Wired Nets

- The wor and trior nets shall create wired or configurations so that when any of the drivers is 1.

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

- The wand and triand nets shall create wired and configurations so that if any driver is 0, the value of the net is 0.

wand/triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

- Two types of strengths can be specified in a net declaration as follows:
  - **Charge strength** (small, medium, large) shall only be used when declaring a net of type **trireg**.
    - Examples:

```
trireg a;                          // trireg net of charge strength medium
trireg (large) #(0,0,50) cap1; // trireg net of charge strength large
                                   // with charge decay time 50 time units
trireg (small)signed [3:0] cap2; //signed 4-bit trireg vector of
                                   // charge strength small
```

  - **Drive strength** shall only be used when placing a continuous assignment on a net in the same statement that declares the net.

Nets type

- Net types

wire	tri	tri0	supply0
wand	triand	tri1	supply1
wor	trior	trireg	uwire

- Wire and tri:
  - A wire net can be used for nets that are driven by a single gate or continuous assignment.
  - The tri net type can be used where multiple drivers drive a net.

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Verilog Vectors

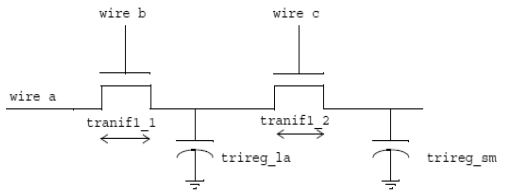
- **Scalar: A net or reg declaration without a range specification shall be considered 1 bit wide and is known as a scalar.**
  - **Example:**

```
wand        w;                  // a scalar net of type "wand"
reg         a;                  // a scalar register
wire        w1,w1;              // declares two wires
trireg (small) storeit;         // a charge storage node of strength small
```

- **Vector: Multibit net and reg data types shall be declared by specifying a range, which is known as a vector.**
  - **Example:**

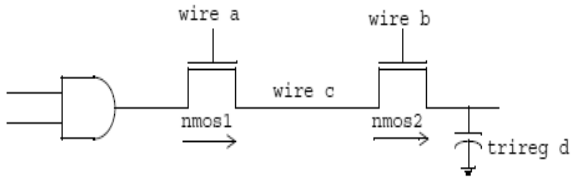
```
tri [15:0] busa;                // a three-state 16-bit bus
Reg signed [3:0] signed_reg;    // a 4-bit vector in range -8 to 7
reg [4:0] x, y, z;              // declares three 5-bit regs
reg [-1:4] b;                   // a 6-bit vector reg
```

Simulation results of charge sharing



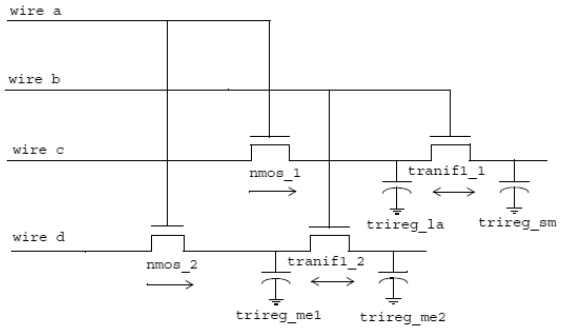
simulation time	wire a	wire b	wire c	trireg_la	trireg_sm
0	strong 1	1	1	strong 1	strong 1
10	strong 1	0	1	large 1	large 1
20	strong 1	0	0	large 1	small 1
30	strong 1	0	1	large 1	large 1
40	strong 1	0	0	large 1	small 1

Simulation values of a trireg and its driver



simulation time	wire a	wire b	wire c	trireg d
0	1	1	strong 1	strong 1
10	0	1	HiZ	medium 1

Simulation results of a capacitive network



simulation time	wire a	wire b	wire c	wire d	trireg_la	trireg_sm	trireg_me1	trireg_me2
0	1	1	1	1	1	1	1	1
10	1	[0]	1	1	1	1	1	1
20	1	0	[0]	1	[0]	1	1	1
30	1	0	0	[0]	0	1	[0]	1
40	[0]	0	0	0	0	1	0	1
50	0	[1]	0	0	0	[0]	[x]	[x]

Trireg Net

- The trireg net stores a value and is used to model charge storage nodes. A trireg net can be in one of two states:
  - **Driven state:** When at least one driver of a trireg net has a value of 1, 0, or x (driven value).
    - The strength of a **trireg** net in the driven state can be supply, strong, pull, or weak, depending on the strength of the driver.
  - **Capacitive state:** When all the drivers of a trireg net are at the high-impedance value (z), the trireg net retains
    - The strength of the value on the trireg net in the capacitive state can be small, medium, or large, depending on the size specified in the declaration of the trireg net.

Memory differences

- A memory of n 1-bit regs is different from an n-bit vector reg.

reg [1:n] rega; // An n-bit register is not the same
reg mema [1:n]; // as a memory of n 1-bit registers

Arrays

- An array declaration for a net or a variable declares an element type that is either scalar or vector

Declaration	Element type
reg x[11:0];	scalar **reg**
wire [0:7] y[5:0];	8-bit-wide vector wire indexed from 0 to 7
reg [31:0] x [127:0];	32-bit-wide **reg**

Array examples

- **Array examples**

```
reg [7:0] mema[0:255]; // declares a memory mema of 256 8-bit registers.
                       //Indices are 0 to 255
reg arrayb[7:0][0:255];  // declare a two-dimensional array of one bit registers
wire w_array[7:0][5:0]; // declare array of wires
integer inta[1:64];      // an array of 64 integer values
time chng_hist[1:1000];          // an array of 1000 time values
integer t_index;
```

- **Assignment to array elements**

```
mema = 0;              // Illegal syntax- Attempt to write to entire array
arrayb[1] = 0;         // Illegal Syntax - Attempt to write to elements  [1][0]..[1][255]
arrayb[1][12:31] = 0;  // Illegal Syntax - Attempt to write to elements [1][12]..[1][31]
mema[1] = 0;           // Assigns 0 to the second element of mema
arrayb[1][0] = 0;      // Assigns 0 to the bit referenced by indices [1][0]
inta[4] = 33559;       // Assign decimal number to integer in array
chng_hist[t_index] = $time; // Assign current simulation time to
                            // element addressed by integer index
```

Tri0 Tri1 Nets

- A tri0 net is equivalent to a wire net with a continuous 0 value of pull strength driving it.
- A tri1 net is equivalent to a wire net with a continuous 1 value of pull strength driving it.
- The truth tables model multiple drivers of strength strong on tri0 and tri1 nets.
  - The resulting value on the net has strength strong, unless both drivers are z, in which case the net has strength pull.

tri0	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0

tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	1

Expressions

- An operand can be one of the following:
  - Constant number (including real) or string
  - Parameter (including local and specify parameters)
  - Parameter (not real) bit-select or part-select (including local and specify parameters)
  - Net
  - Net bit-select or part-select
  - reg, integer, or time variable
  - reg, integer, or time variable bit-select or part-select
  - real or realtime variable
  - Array element
  - Array element (not real) bit-select or part-select
  - A call to a user-defined function or system-defined function that returns any of the above

Module parameters

- Parameters represent constants; hence, it is illegal to modify their value at run time.
- Module parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment.
  - This allows customization of module instances.
  - A parameter can be modified with the defparam statement or in the module instance statement.
- Typical uses of parameters are to specify delays and width of variables

Module parameters example

```
parameter msb = 7; // defines msb as a
                                    constant value 7
parameter e = 25, f = 9;       // defines two
constant numbers
parameter r = 5.7;         // declares r as a real
parameter
parameter byte_size = 8, byte_mask = byte_size - 1;
parameter average_delay = (r + f) / 2;

parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter p1 = 13'h7e;
parameter [31:0] dec_const=1'b1; // value converted
                                 // to 32 bits
parameter newconst = 'h4; //implied range of [2:0]
parameter newconst = 4; implied range of at least
                    [2:0]
```

Parameters

- Verilog HDL parameters do not belong to either the variable or the net group.
- Parameters are not variables; they are constants.
- There are two types of parameters:
  - module parameters
  - specify parameters.

Reduction operators

- The unary reduction operators shall perform a bitwise operation on a single operand to produce a single-bit result.

Operand	&	~&	|	~|	^	~^	Comments
4'b0000	0	1	0	1	0	1	No bits set
4'b1111	1	0	1	0	0	1	All bits set
4'b0110	0	1	1	0	0	1	Even number of bits set
4'b1000	0	1	1	0	1	0	Odd number of bits set

Operators

- Legal operators for use in real expressions

unary + unary -	Unary operators
+ - * / **	Arithmetic
> >= < <=	Relational
! && ||	Logical
== !=	Logical equality
?:	Conditional

- Operators not allowed for real expressions

{} {{}}	Concatenate, replicate
%	Modulus
=== !==	Case equality
~ & | ^ ^~ ~^	Bitwise
^ ^~ ~^ & ~& | ~|	Reduction
<< >> <<< >>>	Shift

Precedence rules for operators

+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (binary)	
^ ^~ ~^ (binary)	
| (binary)	
&&	
||	
?: (conditional operator)	
{} {{}}	Lowest precedence

Expressions

{} {{}}	Concatenation, replication
unary + unary -	Unary operators
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation
&&	Logical and
||	Logical or
==	Logical equality
!=	Logical inequality

===	Case equality
!==	Case inequality
~	Bitwise negation
&	Bitwise and
|	Bitwise inclusive or
^	Bitwise exclusive or
^~ or ~^	Bitwise equivalence
&	Reduction and
~&	Reduction nand
|	Reduction or

~|	Reduction nor
^	Reduction xor
~^ or ^~	Reduction xnor
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
? :	Conditional

Example

- **A replication operation may have a replication constant with a value of zero. This is useful in parameterized code. A replication with a zero replication constant is considered to have a size of zero and is ignored. Such a replication shall appear only within a concatenation in which at least one of the operands of the concatenation has a positive size.**

```
parameter P = 32;

// The following is legal for all P from 1 to 32

assign b[31:0] = { {32-P{1'b1}}, a[P-1:0] } ;

// The following is illegal for P=32 because the zero
// replication appears alone within a concatenation

assign c[31:0] = { {{32-P{1'b1}}}, a[P-1:0] }

// The following is illegal for P=32

initial
        $displayb({32-P{1'b1}}, a[P-1:0]);
```

Conditional operator

- The following example of a three-state output bus illustrates a common use of the conditional operator:
  - wire [15:0] busa = drive_busa ? data : 16'bz;
  - The bus called data is driven onto busa when drive_busa is 1. If drive_busa is unknown, then an unknown value is driven onto busa. Otherwise, busa is not driven.

Concatenations

- A concatenation is the result of the joining together of bits resulting from one or more expressions.
  - The concatenation shall be expressed using the brace characters { and }, with commas separating the expressions within.
    - This example concatenates four expressions:

      {a, b[3:0], w, 3'b101}
    - It is equivalent to the following example:

      {a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}

Shift operators

- There are two types of shift operators:
  - the logical shift operators, << and >>
    - In this example, the reg result is assigned the binary value 0100, which is 0001 shifted to the left two positions and zero-filled.

```
module shift;
reg [3:0] start, result;
initial begin
        start = 1;
        result = (start << 2);
end
```

  - the arithmetic shift operators, <<< and >>>
    - Example 2—In this example, the reg result is assigned the binary value 1110, which is 1000 shifted to the right two positions and sign-filled.

```
module ashift;
reg signed [3:0] start, result;
initial begin
        start = 4'b1000;
        result = (start >>> 2);
end
endmodule
```

Vector bit-select and part-select addressing: Example

For example:

```
reg [31: 0] big_vect;
reg [0 :31] little_vect;
big_vect[ 0 +: 8] // == big_vect[ 7 : 0]
big_vect[15 -: 8] // == big_vect[15 : 8]
little_vect[ 0 +: 8] // == little_vect[0 : 7]
little_vect[15 -: 8] // == little_vect[8 :15]
dword[8*sel +: 8] // variable part-select with fixed width
```

Vector bit-select and part-select addressing

- Bit-selects extract a particular bit from a vector net, vector reg, integer, or time variable, or parameter.
  - The bit can be addressed using an expression.
  - If the bit-select is out of the address bounds or the bit-select is x or z, then the value returned by the reference shall be x.
  - A bit-select or part-select of a scalar, or of a variable orparameter of type real or realtime, shall be illegal.

Vector bit-select and part-select addressing: Example

reg [15:0] big_vect;
reg [0:15] little_vect;

```
big_vect[lsb_base_expr +: width_expr]
little_vect[msb_base_expr +: width_expr]

big_vect[msb_base_expr -: width_expr]
little_vect[lsb_base_expr -: width_expr]
```

- **The msb_base_expr and lsb_base_expr shall be integer expressions, and the width_expr shall be a positive constant integer expression. The lsb_base_expr and msb_base_expr can vary at run time.**

Operands

- There are several types of operands that can be specified in expressions.
  - Reference to a net, variable, or parameter in its complete form.
    - All of the bits making up the net, variable, or parameter value shall be used as the operand.
  - Bit-select operand: A part-select operand shall be used to reference a group of adjacent bits in a vector net, vector reg, integer, or time variable, or parameter.

Array and memory addressing

```
reg [7:0] twod_array[0:255][0:255];
wire threed_array[0:255][0:255][0:7];

twod_array[14][1][3:0]      // access lower 4 bits of word
twod_array[1][3][6]         // access bit 6 of word
twod_array[1][3][sel]       // use variable bit-select
threed_array[14][1][3:0] // Illegal
```

Array and memory addressing

- **This declares a memory of 1024 eight-bit words**:

  ```
  reg [7:0] mem_name[0:1023];
  ```

- **memory address consistS of the name of the memory and an expression for the address, specified with the following format:**

  ```
  mem_name[addr_expr]
  ```

- **The addr_expr can be any integer expression; This illustrates memory indirection:**

  ```
  mem_name[mem_name[3]]
  ```

- **In this example, mem_name[3]addresses word three of the memory called mem_name. The value at word three is the index into mem_name that is used by the memory address mem_name[mem_name[3]].**

- **If the index is out of the address bounds or if any bit in the address is x or z, then the value of the reference shall be x.**

Array and memory addressing

- The declares an array of 256-by-256 eight-bit elements and an array 256-by-256-by-8 one-bit elements:

  ```
  reg [7:0] twod_array[0:255][0:255];
  wire threed_array[0:255][0:255][0:7];
  ```

- To access the array : by the name of the memory or array and an integer expression for each addressed dimension:

  ```
  twod_array[addr_expr][addr_expr]
  threed_array[addr_expr][addr_expr][addr_expr]
  ```

- The addr_expr is any integer expression. The array twod_array accesses a whole 8-bitvector, while the array threed_array accesses a single bit of the three-dimensional array. To express bit-selects or part-selects of array elements, the desired word shall first be selected by supplying an address for each dimension. Once selected, bit-selects and part-selects shall be addressed in the same manner as net and reg bit-selects and part-selects

Vector bit-select and part-select addressing: Example

```
reg [7:0] vect;
vect = 4;          // fills vect with the pattern 00000100
                   // msb is bit 7, lsb is bit 0
```

- If the value of addr is 2, then vect[addr] returns 1.
- If the value of addr is out of bounds, then vect[addr] returns x.
- If addr is 0, 1, or 3 through 7, vect[addr] returns 0.
- vect[3:0] returns the bits 0100.
- vect[5:1] returns the bits 00010.
- vect[expression that returns x] returns x.
- vect[expression that returns z] returns x.
- If any bit of addr is x or z, then the value of addr is x.

Assignments and truncation

- If the width of the right-hand expression is larger than the width of the left-hand side in an assignment, the MSBs of the right-hand expression will always be discarded to match the size of the left-hand side.

```
reg [5:0] a;
reg signed [4:0] b;
initial begin
        a = 8'hff; // After the assignment, a = 6'h3f
        b = 8'hff; // After the assignment, b = 5'h1f
end
```

Example of expression bit-length problem

- Given the following declarations:

  reg [15:0] a, b, answer; // 16-bit regs
- The intent is to evaluate the expression

  answer = (a + b) >> 1; //will not work properly
- where a and b are to be added, which can result in an overflow, and then shifted right by 1 bit to preserve the carry bit in the 16-bit answer.
- A problem arises, however, because all operands in the expression are of a 16-bit width. Therefore, the expression (a + b) produces an interim result that is only 16 bits wide, thus losing the carry bit before the evaluation performs the 1-bit right shift operation.

Example of self-determined expressions

```
reg [3:0] a;        reg [5:0] b;        reg [15:0] c;
initial begin
        a = 4'hF;
        b = 6'hA;
        $display("a*b=%h", a*b);// expression size is self-determined
        c = {a**b}; // expression a**b is self-determined
                // due to concatenation operator {}
        $display("a**b=%h", c);
        c = a**b; // expression size is determined by c
        $display("c=%h", c);
end
```

- Simulator output for this example:

  a*b=16 // 'h96 was truncated to 'h16 since expression size is 6
  a**b=1 // expression size is 4 bits (size of a)
  c=ac61 // expression size is 16 bits (size of c)

Bit length

Expression	Bit length	Comments
Unsized constant number[a]	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % & | ^ ^~ ~^	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: === !== == != > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: && ||	1 bit	All operands are self-determined
op i, where op is: & ~& | ~| ^ ~^ ^~ !	1 bit	All operands are self-determined
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...j}	L(i)+..+L(j)	All operands are self-determined
{i{j,...,k}}	i * (L(j)+..+L(k))	All operands are self-determined