NCDC

NUST CHIP DESIGN CENTRE

Digital Logic Design

SystemVerilog in First Glance

SystemVerilog Design Modules

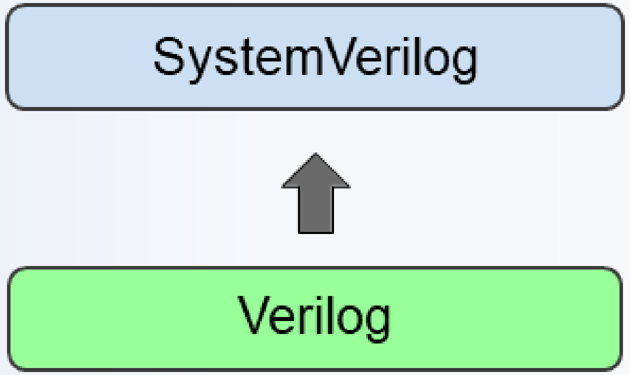
SystemVerilog Fundamentals for Design

SystemVerilog is the most commonly used HDL.

- A substantial upgrade on Verilog...
- ... which has a long history dating back to 1985

SystemVerilog contains:

- A huge number of constructs.
- Wide variety of options, alternative syntax, etc.
- Many redundant features (e.g., transistor level modeling).



This course *only* covers the fundamentals for design.

- Specifically, the most useful and frequently used features.
- Best design practices where there are options.

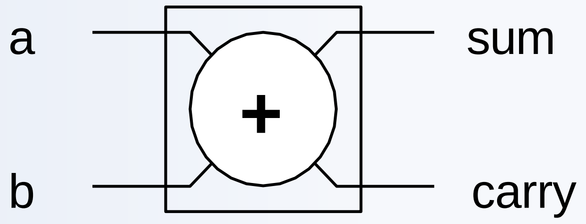
Different level of modeling

- Behavioral
- Data flow
- Gate level
- Switch level

Describing Design Modules

Start with the **module** keyword and identifier. Define the module port list.

- Various syntax options.
- Simplest is ANSI-C format:
  - <direction> <type> <identifier(s)>



Describe the module behavior.

- assign outputs from an expression of inputs.
- End with the **endmodule** keyword. Save in a file with the .sv extension.

Note

- Identifiers are case-sensitive.
- Keywords are always lowercase.

halfadd.sv

```
module halfadd (
  input logic a, b,
  output logic sum, carry);

  assign sum = a ^ b;
  assign carry = a & b;

endmodule
```

Rules for Naming Identifiers

Identifiers start with a letter or an underscore (_).

- Followed by letters, digits, `$` or _

SystemVerilog does not restrict name length.

- Although tools or methodologies might.

Identifiers are case-sensitive.

- `ABC`, `Abc`, `abc` are all different legal names.

All keywords are lowercase.

Convention

- Write everything in lowercase.
  - Exception: certain user-defined type values.

Escaped identifiers allow illegal names.

- Rarely used.

Legal	`unit_32`
	`bus_16_bits`
	`abc$`

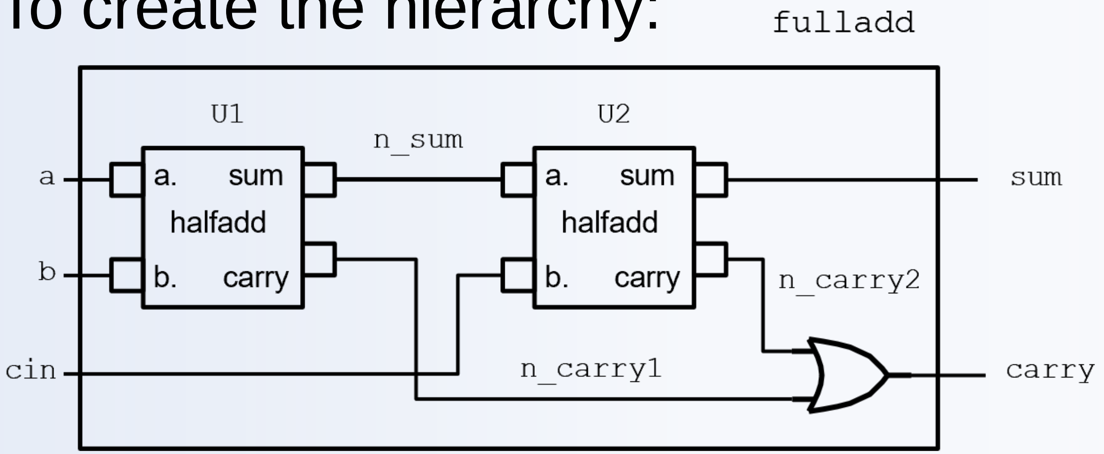
Not Legal	`unit-32`
	`16_bit_bus`
	`$abc`

Escaped	`\unit-32`
	`\16_bit_bus`
	`\$abc`

Representing Hierarchy

A full adder is 2 half adders + or operator. To create the hierarchy:

- Declare local variables.
- Instantiate module(s):
  - Give each instance a unique name.
- Connect instance ports to local ports and variables.
  - `.port(variable)`




```
module fulladd (input logic a, b, cin,
                output logic sum, carry);

  logic n_sum, n_carry1, n_carry2;

  halfadd U1  ( .a(a), .b(b), .sum(n_sum), .carry(n_carry1) );
  halfadd U2  ( .a(n_sum), .b(cin), .sum(sum), .carry(n_carry2) );

  assign carry = n_carry1 | n_carry2;

endmodule
```

Local variables

Module

Instance name

Port mapping

OR operator

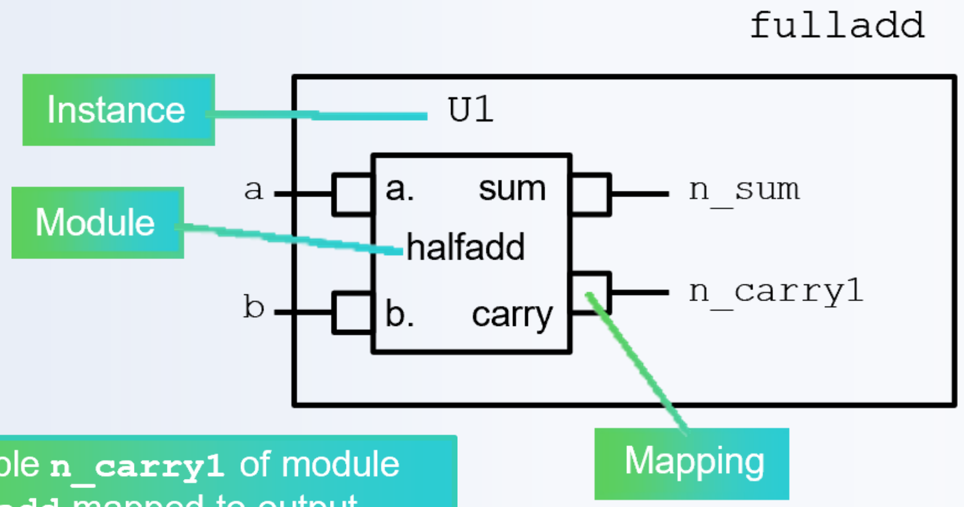
Connecting Hierarchy: Ordered Port Connection

Port mapping can be made by position.

- Local port, variable mapped in order of port declaration.

Very easy to map in the wrong order.

- Not recommended.




```
module fulladd (input logic a, b, cin,
                output logic sum, carry);

  logic n_sum, n_carry1, n_carry2;

  halfadd U1 ( a, b, n_sum, n_carry1 );
  halfadd U2 ( n_sum, cin, sum, n_carry2 );
...
```

Variable **n_carry1** of module **fulladd** mapped to output **carry** of instance **U1** of module **halfadd**

```
module halfadd (input logic a, b,
                output logic sum, carry)
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

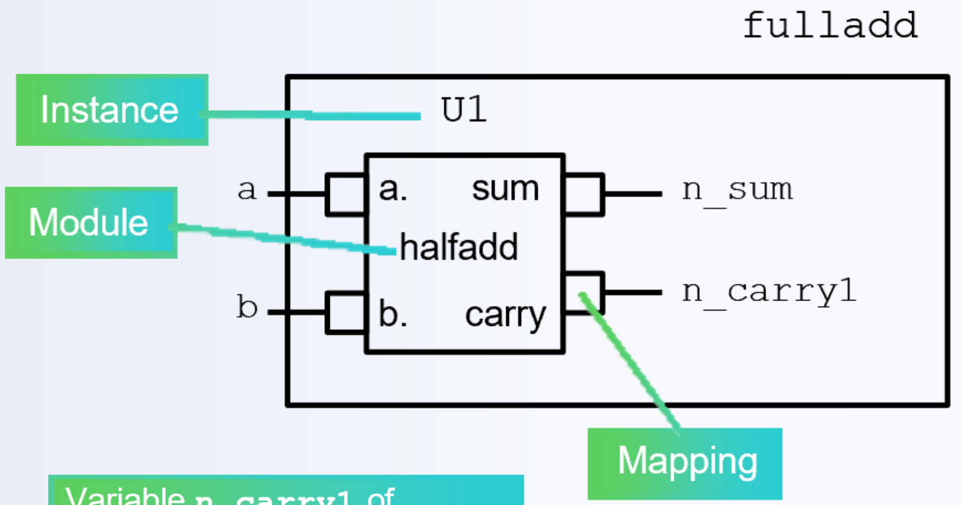
Connecting Hierarchy: Named Port Connection

Named port connection is much safer.

- `.port(variable)`

Where port and variable names match, there is a shortcut.

- `.sum = .sum(sum)`




```
module fulladd (input logic a, b, cin,
                output logic sum, carry);

  logic n_sum, n_carry1, n_carry2;

  halfadd U1 ( .a, .b, .sum(n_sum), .carry(n_carry1) );
  halfadd U2 ( .a(n_sum), .b(cin), .sum, .carry(n_carry2) );
...
```

Variable **`n_carry1`** of module **`fulladd`** mapped to output **`carry`** of instance **`U1`** of module **`halfadd`**

```
module halfadd (input logic a, b,
                output logic sum, carry)
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

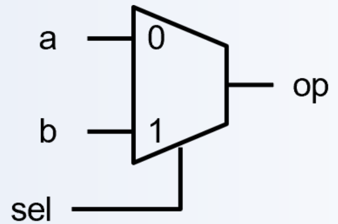
Procedural Blocks

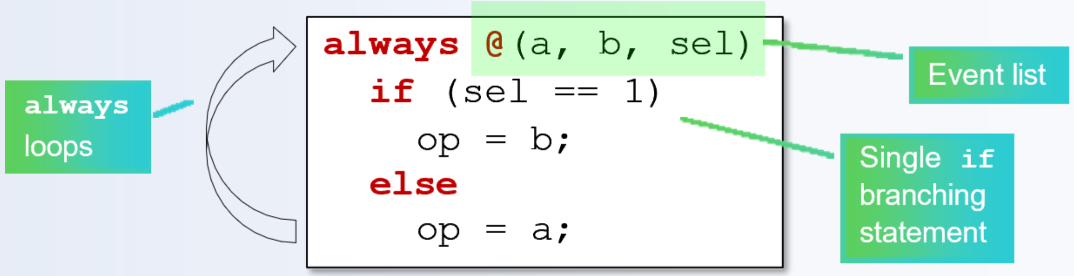
Procedural blocks define complex behavior.

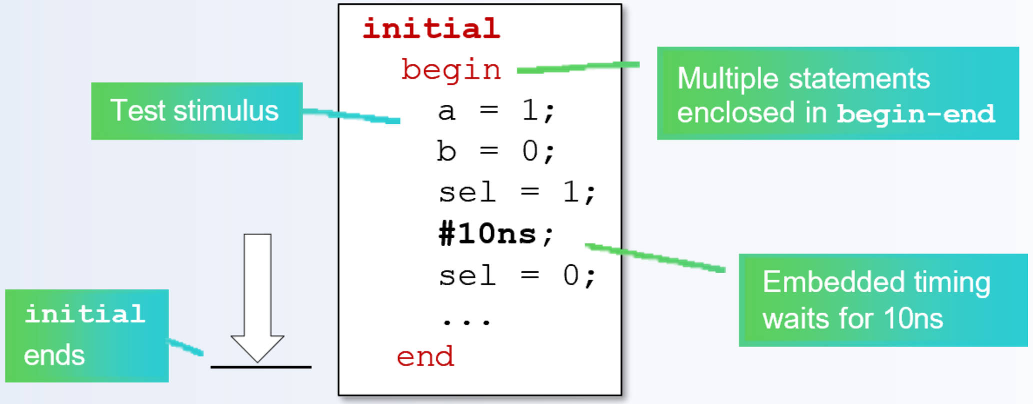
- For example, conditional and repetitive combinational logic.
- Registered logic.

Two general procedural blocks:

- `always`
  - Executes at the start of simulation.
  - When at end, loops back to beginning.
  - Further execution controlled by the event list.
- `Initial`
  - Executes at start of simulation.
  - Embedded timing pauses execution.
  - When at end, terminates.
  - Testbench construct.
- Multiple procedural statements need `begin`...`end`.







Synchronizing Block Execution

The @ event expression controls block execution.

Change in value of any variable in expression triggers block.

Incorrect event expressions are an issue.

- For example, in combinational logic, event list must be *complete.*
  - Must contain all variables read in block.
- Otherwise, RTL and gate-level behavior differ.

Expressions can be edge-sensitive.

- Trigger on a specific transition.
- Use `posedge`, `negedge` or (rarely) `edge`.
- Essential for sequential (register) logic.

```
always @(a, b, sel)
  if (sel == 1)
    op = b;
  else
    op = a;
```

Change in value of `a`, `b` or `sel` triggers block

```
always @(a, b)
  if (sel == 1)
    op = b;
  else
    op = a;
```

Incomplete event list

```
always @(posedge clock)
  d <= q;
```

Positive-edge triggered

Different assignment (see later)

Specialist RTL Procedural Blocks

RTL code uses special always blocks

- Do **not** use `initial` or `always`

`always_comb`

- Combinational logic
- Implicit, complete event list
- Uses `=` assignment

`always_ff`

- Registered logic
- Requires an edge-triggered event list
  - Clock and reset signals only
- Uses `<=` assignment

Detailed descriptions later

```
always_comb
  if (sel == 1)
    op = b;
  else
    op = a;
```

All variables read in block automatically added to event list

```
always_ff @(posedge clock)
  d <= q;
```

Positive-edge triggered

Different assignment (see later)

Rules for Comments and White Space

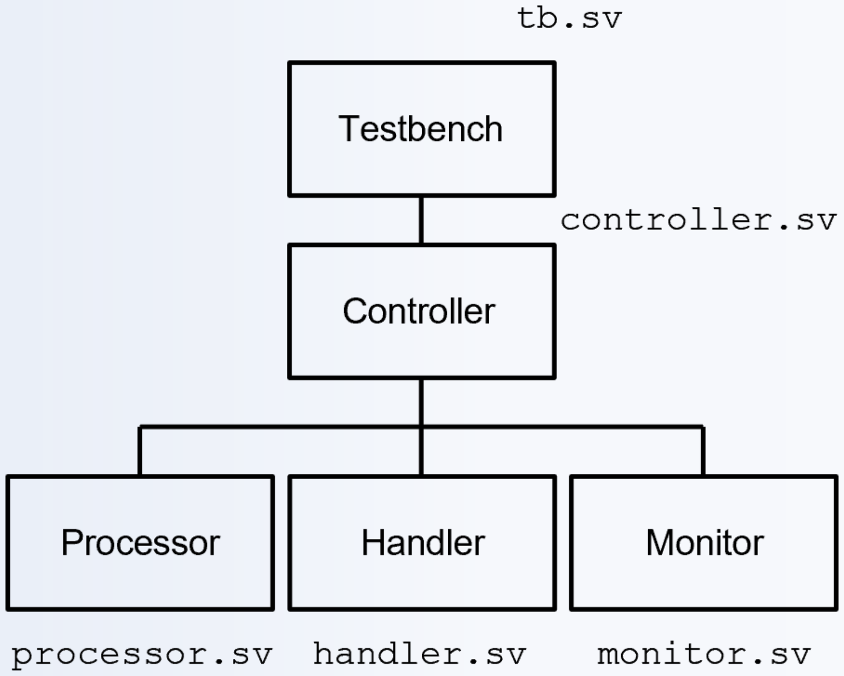
```
// A one-line comment starts with // and ends with newline character
/* A block comment starts anywhere with /*
 and ends anywhere with */
module muxadd (
 input logic a, b, sel,    // module inputs
 output logic sum, carry, y);

// SystemVerilog is a free-format language
// White space is needed only to separate some language tokens
// Use additional white space to enhance readability
assign sum   = a ^ b;
assign carry = a & b;

// Also use indentation (2 space is best) to enhance readability
always @(a, b, sel)
 if (sel == 1) y = b;
 else
  y = a;
...
```

Comments should be meaningful, not stating the obvious

Compiling the Design

Module compilation order is generally not important.

- One common exception is packages.
  - Contains declarations shared over multiple modules.
  - Must be compiled before modules that use the package.
  - More information on packages later.




```
xrun -f run.f
```

run.f

```
// controller test
tb.sv
controller.sv
processor.sv
handler.sv
monitor.sv
```

Standard SystemVerilog Types

Value Sets

4-state `logic` variables initialize to `X` at the start of simulation.

- Helps detect initialization/reset issues.
  - Variables that remain at `X` have not been reset correctly.

2-state `bit` variables initialize to `0`.

- Hides initialization issues.
- Used for RTL *only* in very limited situations.

Value	Associated Informal Terms
0	Zero, Low, False
1	One, High, True
Z	High Impedance, Tri-State, Undriven,
X	Uninitialized, Unknown (bus contention)

`logic` (values 0, 1, Z, X)

`bit` (values 0, 1)

Limited use in RTL

RTL Data Types

SystemVerilog provides three “data types” for RTL design.

- Variables
  - `var`
  - General purpose RTL use.
- Nets
  - E.g., `wire`
  - Include resolution tables to resolve multiple drivers.
  - Used for multiply-driven connections *only.*
    - E.g., tri-states and bidirectionals.
- Constants
  - True constants (`localparam`).
  - Instance-specific constants (`parameter`).
    - Can be overridden for an instance basis giving greater flexibility.

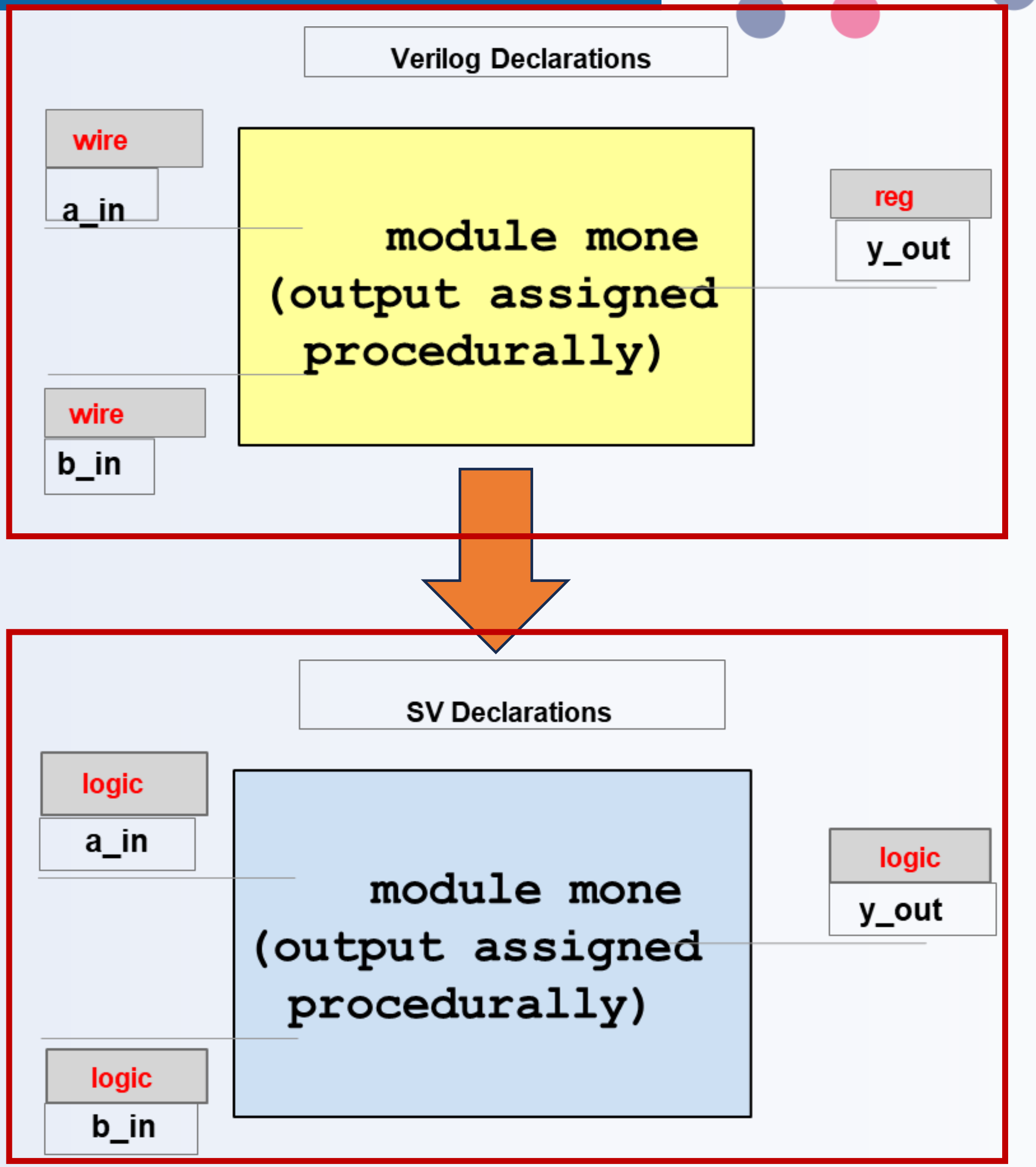
Quick Reference Guide: Verilog Data Type Rules

- Verilog has strict data type rules:
  - Variables (registers) are assigned values in procedural blocks.
  - Nets are driven by continuous assignments, module inputs, module instance outputs, or primitive instances.
- These lead to the following connectivity characteristics:
  - Module inputs are always nets.
  - Module outputs are variables if driven by a procedural block, or nets in all other cases.
  - Connections to the input ports of a module instance are variables if driven by a procedural block, or nets in all other cases.
  - Connections to the output ports of a module instance are always nets.
  - Connections to bidirectional `inout` ports are always nets.

Relaxation of Datatype Rules

SystemVerilog relaxes the rules for a variable by:

- Assigning a SystemVerilog variable:
  - In any number of `initial` or `always` blocks:
    - As in Verilog currently
  - From a single continuous assignment
  - From a single module `out` port
  - From a single primitive output
- Thus, you can declare most design signals to be variable.
  - As the `var` keyword is optional, you can simply declare most signals as type `logic`:
    - `logic my_var;`



Verilog Data Type Rules: Example

```
module mone (output reg y_out,
             input wire a_in, b_in);
 always @(a_in or b_in)
    y_out = a_in && b_in;
endmodule
```

```
module mtwo (output wire y_out,
             input wire a_in, b_in);
  assign y_out = a_in || b_in;
endmodule
```

```
module test;
  reg a, b;
  wire and_out, or_out;
  mone u1 (and_out, a, b);
  mtwo u2 (or_out, a, b);
  initial begin
      a = 0;
      b = 0;
      ...
  end
endmodule
```

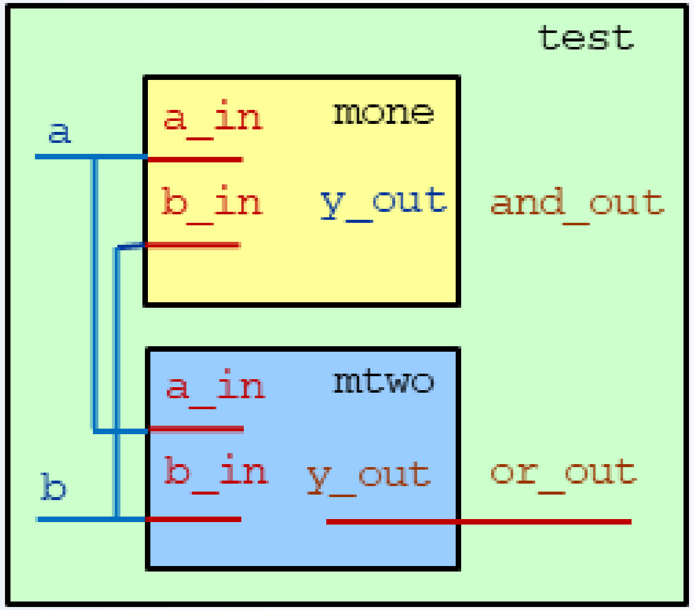
`y_out` is `reg` in `mone` – assigned in a procedural block

`a_in`, `b_in` are `wire` in `mone` and `mtwo` – module inputs

`y_out` is `wire` in `mtwo` – assigned by a continuous assignment

`and_out`, `or_out` are `wire` in module `test` – instance outputs

`a` , `b` are `reg` in module `test` – assigned in an `initial` procedural block



SV Relaxed Data Type Rules: Example

```
module mone(output logic  y_out,
            input logic a_in, b_in);
 always @(a_in or b_in)
    y_out = a_in && b_in;
endmodule
```

y_out is logic in mone – assigned in a procedural block

a_in, b_in are logic in mone and mtwo – connected to single module inputs

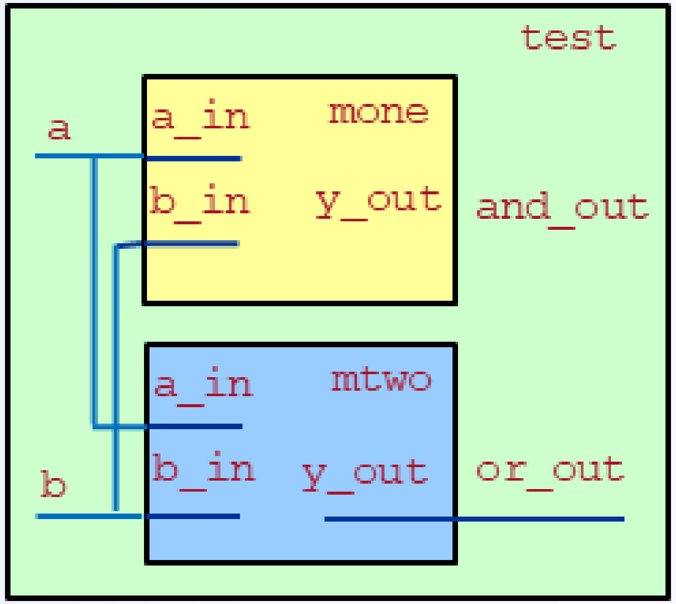
```
module mtwo(output logic y_out,
            input logic a_in, b_in);
  assign y_out = a_in || b_in;
endmodule
```

y_out is logic in mtwo – assigned by a single continuous assignment

```
module test;
  logic a, b;
  logic and_out, or_out;
  mone u1 (and_out, a, b);
  mtwo u2 (or_out, a, b);
  initial begin
     a = 0;
     b = 0;
     ...
  end
endmodule
```

and_out, or_out are logic in module test – single instance outputs

a , b are logic in module test – assigned in an initial procedural block



How to Overcome Variable Assignment Restrictions

Although SystemVerilog relaxes the rules on data type declaration, there are still some restrictions on assignments to variables. These restrictions aim to prevent multiple drivers on a single variable.

- You cannot combine procedural assignments with continuous assignments or module output drivers on the same variable.
- You cannot have multiple continuous assignments or multiple output ports drive the same variable.
- Only net types can have multiple drivers.

```
module test;
  logic in1, in2, in3;
  logic op;
  mone u1 (op, in1, in2);
  mtwo u2 (op, in1, in3);
endmodule
```

Error

Multiple drivers on `logic op`

```
module test;
  logic in1, in2, in3;
  wire op;
  mone u1 (op, in1, in2);
  mtwo u2 (op, in1, in3);
endmodule
```

Variable Rules

All `logic` types are variables by default.

- The variable keyword `var` is rarely required. Variables can

only have a single driver. *One* of the following:

- An `assign` statement.
- A module `output` port.
  - In an instantiation port map.
- An `always_comb` block.
- An `always_ff` block.

Multiple drivers on a variable give compilation errors.

- If you *need* multiple drivers, declare a net type.
  - E.g., `wire`

```
logic var1;
```

Equivalent

```
var logic var1;
```

Multiple drivers on **logic op** illegal

```
module test;
  logic in1, in2, in3;
  logic op;
  mone u1 (op, in1, in2);
  mtwo u2 (op, in1, in3);
endmodule
```

✗ Error

wire op for multiple drivers

```
module test;
  logic in1, in2, in3;
  wire op;
  mone u1 (op, in1, in2);
  mtwo u2 (op, in1, in3);
endmodule
```

☑

Assignments to a Variable from Multiple Procedures

Special procedural blocks (`always_ff`, `always_comb`, `always_latch`) allow only a single driver to variables.

```
logic op;

always_comb
  if (sel)
    op = a;
  else
    op = b;

 always_comb
   if (sel2)
    op = c;
  else
    op = d;
```

Error

`always_comb` allows only a single driver on `logic op`.

Net Data Types

Nets are used where multiple drivers are required.

- E.g., tristate, bidirectional.
- Always 4-state.

Include resolution tables to resolve multiple drivers.

Many different net types.

- `wire` is the only one in common use.

Net types can only be driven by:

- `assign` statements.
- Module `output` or `inout` ports.

Cannot be driven from a procedural block.

- `initial` or *any* form of always.

Multiple drivers require **`wire`**

```
logic ena1, ena2, data1, data2;
wire dataout;

// if ena1 true, drive data1 else drive Z
assign dataout = ena1 ? data1 : 1'bz;
assign dataout = ena2 ? data2 : 1'bz;
```

`assign if` statement

Output port driver OK

Assign OK

Net cannot be driven from procedural block

```
wire op;

mone u1 (op, in1, in2);  ☑

assign op = a ^ b;  ☑

always_comb
  op = a & b;
```

Error

Net Data Type Resolution

Only a net can resolve the value of multiple drivers

`wire` resolution:

- Simultaneous drive of 0 and 1 results in unknown (X).
- Simultaneous drive of 0 and Z results in 0.
  - For modelling tri-states.

Other resolutions exist, but rarely used.

- E.g., `wand` models wired-AND logic.

data1 \ data2	0	1	z	x
0	0	0	0	0
1	0	1	1	x
z	0	1	z	x
x	0	x	x	x

`wand` resolution

data1 \ data2	0	1	z	x
0	0	x	0	x
1	x	1	1	x
z	0	1	z	x
x	x	x	x	x

`wire` resolution

Default and Implicit Types

Declarations without a data type default to `wire`.

- `wire` allows multiple drivers.
- Lose the "single driver" compilation check.

You should fully declare all variables.

```
module halfadd (
  input logic a, b,
  output sum, carry);
...
```

Variables

No data type – **`sum`** and **`carry`** default to **`wire`**

In some constructs, undeclared identifiers are allowed.

- Called implicit declarations.
- Declared as a single bit `wire`.

Check identifiers carefully.

- Typos can lead to broken connections.

```
module fulladd (input logic a, b, cin,
                output logic sum, carry);

  logic n_sum, n_carry1;

  halfadd U1 ( .a(a), .b(b),
               .sum(n_sum),
               .carry(carry1) );
...
```

Typo in variable name implicitly declares **`wire`**

Declaring Vectors

A vector is declared an array.

- Collection of individual bits.
- Defined with a range specification.
- Indexed with an integer.

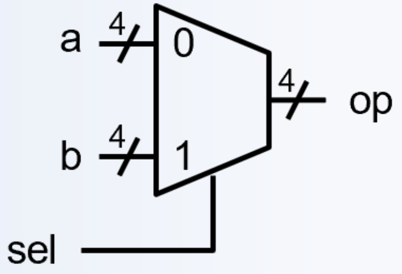
Define the range when declaring the variable:

- Bounds can be descending or ascending.
  - [msb : lsb] or [lsb : msb]
- Bounds can be negative, zero, or positive.
- Bounds can be expressions...
  - ...but must be constant and known at start of simulation.

By convention, bounds are descending to 0.

- E.g., `[3:0]`

```
module mux4 (
  input  logic [3:0] a, b,
  input  logic       sel,
  output logic [3:0] op
);

always_comb
  if (sel == 1)
    op = b;
  else
    op = a;

endmodule
```



Using Vector Ranges

A vector can be sliced with a range of one or more contiguous bits.

- Without a range, the whole vector is selected.

Elements are assigned in order of declaration.

- Unselected elements are unchanged.

Slice must be in the same direction as the declaration.

```
logic [3:0] idec;
logic [0:3] oasc;

assign oreg = ireg;
// osc[0] = idec[3]
// osc[1] = idec[2]
// osc[2] = idec[1]
// osc[3] = idec[0]

assign oreg[2:0] = ireg[2:0];
```

Assigned in order of declaration

Error

Cannot slice ascending vector with descending bounds

```
logic [3:0] inp;
logic [3:0] outp;

assign outp = inp;
// outp[3] = inp[3]
// outp[2] = inp[2]
// outp[1] = inp[1]
// outp[0] = inp[0]

assign outp[3] = inp[0];
// outp[3] = inp[0]

assign outp[3:0] = inp[1:0];
// outp[3] = inp[1]
// outp[2] = inp[0]
```

Without range, whole vector selected

Assigned in order of declaration

Individual element

Slice

Assigning Between Different Widths

Vector widths do not need to match in an assignment!

- If the source is wider than the target, the value truncated is from the left-most bit.
- If the unsigned source is shorter than the target, the value zero-extended is from the left-most bit.
- If the signed source is shorter than the target, the value is sign-extended.
  - Selections and concatenations are not considered signed.

```
logic [3:0] zbus;      // 4 bits
logic [5:0] widebus;  // 6 bits

always_comb
 zbus = widebus;     // same as
 //           <- widebus[5]
 //           <- widebus[4]
 // zbus[3] <- widebus[3]
 // zbus[2] <- widebus[2]
 // zbus[1] <- widebus[1]
 // zbus[0] <- widebus[0]
```

```
logic [3:0] zbus;      // 4 bits
logic [5:0] widebus;  // 6 bits

always_comb
  widebus = zbus; // same as
  // widebus[5] <- 0
  // widebus[4] <- 0
  // widebus[3] <- zbus[3]
  // widebus[2] <- zbus[2]
  // widebus[1] <- zbus[1]
  // widebus[0] <- zbus[0]
```

Array Types and Dimensions

SystemVerilog allows arrays:

- Of any type.
- With any number of dimensions.

Multi-dimensional arrays in RTL are used for:

- Modeling memories.
- Declaring register arrays.

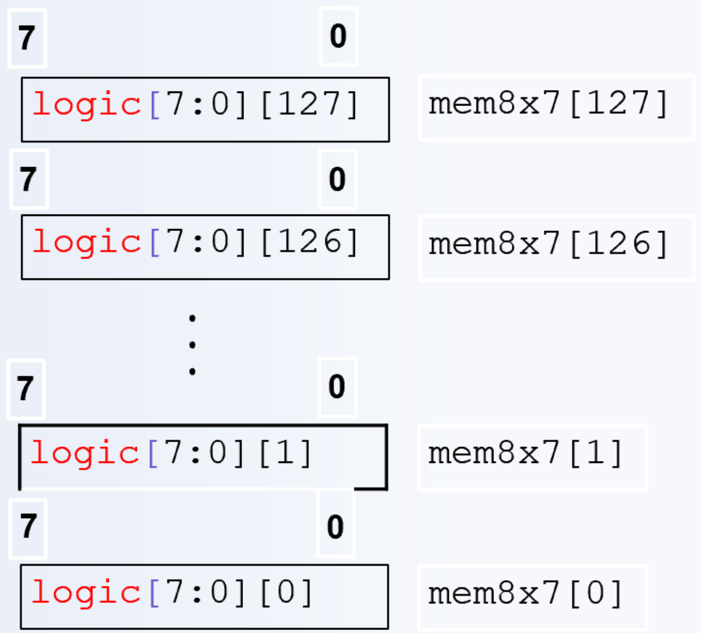
Therefore, the typical RTL arrays are 2D.

- Address index declared after name.
- Each array element is stored as a separate variable.
- Indexing priority is address first, then data.

Indexing order

2 1

```
logic[7:0] mem8x7 [127:0];

// mem8          = array of bytes
// mem8x7[0]     = 0th byte
// mem8x7[0][0]  = 0th bit of 0th byte
```



Defining Literal Values

You can specify a literal value as:

`<size>'<base><value>`

- `size` is an optional positive decimal number of bits.
  - If omitted, is *at least* 32 bits.
- `base` is a character to indicate binary, octal, decimal, or hexadecimal radix.
  - B/b, O/o, D/d, H/h.
  - If omitted, defaults to decimal.
- `value` is legal digits for base.
  - Can include underscores "_" if not the first character.
    - Non-consecutive.
  - Can include Z/z and X/x digits if the base is binary, octal, or hexadecimal.

```
...
logic [3:0] abus;
...
abus = 4'b1001; // 1001
abus = 4'd14;   // 1110
abus = 4'h2f;   // 1111
...
```

More Examples

```
8'b1100_0001    8-bit binary
10'd1000        10-bit decimal
16'hff01        16-bit hexadecimal
12              32-bit decimal
'h83a           32-bit hexadecimal
```

Literal Assignment Rules (Unsigned)

Size and value of a literal need not match the target.

Value is extended or truncated to literal size.

- Extended with 0 if leftmost bit is 0 or 1.
- Extended with leftmost bit if Z or X.

Value then further extended or truncated to the target size.

- According to normal rules.

Mismatches can lead to unexpected values.

Avoid zero-extension using unsized literals.

- If size omitted, literal is sized to target.
- If base also omitted, value fills entire vector.

```
logic [5:0] databus;

// zero value extension
databus = 6'b0;    // 000000
databus = 6'b1;    // 000001

// leftmost bit value extension
databus = 6'bz;    // zzzzzz
databus = 6'bx;    // xxxxxx

// value truncation by literal size
databus = 6'hff;   // 111111

// under-sized value zero-extended
databus = 4'bx;    // 00xxxx

// unsized value extended to target
databus = 'bx;     // xxxxxx

// unsized, unbased value
databus = '1;      // 111111
```

Declaring Module Parameters

A module parameter is an instance-specific constant.

- Parameterizes the module definition.
  - Width, depth, etc.
- Has a default value in declaration.
- Type is derived from value.
- Can be overridden for each individual instance.
  - Using a named parameter override.
  - Could also use positional override.

Generic MUX default width 2

```
module mux
  #(parameter WIDTH = 2)
  (input  logic [WIDTH-1:0] a, b,
   input  logic              sel,
   output logic [WIDTH-1:0] op);

always_comb
  if (sel)
    op = a;
  else
    op = b;
endmodule
```

Allows generic modules.

- Scaled on instantiation.

Tip: use uppercase identifier for parameter.

- Avoids clash with local identifiers.
- Helps readability.

```
logic [1:0] a2, b2, op2;
logic [3:0] a4, b4, op4;
logic sel;

mux mux2 (.a(a2),.b(b2),.sel,.op(op2));

mux #(.WIDTH(4)) mux4 (.a(a4),.b(b4),.sel,.op(op4));
```

Default width 2

Named override and sets width to 4

Local Parameters

A localparam is a true constant.

- Unlike parameter, localparam cannot be *directly* overridden hierarchically.
  - Although, it can be derived from parameter values.

```
localparam DWIDTH = 16;
localparam AWIDTH = 6;
localparam MSB = 7;
localparam MEMSIZE = DWIDTH * (1 << AWIDTH);
```

Use for:

- Constants that should not change.
- Naming literals or expressions.
- Deriving values from parameters.
  - Where these values should not be overridden.

Can also be declared in parameter list of module.

- For use in port list.

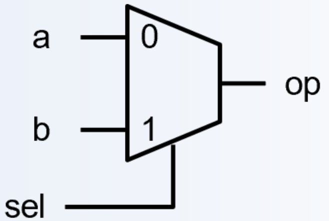
```
module us_mult
  #(parameter WIDTH_A = 4, WIDTH_B = 4,
    localparam WIDTH_OP = WIDTH_A + WIDTH_B)
  (input  logic [WIDTH_A-1:0] a,
   input  logic [WIDTH_B-1:0] b,
   output logic [WIDTH_OP-1:0] op);

  assign op = a * b;

endmodule
```

Making Procedural Statements

Overview: Procedural Blocks

Discussed in previous slide.

- `Always`
  - Executes at start of simulation.
  - When at end, loops back to the beginning.
  - Further execution controlled by the event list.
  - Change in value of any variable in list
  - triggers block.
- `initial`
  - Executes at start of simulation.
  - Embedded timing pauses execution.
  - When at end, terminates.
- Multiple procedural statements need `begin`...`end`.
- Procedural blocks cannot be assigned to nets (`wire`).




```
always @(a, b, sel)
  if (sel == 1)
    op = b;
  else
    op = a;
```

Event list

always loops

```
initial
  begin
    a = 1;
    b = 0;
    sel = 1;
    #10ns;
    sel = 0;
    ...
  end
```

Test stimulus

Multiple statements enclosed in `begin-end`

Pause execution or 10ns

initial ends

Specialist RTL Procedural Blocks

RTL code only uses special-always blocks.

`always_comb`

- Combinational logic
- Implicit, complete event list
- Uses = assignment

`always_ff`

- Registered logic
- Requires an edge-triggered event list
  - Clock and reset signals only
- Uses <= assignment

Variables driven by these blocks cannot be driven by anything else.

```
always_comb
  if (sel == 1)
    op = b;
  else
    op = a;
```

All variables read in block automatically added to event list

```
always_ff @(posedge clock)
  d <= q;
```

Positive-edge triggered

Different assignment (see later)

Making Case Statements: case

```
case (expression)
 item { , item } : statement(s)
  item { , item } : statement(s)
  default : statement(s)
endcase
```

Checks an expression against a series of matches.

- Executes statement(s) associated with first match.
- Multiple statements in a branch enclosed in `begin`...`end`.

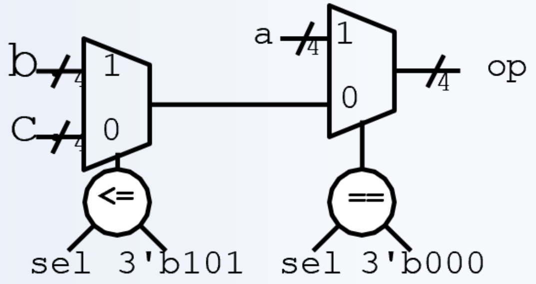
Case item match expressions can overlap.

- Compares matches to case expression in the order they appear.
  - Using case equality comparison (===).

Optional `default` item executed if no other matches.

- In this example, inputs containing Z or X.

```
logic [3:0] a, b, c;
logic [2:0] sel;
logic [3:0] op;

always_comb
  case (sel)
    0         : op = a;
    1,2,3,4,5 : op = b;
    6,7       : op = c;
    default   : op = 'b0;
  endcase
```



Making Case Statements: casez

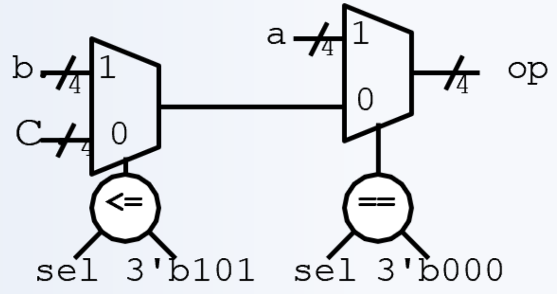
Case statement with do-not-cares.

- Treats `Z` and `?` characters as do-not-care bit positions in:
  - Case expression `(sel)`
  - Case item expression `3'b0??`
- Do-not-care bits are not considered in matching
  - E.g., `1'b?` will match `1'b0`, `1'b1`, `1'bZ` or `1'bX`

Also `casex`

- Treats `Z`, `X` or `?` as do-not-care.
  - Dangerous as treats uninitialized X value in case
  - expression as do-not-care.
  - Can hide initialization issues.
  - Not recommended to use.

```
logic [3:0] a, b, c;
logic [2:0] sel;
logic [3:0] op;

always_comb
  casez (sel)
    3'b000         : op = a;
    3'b0??, 3'b?0? : op = b;
    3'b11Z         : op = c;
    default        : op = 'bx;
  endcase
```



For Loop

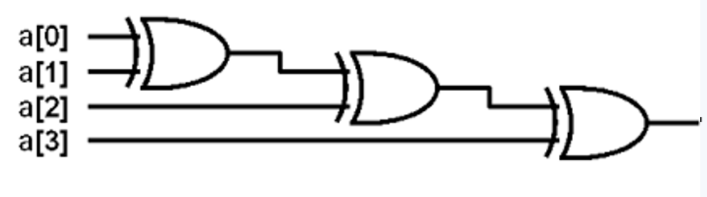
```
for ( initialization;
   termination_condition;
   step_expression)
 statement(s)
```

For loop has 3 clauses, and is executed as follows:

- Declare and initialize loop variable.
  - Good use for 32-bit 2-state type int.
  - Loop variable only visible inside the loop.
- While termination condition true.
  - Executes loop statements.
  - Then executes step expression to modify loop variable.

```
logic [3:0] a;
logic parity;

always_com begin
  parity = 1'b0;

  for (int i = 0; i <= 3; i++)
    parity = parity ^ a[i];

end
```

i++ shortcut
for i=i+1



Foreach Loop

```
foreach ( array [loop_variable])
```

foreach iterates over all elements of an array.

- Bounds and direction extracted from declaration.
- More convenient than `for loop`.

`foreach` loop variable:

- Does not have to be declared.
- Only visible inside loop.

Use multiple loop variables for multi-dimensional arrays.

- Equivalent to nested loops.

```
logic [3:0] a;
logic parity;

always_com begin
  parity = 1'b0;

  foreach (a [i])
    parity = parity ^ a[i];

end
```

Equivalent `for`

```
for (int i = 3; i >= 0; i--)
     parity = parity ^ a[i];
```

```
logic [7:0] vecarr [2:0];

always_comb
  foreach (vecarr [i, j])
    vecarr[i][j] = i + j;
```

i	j
2	7->0
1	7->0
0	7->0

Repeat Loop

```
repeat ( expression ) statement
```

Executes for a fixed number of iterations.

- Number can be an expression.

Used where an index is not required.

- For example, not indexing an array.

Example is a shift-multiplier.

```
logic [3:0] a, b;
logic [7:0] result;

logic [7:0] temp_a;
logic [3:0] temp_b;

always_comb begin
  temp_a = a;
  temp_b = b;
  result = 0;

  repeat ( 4 ) begin
    if ( temp_b[0] )
      result = result + temp_a;
    temp_a = temp_a << 1;      // left
    temp_b = temp_b >> 1;      // right
  end

end
```

break and continue

`break` and `continue` are allowed in loops.

- `break`
  - Jumps to the end of the loop.
  - Usually under conditional control.
  - Example left rotates `data` until msb = 1.

```
repeat (8) begin
  data = {data[6:0], data[7]};
  if (data[7])
    break;
end
...
```

- `continue`
  - Jumps to the next iteration of a loop.
  - Usually under conditional control.
  - Example counts number of zeros in `data`.

```
foreach (data [i]) begin
  if (data[i])
    continue;
  count = count + 1;
end
```

Using Operators

SystemVerilog Operators

Category	Symbol(s)
bit-wise	`~ & | ^ ~^`
reduction	`& ~& | ~| ^ ~^ ^~`
arithmetic	`** * / % + -`
shift	`<< >> <<< >>>`
relational	`< > <= >=`
equality	`== != === !==`
logical	`! && ||`
conditional	`?:`
concatenation	`{}`
replication	`{{}}`

Bit-Wise Operators

not	~
and	&
or	|
xor	^
xnor	~^ ^~

- Bit-wise operators operate on vectors.
- Operations are performed bit-by-bit on individual bits.
- The result is `1'b0`, `1'b1` or `1'bx`.
- Unknown bits in an operand do not necessarily lead to unknown bits in the result.

```
logic [3:0] veca, vecb, vecc;
logic [3:0] num;

initial begin
  veca = 4'b1001;
  vecb = 4'b1010;
  vecc = 4'b11x0;

  num = ~veca;           // num = 0110
  num = veca & 4'b0111;  // num = 0001
  num = veca & vecb;     // num = 1000
  num = veca | vecb;     // num = 1011
  num = vecb & vecc;     // num = 10x0
  num = vecb | vecc;     // num = 1110
end
```

Unary Reduction Operators

and	&
or	|
xor	^
nand	~&
nor	~|
xnor	~^ ^~

- Reduction operators perform a bit-wise operation on all the bits of a single operand.
- The result is always `1'b0`, `1'b1` or `1'bX`.

```
localparam CONST_A = 4'b0100,
           CONST_B = 4'b1111;

logic val;

initial begin
  val = &CONST_A ;                 // 0
  val = |CONST_A ;                 // 1
  val = &CONST_B ;                 // 1
  val = |CONST_B ;                 // 1
  val = ^CONST_A ;                 // 1
  val = ^CONST_B ;                 // 0
  val = ~|CONST_A;                 // 0
  val = ~&CONST_A;                 // 1
  val = ^CONST_A && &CONST_B;      // 1
end
```

Arithmetic Operators

add	+
subtract	-
multiply	*
divide	/
modulus	%

```
localparam CONST_INT = -3,
           CONST_5   = 5;

logic [3:0] veca, vecb, vecc, num;

integer val;

initial begin
  veca = 3;
  vecb = 4'b1010;
  vecc = 14;
  val = CONST_5 * CONST_INT; // -15
  val = (CONST_INT + 5)/2;   // 1
  val = CONST_5/CONST_INT;   // -1
  num = veca + vecb;         // 1101
  num = veca + 1;            // 0100
  num = CONST_INT;           // 1101
  num = vecc % veca;         // 0010
end
```

Shift Operators

logical shift << >>

- Ignores operand signs.
- Fills extra bits with 0.
- Implements division or multiplication by powers of two.

arithmetic shift <<< >>>

- Ignores the right operand sign.
- Left shift operates like logical left shift.
- Right shift preserves the left operand sign if the result is a signed expression.

```
logic          [7:0] veca, vecb;
logic signed [7:0] vecsign;

initial begin
 veca    = 8'b10011001;
 vecsign = 8'b10011001;

 vecb = veca <<  1;      // 00110010
 vecb = veca >>  1;      // 01001100
 vecb = vecsign <<< 1;   // 00110010
 vecb = vecsign >>> 1;   // 11001100
 vecb = veca << -1;      // 00000000
end
```

Negative integer implemented in 2s complement
but treated as binary for number of shifts: -1 is 2**32-1

Relational Operators

less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=

The result is:

- `1'b0` if the relation is false.
- `1'b1` if the relation is true.
- `1'bX` if either operand contains any `Z` or `X` bits.

```
logic [3:0] veca, vecb, vecc;
logic val;

initial begin
  veca = 4'b0011;
  vecb = 4'b1010;
  vecc = 4'b0x10;

  val = vecc > veca ;   // val = X
  val = vecb < veca ;   // val = 0
  val = vecb >= veca;   // val = 1
  val = vecb > vecc ;   // val = X
end
```

Logical Equality and Case Equality Operators

logical equality ==

- Unknown bits give an unknown result.

	0	1	z	x
0	1	0	0	0
1	0	1	0	0
z	0	0	1	0
x	0	0	0	1

- Also logical inequality !=

```
...
a = 2'b1x;
b = 2'b1x;

if (a == b)
  // values match & do not contain Z or X
else
  // values do not match or contain Z or X
```

else branch executed

case equality ===

- Result is always known.

	0	1	z	x
0	1	0	x	x
1	0	1	x	x
z	x	x	x	x
x	x	x	x	x

- Also case inequality !==

```
...
a = 2'b1x;
b = 2'b1x;

if (a === b)
  // values match exactly
else
  // values do not match
```

if branch executed

Wildcard Equivalency Operator

Checking selected *bits* from a large vector may require complex if expressions.

Wildcard equality allows bits to be defined as do-not-care.

- Like `casez`

An `X`, `Z` or `?` in the ***right-hand*** operand matches any value in the left.

- Asymmetric – only right side can have wildcard bits.

Also wildcard inequality (**!=?**).

```
logic [7:0] status = 8'b1101001;

if ((status[7:6] == 2'b11) | status[3] | status[0])
  $display("pass");

if (status ==? 8'b11?1??1)
  $display("pass");

if (status ==? 8'b11???0?)
  $display("pass");
```

Complex expression...

...replaced by wildcard equality

All if statements pass

Logical Operators

Reduce each operand to a single bit...

- Using OR reduction

...then perform a single bit operation.

not	`!`
and	`&&`
or	`||`

Operands are reduced to either true (`1'b0`) or false (`1'b1`) or unknown (`1'bX`).

1– if all bits 0

2– if any bit 1

`X` – if any bit is `Z` or `X` and no bit is `1`

```
localparam  FIVE = 5;
localparam CONST_A = 4'b0011,
           CONST_B = 4'b10xz,
           CONST_C = 4'b0z0x;

logic ans;

initial begin
  ans = !CONST_A;              // 0
  ans = CONST_A && 0;          // 0
  ans = CONST_A || 0;          // 1
  ans = CONST_A && FIVE;       // 1
  ans = CONST_A && CONST_B;    // 1
  ans = CONST_C || 0;          // X
end
```

Conditional Operator

Short form of the `if` statement for simple conditions.

Syntax

result = <condition> **?** <true value> **:** <false value>;

```
logic [3:0] a, b, op;
logic       sel;

assign op2 = sel ? a : b;
```

Operator use in **`assign`**

```
logic [3:0] a, b, op;
logic       sel;

always_comb
  op3 = sel ? a : b;
```

Operator use in procedural block

```
logic [3:0] a, b, op;
logic       sel;

always_comb
   if (sel == 1)
     op = b;
   else
     op = a;
```

Equivalent **`if`** statement

Concatenation Operator

concatenation { }

- Can select and join bits from different vectors to form a new vector.
  - Forms unsigned expression.
- Can reorganize vector bits to form a new vector.
  - Endian swaps / reverse / rotate
- Can use on either side of an assignment!

Literals in a concatenation must be explicitly sized.

- To calculate bit positions of result.

```
logic  [7:0]  veca,  vecb,  vecc,  vecd,  new;  logic
[3:0] nib1, nib2;

initial begin
 veca = 8'b00000011; vecb = 8'b00000100;
 vecc = 8'b00011000; vecd = 8'b11100000;

 new = {vecd[6:5], vecc[4:3], vecb[3:0]};
 // new = 8'b11_11_0100

 new = {2'b11, vecb[7:5], veca[4:3], 1'b1};
 // new = 8'b11_000_00_1

 new = {vecd[4:0], vecd[7:5]};
 // rotate vecd right 3 places
 // new = 8'b00000_111

 {nib1, nib2} = veca;
 // nib1 = 4'0000, nib2 = 4'0011
end
```

Replication Operator

replication `{ { } }`

- Reproduces a concatenation a set number of times.
- Syntax
  `{replicate{sized_expr}}`
- `replicate` must be:
  - A constant number or expression.
  - Without any `Z` or `X` values.

```
logic  veca = 1'b1;
logic [1:0] vecb = 2'b11;
logic [3:0] vecc = 4'b1001;
logic [7:0] bus;

initial begin
  // single bit veca replicated 8 times
  bus = {8{veca}}; // bus = 11111111

  // 4x veca concatenated with 2x vecc[1:0]
  bus = { {4{veca}}, {2{vecc[1:0]}} };
  // bus = 1111_01_01

  // vecc concatenated with 2x vecb
  bus = { vecc, {2{vecb}} };  // bus = 1001_11_11

  // vecc concatenated with 2x 1'b1
  // and replicated 2 times
  bus = { 2{vecc[2:1], {2{1'b1}}} };
  // bus = 00_1_1_00_1_1
end
```

Assignment Patterns

Define a list of values for an assignment to:

- Array elements.
- Structure fields (see later).

It is equivalent to individual assignments:

- Un-sized literals are allowed.

Pattern “keys” can be used:

- Array element index.
- `default` keyword.

There must be a pattern value for every array element:

- Either explicitly or using `default`.

```
logic[7:0] regarr [3:0];
logic[7:0] mem8x7 [127:0];


regarr = '{0,1,2,3};
/* equivalent to
regarr[3] = 0;
regarr[2] = 1;
regarr[1] = 2;
regarr[0] = 3;   */

regarr = '{3:1, default:0};
/* equivalent to
regarr[3] = 1;
regarr[2] = 0;
regarr[1] = 0;
regarr[0] = 0;   */

// initialize mem to 0
mem8x7 = '{default:0};
```

Thank You

NCDC
NUST
CHIP
DESIGN
CENTRE