Aleksandar Milenkovic
The University of Alabama in Huntsville
Email: milenka@ece.uah.edu
Outline

- Introduction
- Data Types
- Modeling Combinational Logic
- Modeling Synchronous Logic
- Misc
# Data Types and Declarations

- **Variable**
  ```
  // variable
type variable_name1, variable_name2, ...;
  int myint;
  ```

- **Signal**
  ```
  // signal, sc_signal declaration
  // used for interprocess communication and for
  // connecting module instances
  sc_signal<type> signal_name1, signal_name2, ...;
  ```

- **Port**
  ```
  // port
  sc_in<type> input_name1, input_name2, ...;
  sc_out<type> output_name1, output_name2, ...;
  sc_inout<type> inout_name1, inout_name2, ...;
  ```
SystemC Data Types

- SystemC programs may use any C++ type along with any of the built-in ones for modeling systems

- SystemC Built-in Types
  - **sc_bit, sc_logic**
    - Two- (‘0’, ‘1’) and four-valued (‘0’, ‘1’, ‘X’, ‘Z’) single bit
  - **sc_int, sc_uint**
    - 1 to 64-bit signed and unsigned integers
  - **sc_bigint, sc_biguint**
    - arbitrary (fixed) width signed and unsigned integers
  - **sc_bv, sc_lv**
    - arbitrary width two- and four-valued vectors
  - **sc_fixed, sc_ufixed**
    - signed and unsigned fixed point numbers
C++ Types for RTL SystemC

- bool
- int, unsigned int, long, unsigned long
- signed char, unsigned char
- short, unsigned short
- enum
- struct
Sc_bit type

- Operators supported
  - \{&, |, ^, \sim\} - bitwise \{AND, OR, XOR, NOT\}
  - \{=, &=, |=, ^=\} – assignment, compound \{AND, OR, XOR\}
  - \{==, !=\} – equality, inequality

- sc_bit type can be mixed with operands of type bool

```cpp
class sc_signal<sc_bit> flag;
bool ready;

flag = sc_bit(‘0’); // assigns the value ‘0’
ready = ready & flag; // ‘0’ eqs false
if (ready == flag) // ok to do this
```
Sc_bv type

- Arbitrary bit vector
- Rightmost index of the vector is 0 (LSB)
- Examples

```cpp
sc_bv<8> ibus; // variable ibus
sc_out<sc_bv<4> > abus; // port abus
sc_bv<4> mbus; // variable mbus

sc_bv<8> all_ones('1');

ibus = "00110010";
mbus = "010"; // zero extended to "0010"
mbus = "10011"; // truncated "0011"

// to print a value of a bit vector use variable
cout << "The value of ibus is " << ibus << endl;
```
Sc_bv operators

- Operators supported
  - \{&, |, ^, ~\} - bitwise \{AND, OR, XOR, NOT\}
  - \{<<, >>\} – bitwise shift \{left, right\}
  - \{=, &=, |=, ^=\} – assignment, compound \{AND, OR, XOR\}
  - \{==, !=\} – equality, inequality
  - \[ \] – bit selection
  - (,) – concatenation

- Method
  - range() – range selection
  - and_reduce() – reduction AND
  - or_reduce() – reduction OR
  - xor_reduce() – reduction XOR
Sc_bv operators: Examples

```
sc_bv<8> ibus;
sc_out<<sc_bv<4> > abus;
sc_bv<4> mbus;

ibus = "00110000";
mbus = "1011";
ibus[5] = '0'; // ibus = "00010000"
ibus.range(0,3) = ibus.range(7,4); // "00011000"
mbus = (ibus[0], ibus[0], ibus[0], ibus[1]); // mbus="0000"
ibus[0] = ibus.and_reduce();
ibus[1] = mbus.or_reduce();
```
Sc\_bv operators (cont’d)

- Bit selection and the range() method can be applied only to variables (not to ports and signals)
- use temporary variables for ports and signals

```c
sc_signal<sc\_bv<4> > dval;
sc\_in<sc\_bv<8> > addr;
sc\_bv<4> var\_dval;
sc\_bv<8> var\_addr;
sc\_bit ready;
// read bit 2 of input addr
var\_addr = addr.read();
ready = var\_addr[2];
// assign “011” to a range of signal dval
var\_dval = dval;
var\_dval.range(0,2) = “011”;
dval = var\_dval;
```
Sc_bv operators (cont’d)

- No arithmetic operations are allowed on the bit vector types
  - convert to signed or unsigned integer, perform operations, and convert back to a bit vector

```cpp
sc_in<sc_bv<4> > pha1;
sc_signal<sc_bv<6> > pha2;
sc_uint<4> uint_pha1;
sc_uint<6> uint_pha2;

uint_pha1 = pha1; // overloaded assignment
uint_pha2 = pha2;
uint_pha2 = uint_pha2 - uint_pha1;
pha2 = uint_pha2; // overloaded assignment
```
Logic Type

- Four values
  - ‘0’, SC_LOGIC_0
  - ‘1’, SC_LOGIC_1
  - ‘X’, SC_LOGIC_X
  - ‘Z’, SC_LOGIC_Z

- Operators
  - {&, |, ^, ~} - bitwise {AND, OR, XOR, NOT}
  - {<<, >>} – bitwise shift {left, right}
  - {=, &=, |=, ^=} – assignment, compound {AND, OR, XOR}
  - {==, !=} – equality, inequality
Sc_lv Type

- Arbitrary Width Logic Type
- Bit selection, range(), arithmetic
  (see notes for bit vectors)

```cpp
sc_lv<4> dbus; // variable dbus
sc_signal<sc_lv<8> > cnt; // signal, cnt(7 downto 0)
sc_out<sc_lv<16> > sensor; // port

sc_lv<8> allzs(SC_LOGIC_Z);
sc_lv<8> allxs(SC_LOGIC_X);

dbus = "0011";
sensor = "1011X0X01100ZZZZ";

dbus[2] = 'X'; // dbus = "0X11"
dbus[0] = dbus[3]; // dbus = "0X10"
cnt = (dbus[3], dbus[3], dbus[3], dbus[3], dbus[3],
dbus[2], dbus[1], dbus[0]); // concatenation
```
Sc_Iv Type (cont’d)

- Assignments are overloaded to allow translation to and from a logic vector and an integer type.
- Bit vector type and logic vector type can be assigned to each other.
- If a logic vector contains ‘X’ or ‘Z’, the result of translation is undefined and a runtime warning is issued.

```cpp
sc_uint<4> driver;
sc_int<8> q_array;

driver = dbus;
q_array = dbus; // zeros are filled as dbus is unsigned
sensor = q_array; // q_array is signed => sign extended
dbus = driver;

cout << "Data bus has value = " << dbus << endl;
```
Sc_int: Signed Integer Types

- Precision limited to 64 bits
- 2’s complement representation
- All operations are performed using 64 bits and the result is truncated to the target size
Sc_int: Operators & Methods

Operators

- Bitwise operators: {&, |, ^, ~} - AND, OR, XOR, NOT
- Shift operators: {<<, >>} - Arithmetic shift, left, right
- Arithmetic operators: {+, -, *, /, %} - Addition, Subtraction, Multiply, Divide, Modulus
- Assignment and compound operators: {=, &=, |=, ^=, +=, -=, *=, /=, %=} - assignment, compound {AND, OR, XOR}, compound {+, -, * , /, %}
- Equality and inequality: {==, !=} - equality, inequality
- Comparison operators: {<, <=, >, >=} - less than, less than equal, ...
- Increment and decrement: {++, --} - increment, decrement
- Bit selection: [ ]
- Concatenation: (,)

Methods

- range()
Sc\_uint

- Unsigned integer
- Fixed precision integer type to 64 bits
- The same set of operators as sc\_int
- Can be converted to sc\_int and vice versa
Arbitrary Precision Signed/Unsigned Integer

- \( \text{sc\_bigint<WIDTH>, sc\_biguint<WIDTH>} \)
- Used when a precision of more than 64 bits is required
- Format parameters for to\_string()
  - SC\_BIN, SC\_OCT, SC\_HEX, SC\_DEC(default)

```cpp
sc\_bigint<100> comp, big\_reg; // variables

// decimal format
cout << "The value of big\_reg is " << big\_reg.to\_string() << endl;

// hexadecimal format
cout << "The value of big\_reg is " << big\_reg.to\_string(SC\_HEX) << endl;
```
Resolved Types

// resolved logic scalar port type
sc_out_resolved
sc_inout_resolved

// resolved logic vector type
sc_out_rv<WIDTH>
sc_inout_rv<WIDTH>

// resolved logic scalar signal type
sc_signal_resolved

// resolved logic vector signal type
sc_signal_rv<WIDTH>

Resolution Table

<table>
<thead>
<tr>
<th></th>
<th>'0'</th>
<th>'1'</th>
<th>'X'</th>
<th>'Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
<td>'X'</td>
<td>'X'</td>
<td>'0'</td>
</tr>
<tr>
<td>'1'</td>
<td>'X'</td>
<td>'1'</td>
<td>'X'</td>
<td>'1'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
</tr>
<tr>
<td>'Z'</td>
<td>'0'</td>
<td>'1'</td>
<td>'X'</td>
<td>'Z'</td>
</tr>
</tbody>
</table>
User-defined Data Types

- Can be created using the enum and the struct types
- A signal can be declared to be of such a type
- Four overloaded functions that operate on the new data type have to be provided
  - Operator = (assignment)
  - Operator == (equality)
  - Operator << (stream output)
  - sc_trace()
User-defined Data Types: An Example

```c++
// micro_bus.h
#include "systemc.h"
const int ADDR_WIDTH = 16;
const int DATA_WIDTH = 8;

struct micro_bus {
    sc_uint<ADDR_WIDTH> address;
    sc_uint<DATA_WIDTH> data;
    bool read, write;

    micro_bus& operator= (const micro_bus&);
    bool operator== (const micro_bus&) const;
}

// to be continued
```
User-defined Data Types: An Example

```cpp
inline micro_bus&
micro_bus::operator= (const micro_bus& arg) {
    address = arg.addresses;
    data = arg.data;
    read = arg.read;
    write = arg.write;
    return (*this);
}

inline bool
micro_bus::operator== (const micro_bus& arg) const {
    return (
        address == arg.address) && (data == arg.data) &&
        (read == arg.read) && (write == arg.write) );
}

// to be continued
```
User-defined Data Types: An Example

```cpp
inline ostream&
operator<< (ostream& os, const micro_bus& arg){
    os << "address=" << arg.address <<
        " data=" << arg.data << " read=" << arg.read <<
        " write=" << arg.write << endl;
    return os;
}

inline void sc_trace (sc_trace_file *tf,
    const micro_bus& arg, const sc_string& name) {
    sc_trace (tf, arg.address, name+".address");
    sc_trace (tf, arg.data, name+".data");
    sc_trace (tf, arg.read, name+".read");
    sc_trace (tf, arg.write, name+".write");
}

// how to declare two signals of this type
sc_signal<micro_bus> bus_a, bus_b;
```
Recommendations for SystemC RTL

- For one bit use `bool` data type
- For vectors and unsigned arithmetic use `sc_uint<n>`
- For signed arithmetic use `sc_int<n>`
- For big vectors use `sc_bigint<n>`, `sc_biguint<n>`
- Use `sc_logic` and `sc_lv<m>` types only for signals that will carry four logic values
- Use resolution types only when resolution is required such as when a port or a signal has multiple drivers
Outline

- Introduction
- Data Types
- Modeling Combinational Logic
- Modeling Synchronous Logic
- Misc