ELEC 2221 Digital Systems and Signal Processing

SystemVerilog

Dr. JingHuey Khor
Session Outline

- Data Type
- Array
- User Defined Data Types
- Enumerated Types
- Literals
- System Verilog Operators
- Procedural Blocks
- Blocking and Non-Blocking Assignments
## Basic Data Type

<table>
<thead>
<tr>
<th>2 Valued Data Type (0,1)</th>
<th>4 Valued Data Type (0,1,X,Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
</tr>
<tr>
<td>shortint</td>
<td>16 bits</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
</tr>
<tr>
<td>longint</td>
<td>64 bits</td>
</tr>
</tbody>
</table>
## Packed Array vs Unpacked Array

<table>
<thead>
<tr>
<th>Packed Array</th>
<th>Unpacked Array</th>
<th>Packed and Unpacked Array</th>
</tr>
</thead>
</table>

- **Guaranteed to be laid out contiguously in memory**
- **Can be copied onto any other packed object of same size**
- **Can be sliced, e.g. v[4:3]**
- **Restricted to bit, logic, int**

- **Can be arranged in memory as simulator chooses**
- **Only unpacked arrays of the same type can be copied**
- **Can be used with all data types**
User Defined Data Types

typedef bit[3:0] nibble;
nibble a,b;

// a and b are variables with nibble data types
Enumerated Types

• Provide a means to declare an abstract variable that can have a specific list of valid values.

• Each value is identified with a user-defined name, or label.

Example:

enum{red,green,blue} RGB;

Variable RGB have the values of red, green, and blue
Enumerated Type Values

- The actual value represented by the label in an enumerated type list is an int type.
- Example:

```c
enum{red,green,blue} RGB;
```

0 1 2
Enumerated Type Values

- `enum { one =1, five=5, ten=10 } state;`

  User can assign values to enum constants

- `enum { A=1, B, C, X=24, Y, Z } list1;`

- `enum { A=1, B, C, D=3} list2;`  
  
  Error!
Base Type of Enumerated Types (1)

- enum bit \{True, False \} Boolean;

**Enumerated type with a 1 bit wide, 2 state base type**

- enum logic [1:0] \{WAITE, LOAD, READY\} state;

**Enumerated type with a 2 bit wide, 4 state base type**

- enum logic [2:0] \{WAITE = 3’b001, LOAD=3’b010, READY = 3’b100\} state;

**Value size must match the size of the base type**

- enum \{WAITE = 3’b001, LOAD = 3’b010, READY = 3’b100\} state;  
  **Error!!**
Base Type of Enumerated Types (2)

• enum logic { A=1'b0, B, C} list5;

• enum logic {ON=1'b1, OFF=1'bz} out;

Legal to assign values of X or Z to the enumerated labels

• enum logic [1:0] { WAITE, ERR=2’bxx, LOAD, READY} state;

ERROR: cannot determine a value for LOAD
typedef enum { WAITE, LOAD,READY} states_t;

states_t state, next_state;

int foo;

state = next_state; // same type (states_t)

foo = state +1;

The enumerated type of state is represented as a base type of int, which is added to the literal integer 1.
state = foo +1;
state = state +1;
state++;
next_state += state;

It is error to directly assign int result to a variable of the enumerated type state, states_t type
Type Cast

type’(expression)

```
longint a,y;
real r;
y= a+longint’(r**3);
```

```
next_state=states_t’ (state++);
```
Literals

• General syntax: [size[‘base]]value
• Examples:
  1’bo – binary 0
  4’hF – hex base, binary equivalent: 1111
  10 – decimal base (default)
  ‘o6 – octal base, binary equivalent: 110

size is the number of bits, default is 32 bits
`base` represents the radix, default is decimal

<table>
<thead>
<tr>
<th>Base</th>
<th>Symbol</th>
<th>Legal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>b,B</td>
<td>0,1,x,X,z,Z,?,_</td>
</tr>
<tr>
<td>octal</td>
<td>o,O</td>
<td>0-7, x,X,z,Z,?,_</td>
</tr>
<tr>
<td>decimal</td>
<td>d,D</td>
<td>0-9, x,X,z,Z,?,_</td>
</tr>
<tr>
<td>hexadecimal</td>
<td>h,H</td>
<td>0-9,a-f,A-F,x,X,z,Z,?,_</td>
</tr>
</tbody>
</table>

? is the same as z or Z
_ (underscore) is ignored, it is used to enhance legibility of long literals
SystemVerilog Operators

Operator groups

- arithmetic
- logical
- relational
- equality
- reduction (bitwise logic)
- shift
# Arithmetic Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (remainder)</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
</tr>
</tbody>
</table>
Logical Operators

<table>
<thead>
<tr>
<th>!</th>
<th>Negation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Uses are similar to those in C
## Relational and Equality Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
</tbody>
</table>

**Equality operators:**

1. `==` and `!=` result in `X` if any of their operands contains an `X` or `Z`, e.g. `X == 1 -> X`.
2. `===` and `!==` check the 4-state logic explicitly therefore, `X` and `Z` values shall either match or mismatch, never resulting in `X`, e.g. `X === 1 -> 0` (a mismatch).
3. `==?` and `!=?` treat `X` or `Z` as wild cards that match any value, thus, they too never result in `X`, e.g. `X ==? 1 -> 1` (a match).
# Reduction Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~</code></td>
<td>bitwise negation (binary complement)</td>
</tr>
<tr>
<td><code>~&amp;</code></td>
<td>bitwise NAND</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>`~</td>
<td>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td>bitwise XOR</td>
</tr>
<tr>
<td><code>~^</code></td>
<td>bitwise XNOR</td>
</tr>
</tbody>
</table>
Shift Operators

| >>   | logical right shift |
| <<   | logical left shift  |
| >>>  | arithmetic right shift |
| <<<  | arithmetic left shift |

Uses are similar to those in C which has the logical shifts; distinction between logical and arithmetic shifts was first introduced in Verilog-2001
Other Operators

- Concatenation
  
  `{}` e.g. `4'b1001, 2'b11} = 6'b100111`

- Replication
  
  `{n{m}}` - replicate m for n times, e.g. `{8{1'b0}} = 8'b00000000`

- Conditional
  
  `? : e.g. (c==1'b1 ? a : 1'bZ)`

- Assignment
  
  `=, +=, -=, *=, /=, %=, &=, ^=, <<=, >>=, <<<, >>>=`

- Increment/decrement
  
  `++,-` use with caution, `x++` is equivalent to: `x = x+1` not: `x <= x+1`!

  in sequential logic counters are implemented using: `cnt <= cnt+1`
Concatenation and Replication Examples

```plaintext
logic a,b;
{a,b} = 2'b11;
{a,b} = {1'b0, 1'b1}; // same as {a,b} = 2'b10;

string sv = "Hello";
string s;
s = {sv, " ", "world"};

$display("%s\n", s); // displays ‘Hello world’

logic [7:0] number8 = 'h80;
logic [15:0] number16;
logic signBit = number8[7];

assign number16 = { {8{signBit}}, number8}; // manual sign extension from 8 to 16 bits
```
Procedural Blocks (Processes)

• initial // executes only once at the beginning of simulation

• final // executes once at the end of simulation

• always, always_comb, always_latch, always_ff
  // these blocks execute continuously when triggered by events

• task // executes when called, similar to function with no return value

• function // executes when called and returns a value
Example 1

always_ff infers edge triggered flip-flops in synthesis

```
always_ff @ (posedge clk) // process code executes when clk is rising
begin
    q <= d ; // q changes when clk is rising
end
```
Example 2

Register – an array of D-type flip-flops

module dffreg # (parameter n = 8) ( // 8 flip-flops
output logic [n-1:0] q, // array from n-1 down to 0
input logic [n-1:0] d,
input logic clk, nreset );

always_ff @ (posedge clk or negedge nreset)
if(~nreset) // reset here is active low
  q <= {n{1'b0}} ; // {n{}} means replicate n times and concatenate
else
  q <= d;
endmodule
Example 3

D latch with enable

```haskell
// always_latch synthesises into transparent latch
always_latch // note: sensitivity inferred from code
    if (en)
        begin
            q = d;
            qbar = ~ d;
        end
```

```haskell
// If no qbar pin available on latch, e.g. MachXO
always_latch
    if (en)
        q = d;

assign qbar = ~q;
```
Example 4

always_comb // sensitivity inferred from code
begin
    sum = cin ^ a ^ b;
    cout = a & b | a & cin | b & cin;
end

// alternative implementation:
// an assign statement is an always_comb block in its own right
assign sum = cin ^ a ^ b;
assign cout = a & b | a & cin | b & cin;
• A blocking assignment updates its target immediately

```vhdl
always_ff @ (posedge clock)
begin
    a = 1'b0; // assign 0 to a immediately
    b = 1'b0; // assign 0 to b immediately
    #10ns // after 10 ns
    a = ~a; // assign 1 to a immediately
    b = a; // b becomes 1 immediately
end
```

• A non-blocking assignment evaluates, but its target is updated after all active processes execute repeatedly and stop when there are no more events

```vhdl
always_ff @ (posedge clock)
begin
    a <= 1'b0; // assign 0 to a on completion of this time point
    b <= 1'b0; // assign 0 to b on completion of this time point
    #10ns // after 10 ns
    a <= ~a; // a will become 1 on completion of this time point
    b <= a; // b will become 0 (!) on completion of this time point
end
```
# Blocking and Non-blocking Assignments

<table>
<thead>
<tr>
<th>Blocking: Evaluation and assignment are immediate</th>
<th>[ a = b ]</th>
<th>[ x = a \land b ]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ b = a ]</td>
<td>[ y = x \lor c ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Blocking: Assignment is postponed until all r.h.s. evaluations are done</th>
<th>[ a \leq b ]</th>
<th>[ x \leq a \land b ]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ b \leq a ]</td>
<td>[ y \leq x \lor c ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When to use: (only in always blocks!)</th>
<th>Sequential Circuits</th>
<th>Combinational Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assignment Styles for Sequential Logic

Flip-Flop Based Digital Delay Line

- Will nonblocking and blocking assignments both produce the desired result?

module nonblocking(in, clk, out);
    input in, clk;
    output out;
    reg q1, q2, out;
    always @ (posedge clk)
    begin
        q1 <= in;
        q2 <= q1;
        out <= q2;
    end
endmodule

module blocking(in, clk, out);
    input in, clk;
    output out;
    reg q1, q2, out;
    always @ (posedge clk)
    begin
        q1 = in;
        q2 = q1;
        out = q2;
    end
endmodule
Assignment Styles for Sequential Logic

always @ (posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end

“At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2.”

- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic

- Guideline: use nonblocking assignments for sequential always blocks
Assignment Styles for Sequential Logic

```verilog
reg qa, qb, qc;
always @(posedge clk)
begin
    qa <= a;
    qb <= qa;
    qc <= qb;
end
```

```verilog
reg qa, qb, qc;
always @(posedge clk)
begin
    qc <= qb;
    qb <= qa;
    qa <= a;
end
```

```verilog
reg qa, qb, qc;
always @(posedge clk) qa <= a;
always @(posedge clk) qb <= qa;
always @(posedge clk) qb <= qa;
```

---

**DFF Diagram**

- **a**
- **D Q**
- **clk**
- **qa**
- **qb**
- **qc**

**Timing Diagram**

- **clk**
- **a**
- **qa**
- **qb**
- **qc**

---

**Wire Diagram**

```verilog
reg qa, qb, qc;
// synthesizes to a wire
always @*
begin
    qa = a;
    qb = qa;
    qc = qb;
end
```

- **a**
- **qa**
- **qb**
- **qc**