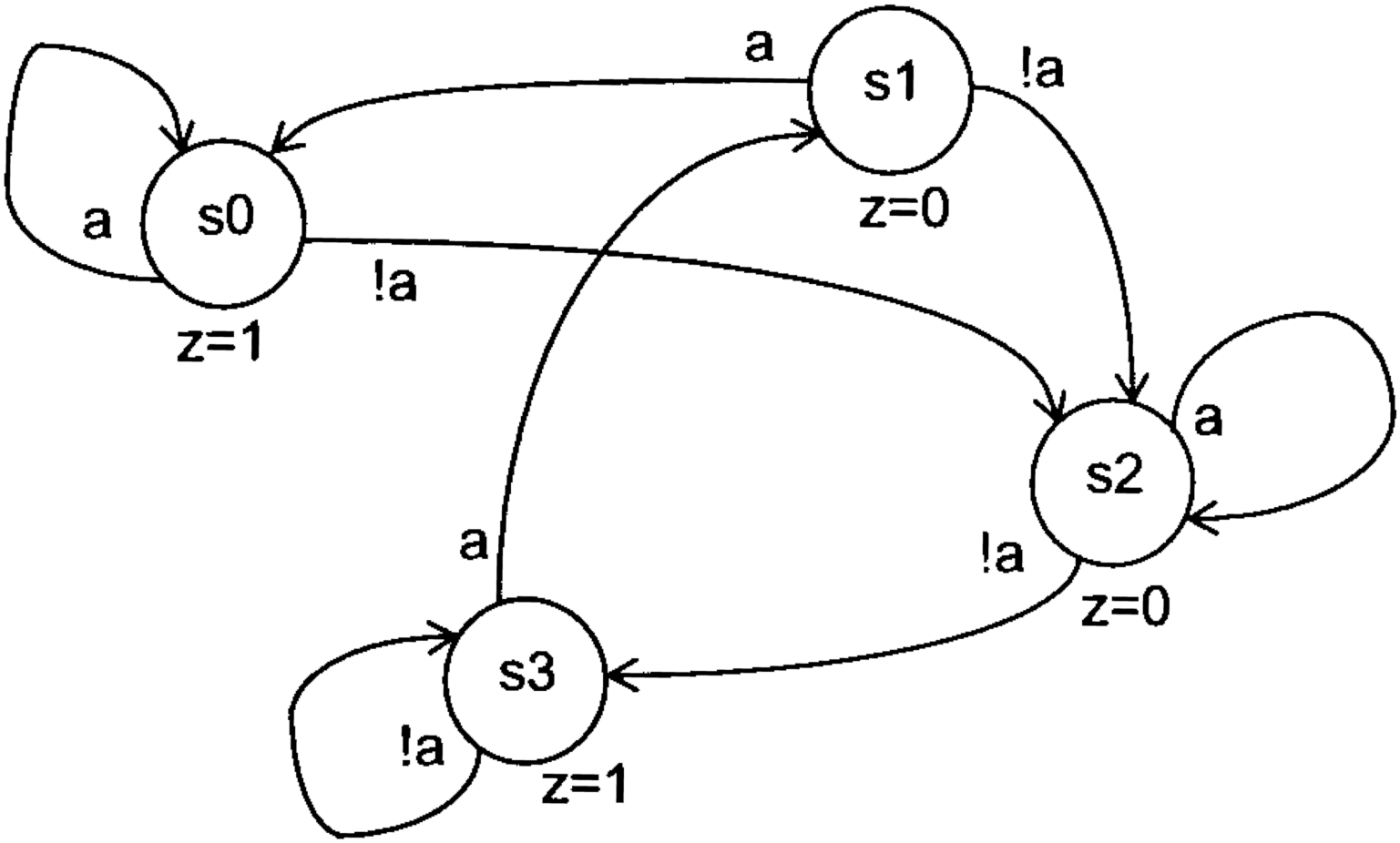
Shift Register (2)

```
#include “systemc.h”

SC_MODULE(shifter) {
   sc_in<bool> clk, din;
   sc_out<bool> dout;
   sc_signal<bool> q;

   void prc_shift() { q = din; dout = q;}

   SC_CTOR(shifter) {
     q = 0;
     SC_METHOD(prc_reg1);
     sensitive_pos << clk;
     SC_METHOD(prc_reg2);
     sensitive_pos << clk;
   }};
```

Finite State Machines



Modeling Moore FSM



Modeling Moore FSM

```
#include “systemc.h”

SC_MODULE(moore) {
   sc_in<bool> clk, reset, a;
   sc_out<bool> z;

   enum state_type {s0, s1, s2, s3 };
   sc_signal<state_type> moore_state;

   void prc_moore();

   SC_CTOR(moore) {
     SC_METHOD(prc_moore);
     sensitive_pos << clk;
   }};
```

Modeling Moore FSM

```
void moore::prc_moore() {
   if (reset) {
      moore_state = s0;
      z = 1;
   }
   else
      switch (moore_state) {
         case s0:
             z = 1;
             moore_state = a ? s0 : s2;
             break;
         case s1:
             ...
         case s2:
             ...
         case s3:
             ...
     }}
```

A extra FF is derived for z

Modeling Moore FSM

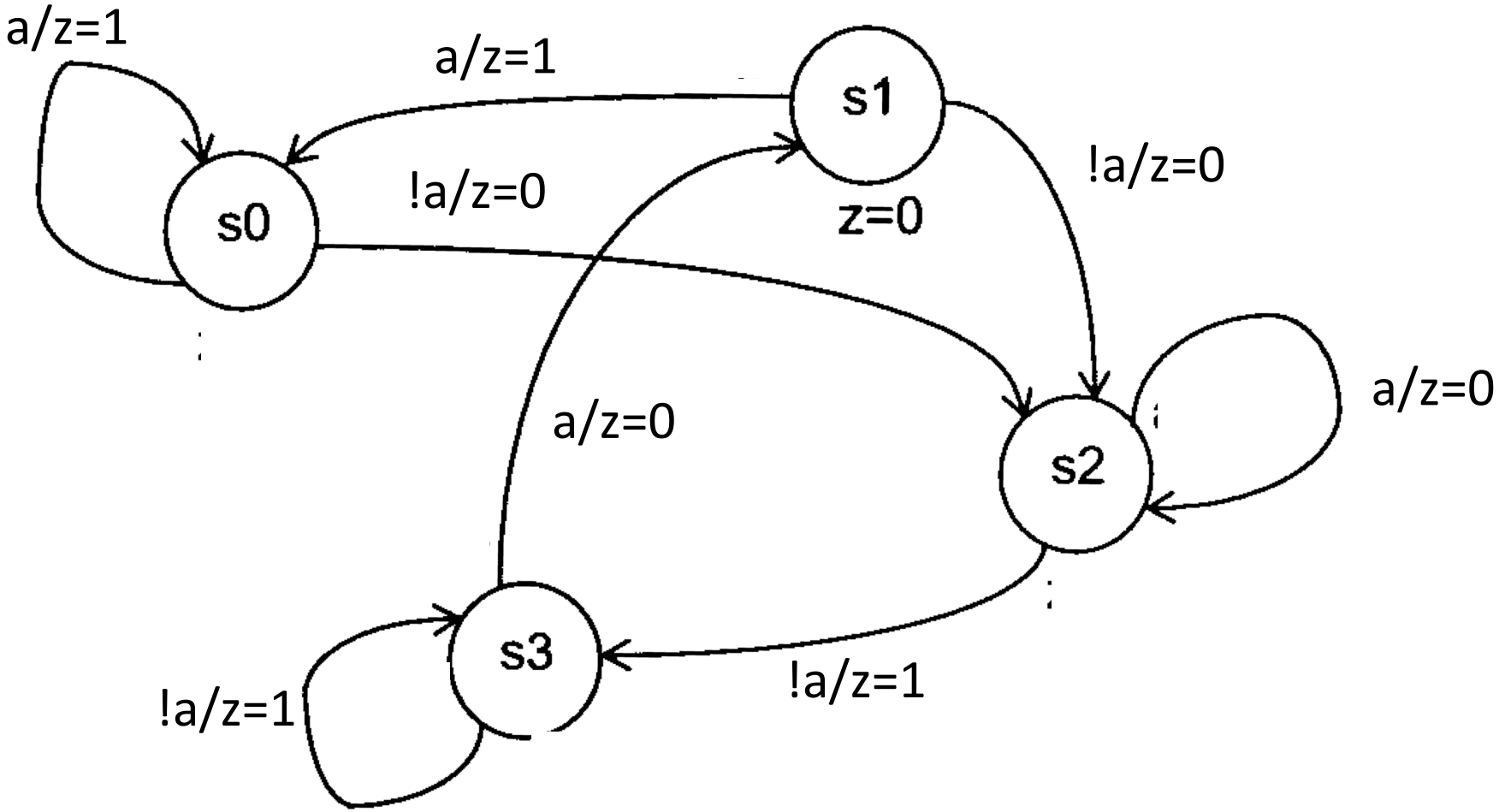
```
#include “systemc.h”

SC_MODULE(moore) {
   sc_in<bool> clk, reset, a;
   sc_out<bool> z;

   enum state_type {s0, s1, s2, s3 };
   sc_signal<state_type> moore_state;

   void prc_states();
   void prc_output();

   SC_CTOR(moore) {
     SC_METHOD(prc_states); sensitive_pos << clk;
     SC_METHOD(prc_output); sensitive << moore_state;
   }};
```

Separate state transition logic from output logic to avoid introducing FFs for output signals.

Modeling Moore FSM

```
void moore::prc_states() {
   if (reset)
      moore_state = s0;
   else
      switch (moore_state) {
         case s0:
              moore_state = a ? s0 : s2;
              break;
              //Other cases
              ...
     }}

void moore::prc_output() {
   switch (moore_state) {
     case s0: z = 1; break;
     case s1: z = 0; break;
     case s2: z = 1; break;
     case s3: z = 0; break;
   }}
```

Modeling Mealy FSM



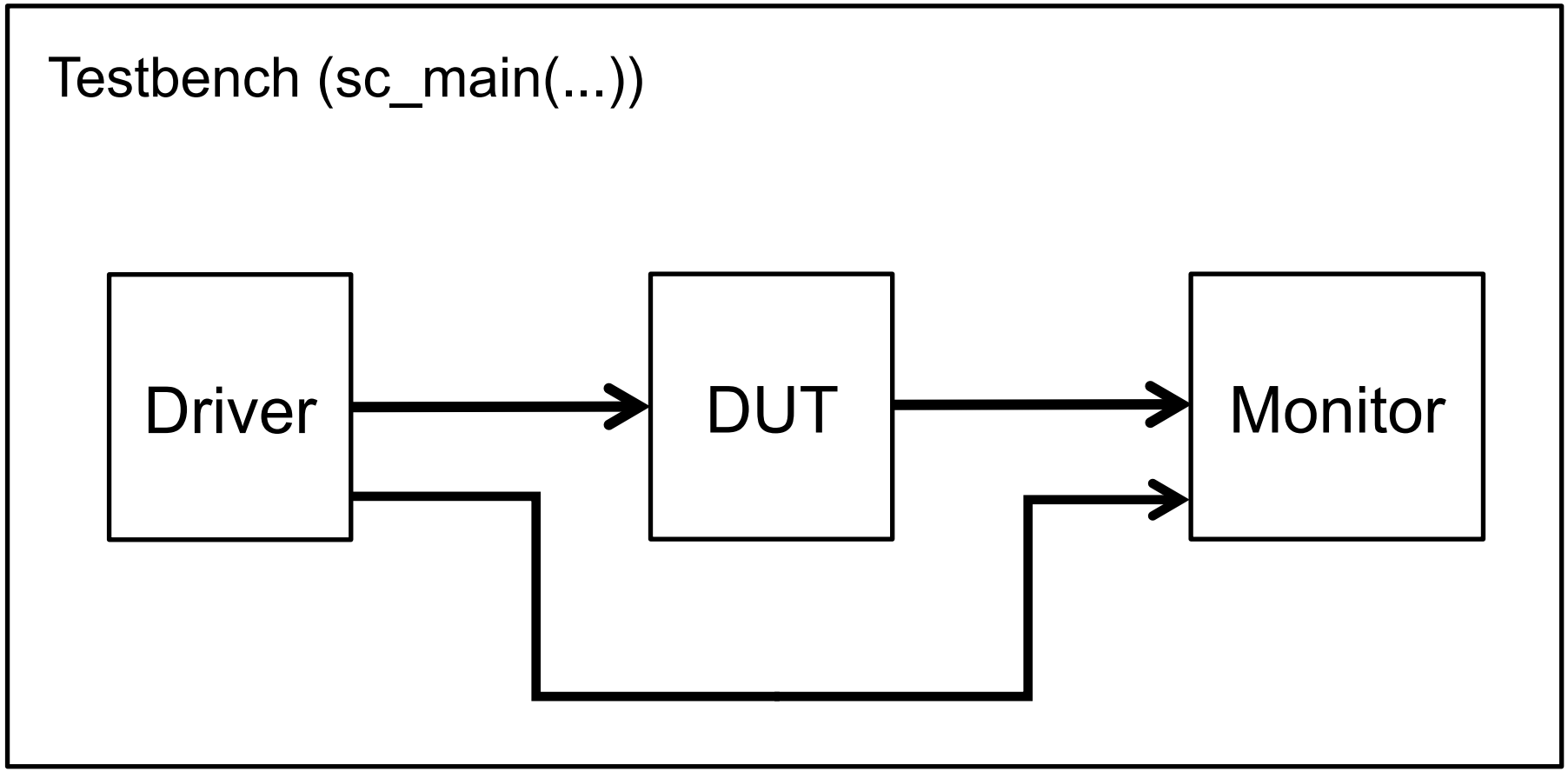
Modeling Mealy FSM

```
#include “systemc.h”

SC_MODULE(mealy) {
   sc_in<bool> clk, reset, a;
   sc_out<bool> z;

   enum state_type {s0, s1, s2, s3 };
   sc_signal<state_type> cstate, nstate;

   void prc_state();
   void prc_output();

   SC_CTOR(mealy) {
     SC_METHOD(prc_state);
     sensitive_pos << clk;
     SC_METHOD(prc_output);
     sensitive << cstate << a;
   }};
```

Mealy FSM

```
void mealy::prc_state() {
   if (reset)
      cstate = s0;
   else
      cstate = nstate;
}

void mealy::prc_output() {
   z = 1;
   switch (cstate) {
     case s0: if (!a) { z = 0; nstate = s2; }
              else   { z = 1; nstate = s0; }
              break;
     case s1: ... break;
     case s2: ... break;
     case s3: ... break;
     }}
```

Verification and Testbench



Clock Signal

sc_clock clock_name ("name**", period, duty_cycle, start_time, positive_first**) ;

name: *name* type: char *

period: *clock period* type: variable of type sc_time or constant of type uint64

duty_cycle: *clock duty cycle* type: double default value: 0.5

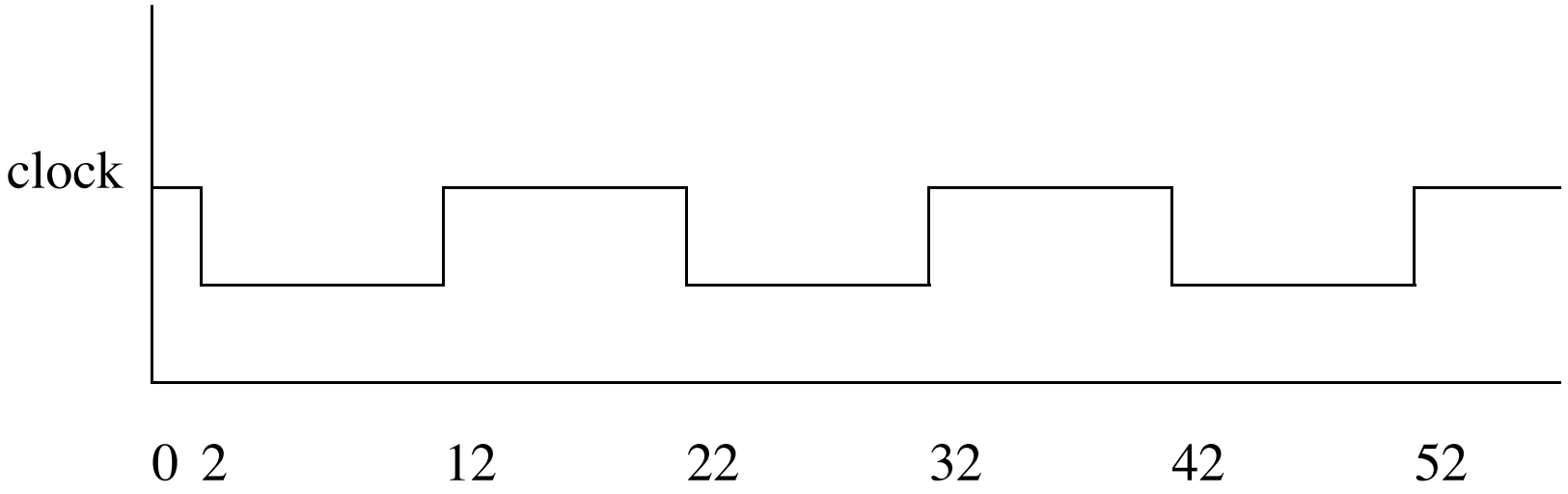
start_time: *time of first edge* type: variable of type sc_time or constant of type uint64

default value: 0

positive_first: *first edge positive* type: bool default value: true

Clock Signal

```
sc_clock clk1("clock1", 20, 0.5, 2, true);
```




```
// To bind a clock object to port
mod.port(clk1.signal());
```

Tracing

Simulation results can be saved in 3 formats:

- VCD – use **`sc_create_vcd_trace_file(name);`**
- WIF – use **`sc_create_wif_trace_file(name);`**
- ISDB – use **`sc_create_isdb_trace_file(name);`**

Create a trace file:

```
sc_trace file *tfile =
        sc_create_vcd_trace_file("myvcddump");
```

Trace a signal:

```
sc_trace(tfile, sig1, "sig_name");
```

Tracing

Create a trace file:

```
sc_trace file *tfile =
      sc_create_vcd_trace_file("myvcddump");
```

Trace a signal:

```
sc_trace(tfile, sig1, "sig_name");
```

After simulation finishes,

```
sc_clock_vcd_trace_file(tfile);
```