Session 4

Verilog Data Types

System Tasks and Compiler Directives

Logic Values

Verilog has 4 logic values

Value Level	Condition in Hardware Circuit
0	Logic Zero, False, Low, etc.
1	Logic One, True, High, etc.
x	Unknown, Uninitialized, etc.
z	High Impedance, Tri-State, Floating, etc.

The 4 logic value system has many names, such as:

1 = high, true, asserted, etc.

0 = low, false, deasserted, etc.

x = unknown, net conflict, undefined, un-initialized, etc.

z = high impedance, tri-state, off, etc.

NOTE: The ‘x’ value in the 4-value logic system does NOT mean “don’t care”

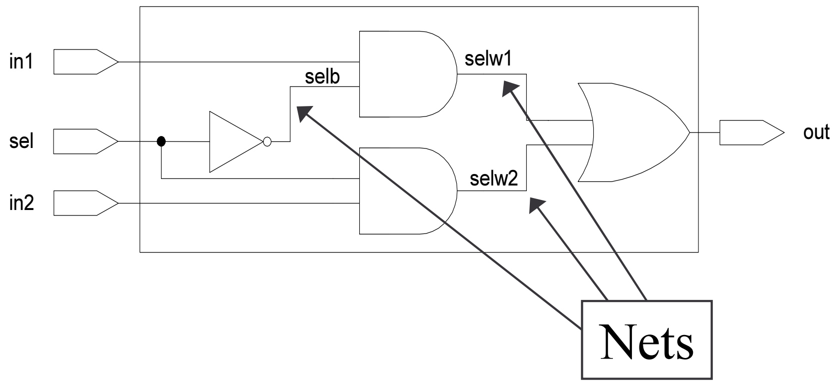
Data Type

- The Verilog Language has Three Major Data Types.
  - Each Data Type has a Subset Class.

Data Type	Description
Nets	Physical Connection Between Devices of a Circuit
Registers	Abstract Storage Elements
Parameters	Runtime Constants

The Net Data Type

- Nets represent the interconnection between hardware elements.
- Verilog automatically places a new value onto a net when the drivers of the net change value. For example: the ‘or’ gate will automatically drive a value to the ‘out’ net based on its inputs.



Net Classes

- Various nets are available to model different functionality.
  - ***wire, wand, wor, tri, triand, trior, trireg***, etc.
- The default data type in Verilog is net, and the default data class is ***wire***.
  - If a signal, variable, port, etc. is left undefined, Verilog will set it to a 1-bit ***wire*** by default.
- Net Declaration Syntax:
  - <net_type> [range] [delay] <net_name> [, net_name];

Examples of Nets

Example	Description
wire a;	Declaration of a single bit wire ‘a’.
wire [7:0] t;	Declaration of an 8 bit vector ‘t’.
wire s = 1’b0;	Declaration of wire ‘s’ assigned to ‘0’.
wire z = b | c;	Wire declaration ‘z’ continuously assigned “b | c”.
wire [15:0] m,n,o;	Three 16-bit vectors: “m”, “n”, and “o”.

Register Data Type

- A register holds a value until a new value is assigned to it.
- Registers are used in Behavioral Modeling, RTL Coding, and for applying stimulus in testbenches.
- Registers are not used for gate or switch level modeling.
- Register Declaration Syntax:
  - <reg_type> [range] <reg_name> [, reg_name];

Registers Classes

- The register type consists of four data classes.

Register Class	Description
reg	Unsigned integer variable of varying bit width.
integer	Signed integer variable, 32 bits wide.
real	Signed floating point variable.
time	Unsigned integer variable, 64 bits wide.

Examples of Registers

Example	Description
reg a;	Scalar register “a”.
reg [7:0] t;	8-bit Vector register “t”;
integer z;	Integer “z”
real q;	Real number “q”

Registers

- Registers are the only data type allowed as a LHS assignment in procedural block.

```
module cntr (count, clk);
input clk;
output [31:0] count;
integer count
always @(posedge clk) begin
  count = count + 1;
end
endmodule
```

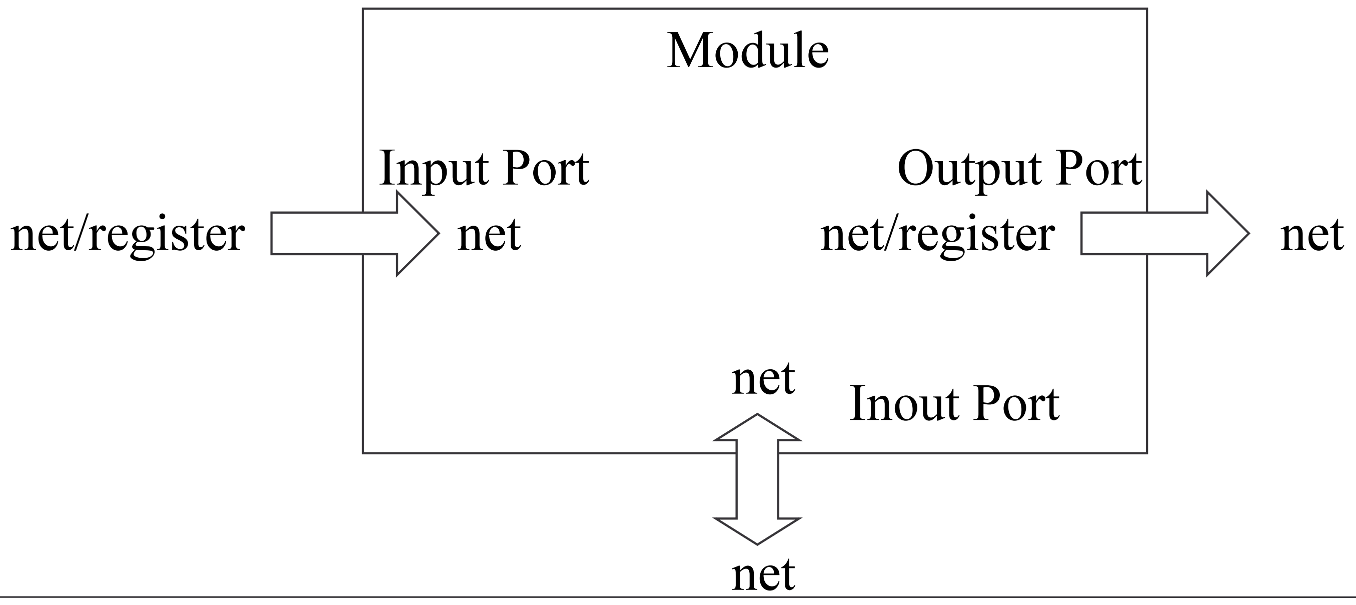
The LHS of a procedural assignment is always a register data type

```
module testbench;
  reg in1,in2;
  examp u1 (out1,out2, in1,in2);
  initial begin
    {in1,in2} =  0;
    #10 in1 = 1;
    #10 in2 = 1;
    #10 $finish;
  end
endmodule
```

Do not confuse the Register data type with actual hardware registers. In RTL coding, declaring a register data type does not automatically infer some hardware storage element such as a Flip-Flop or Latch.

Choosing the Correct Data Class

- One of the most common errors in using Verilog is choosing the incorrect data type declaration!



Follow these rules and you shouldn't have problems with data types in your Verilog code:

Use a register (usually reg) for:
-LHS of all procedural assignments will always be data type 'Register' and typically will be a 'reg'.

Use a net (usually wire) for:
-Continuous assignments using the 'assign' statement.
-Inputs to the design block.
-Inout ports.
-Any wire interconnect in a netlist.
-any output of a Verilog primitive.

Choosing the Correct Data Class (cont.)

- An input port can be driven by a net or a register, but the port itself can only drive a net.
- An output port can be driven by a net or register, but the port itself can only drive a net.
- An inout port can only be driven by a net, and the port itself can only drive a net.
- If a signal is assigned a value from a procedural block, it must be declared as a register.

Common Mistakes - Data Types

- Trying to make a procedural assignment to a signal that is declared either as a net, or undeclared.
  - Gives an illegal assignment error
- Connecting the output of an instance to a signal declared as a register.
  - Gives an illegal output port specification error
- Declaring a signal as a module input and a register.
  - Gives an incompatible declaration error

Parameters

- Parameters are used to declare run-time constants
- Parameters can be used to declare things like: delay values, variable vector widths, etc
- Syntax:
  - parameter <list of assignments>

```
module somegate (out, in1, in2);

input in1, in2;
output out;

parameter width = 8;

wire [width-1:0] w1;
reg [width-1:0] w2;
…
endmodule
```

Overriding Parameter Values (cont.)

- Parameters can be overridden using the 'defparam' statement.
- Syntax:
  - defparam <list of parameters to override>;

```
module somegate (out, in1, in2);

input in1, in2;
output out;

parameter width = 8;

wire [width-1:0] w1;
…
endmodule
```

```
module testbench
…
somegate u1 (out, in1, in2);

defparam u1.width = 4;
…
endmodule
```

Local Parameters

- Local Parameters cannot be overwritten by the ***defparam*** statement constants
- Local Parameters can be used to avoid inadvertently overwriting parameters.
- Syntax:
  - localparam <list of assignments>

```
module somegate (out, in1, in2);

input in1, in2;
output out;

localparam   state1 = 2'b00,
             state2 = 2'b01,
             state3 = 2'b10,
             state4 = 2'b11;

…
endmodule
```

Warning!
Local Parameters are part of the Verilog 2001 Standard and will not work in simulators that do not support Verilog 2001.

Arrays

- Arrays are allowed in Verilog.
- Arrays can be scalar in width by variable depth.
  - 1 x N
- Arrays can be vector in width by variable depth.
  - M x N

- Verilog 2001 supports multi-dimensional arrays.
  - 1 x A x B x C x <n>
  - M x A x B x C x <n>

Array Examples

Example	Description
integer var_a [0:15]	Array of 16 integers
reg [15:0] mem [0:15]	A 16x16 Array.
reg var_c [0:63]	Scalar array of 64 elements.

Memory Arrays

- Memories are declared in Verilog by creating a two-dimensional array.
- Syntax:
  - **reg [<width declaration>] <identifier> [<address depth declaration>];**

```
reg [15:0] mem [0:1023];                // A 1k x 16-bit memory.
```

- Parameters can be used to define memory sizes.

```
parameter bit_size = 16;
parameter address_size = 1024

reg [(bit_size – 1): 0] mem [address_size - 1 : 0];  // A 1k x 16-bit memory.
```

Memory Addressing

- A memory element is addressed by the index to the memory array. You can only reference one word of memory at a time.

```
module my_memory;

parameter bit_size = 16;
parameter address_size = 1024

reg [(bit_size - 1 ): 0] mem_word;                         //For temporary storage
reg [(bit_size – 1): 0] mem [address_size - 1 : 0];  // A 1k x 16-bit memory.

initial begin
  $displayb (mem[5]);              // displays all 16 bits of the memory at location 5
   mem_word  = mem[5];
  $displayb (mem_word[8]);    // displays bit 8 of memory at location 5
end
```

The Verilog 2001 standard provides a way to directly address memory contents. Refer to Palnitkar’s book for examples.

System Tasks and Functions

- The ‘$’ denotes a Verilog system task.
- Syntax: $<identifier>
- There are a number of system tasks that perform various operations, such as:
  - Displaying or monitoring signals: ***$display, $monitor***
  - Finding current simulation time: ***$time, $realtime***
  - Stopping the simulation: ***$stop***
  - Finishing the simulation: ***$finish***
  - Many others, see section 9.5 in Verilog HDL book by Palnitkar.

Verilog also provides the user with the ability to create their own system tasks. This feature is called PLI (Programmers Language Interface). Some PLI is integrated into the Verilog simulators here at AMIS. This includes PLI for tools such as: Test_IDDQ, Powerfault, and Debussy.

Displaying Information

- ***$display*** and ***$monitor*** are the main system tasks for displaying values on variables in a Verilog simulation.
- Syntax (with text):
  - ***$display*** (“<message %”formatter”>”,<variable1>, <variable2>, <variable*n*>*);*
  - ***$monitor*** (“<message %”formatter”>”,<variable1>, <variable2>, <variable*n*>*);*
- Syntax (without text):
  - ***$display*** (<variable1>, <variable2>, <variable*n*>*);*
  - ***$monitor*** (<variable1>, <variable2>, <variable*n*>*);*

String Formatters

Formatter	Result
%d	Display variable with decimal format (default)
%b	Display variable in binary format
%s	Display string (see section 3.2.9 in Palnitkar)
%h	Display variable in Hex format
%c	Display ASCII Character
%m	Display hierarchical name (no argument required)
%v	Display strength
%o	Display variable in octal format
%t	Display in time format (used with $timeformat)
%e	Display real number scientific format
%f	Display real number decimal format
%g	Display real number in scientific or decimal format (whichever is shorter)

Display Examples

```
`timescale 1ns/1ps
module displayexamp;

  integer i;
  reg a;
  reg [3:0] x;
  real z;

always @(i or a or x or z) begin
    $display("time = %t   i = %0h   a = %b   x = %d   z = %f",
                  $realtime,i,a,x,z);
end
initial begin
    $timeformat(-9,3," ns",10);
    for (i = 0;i < 16; i = i + 1) begin
      a = i;
      x = i[3:0];
      z = z + i + 0.171;
      #10;
    end
  end
endmodule
```

```
time =    0.000 ns   i = 0   a = 0   x =  0   z = 0.171000
time =   10.000 ns   i = 1   a = 1   x =  1   z = 1.342000
time =   20.000 ns   i = 2   a = 0   x =  2   z = 3.513000
time =   30.000 ns   i = 3   a = 1   x =  3   z = 6.684000
time =   40.000 ns   i = 4   a = 0   x =  4   z = 10.855000
time =   50.000 ns   i = 5   a = 1   x =  5   z = 16.026000
time =   60.000 ns   i = 6   a = 0   x =  6   z = 22.197000
time =   70.000 ns   i = 7   a = 1   x =  7   z = 29.368000
time =   80.000 ns   i = 8   a = 0   x =  8   z = 37.539000
time =   90.000 ns   i = 9   a = 1   x =  9   z = 46.710000
time =  100.000 ns   i = a   a = 0   x = 10   z = 56.881000
time =  110.000 ns   i = b   a = 1   x = 11   z = 68.052000
time =  120.000 ns   i = c   a = 0   x = 12   z = 80.223000
time =  130.000 ns   i = d   a = 1   x = 13   z = 93.394000
time =  140.000 ns   i = e   a = 0   x = 14   z = 107.565000
time =  150.000 ns   i = f   a = 1   x = 15   z = 122.736000
```

Monitor Example

```
`timescale 1ns/1ps
module displayexamp;

  integer i;
  reg a;
  reg [3:0] x;
  real z;

  initial begin
    $timeformat(-9,3," ns",10);
    $monitor("time = %t   i = %0h   a = %b   x = %d   z = %f",
                 $realtime,i,a,x,z);
  end

  initial begin
    for (i = 0;i < 16; i = i + 1) begin
      a = i;
      x = i[3:0];
      z = z + i + 0.171;
      #10;
    end
  end

endmodule
```

```
time =     0.000 ns   i = 0   a = 0   x =  0   z = 0.171000
time =    10.000 ns   i = 1   a = 1   x =  1   z = 1.342000
time =    20.000 ns   i = 2   a = 0   x =  2   z = 3.513000
time =    30.000 ns   i = 3   a = 1   x =  3   z = 6.684000
time =    40.000 ns   i = 4   a = 0   x =  4   z = 10.855000
time =    50.000 ns   i = 5   a = 1   x =  5   z = 16.026000
time =    60.000 ns   i = 6   a = 0   x =  6   z = 22.197000
time =    70.000 ns   i = 7   a = 1   x =  7   z = 29.368000
time =    80.000 ns   i = 8   a = 0   x =  8   z = 37.539000
time =    90.000 ns   i = 9   a = 1   x =  9   z = 46.710000
time =   100.000 ns   i = a   a = 0   x = 10   z = 56.881000
time =   110.000 ns   i = b   a = 1   x = 11   z = 68.052000
time =   120.000 ns   i = c   a = 0   x = 12   z = 80.223000
time =   130.000 ns   i = d   a = 1   x = 13   z = 93.394000
time =   140.000 ns   i = e   a = 0   x = 14   z = 107.565000
time =   150.000 ns   i = f   a = 1   x = 15   z = 122.736000
```

Compiler Directives

- Compiler directives are indicated with the “back tick” (`). **Not to be confused with the apostrophe (‘).**
- Directs the compiler to take certain actions.
- Some Common Directives:
  - `***timescale*** – Defines the time units and resolution for the simulator
  - `***include*** – allows the insertion of an entire file
  - `***define*** – Used for macro definitions.
- Compiler directives remain active until they are deactivated, overridden, or the simulation completes.

Compiler Directives (cont.)

```
`timescale 1ns/1ps  // Time unit 1ns
`define delayval #10  // macro delayval == #10

module testbench;
reg a, b, c;

dut u1 (out,a,b,c);

`include "designparameters.txt";

initial begin
a=0;   b=0;   c=0;
`delayval a = parama;
`delayval b = paramb;
`delayval c= paramc;
`delayval $finish;
end
endmodule
```

```
\\designparameters.txt file

parameter parama = 0;
parameter paramb = 1;
parameter paramc = 0;
```

Delay Specification

- The ‘#’ denotes the delay specification in procedural statements and gate instances, but not module instances.

Gate Delay Specification

```
module twoinmux (out, in1, in2, sel);

input in1, in2, sel;
output out;

  not #1 u1 (selb, sel);
  and #2 u2 (selw1, in1, selb);
  and #2 u3 (selw2, in2, sel);
  or   #1 u4 (out, selw1, selw2);

endmodule
```

Procedural Statement Delay

```
module testbench;
reg in1, in2, sel;

twoinmux u1 (out,in1,in2,sel);

initial begin
{in1,in2,sel} = 0;
#10 in1 =1;
#10 sel = 1;
#10 $finish;
end
endmodule
```

Module Statement Delay (Illegal!)

```
module testbench;
reg in1, in2, sel;

twoinmux #3 u1 (out,in1,in2,sel);

end
```

Review

- What is the primary difference between a net and a register?
- How do you override a ***parameter***?
- How do you define a memory in Verilog?
- How do you view one bit in a memory array in Verilog?
- Name two system tasks for monitoring signals in a Verilog simulation.
- What does the `***timescale*** directive do?
- Where is it illegal to use the ‘***#’*** delay specification?