ECE 351
Verilog and FPGA Design

Lecture 3: **Logic simulation w/ QuestaSim**
SystemVerilog language rules
Modules, ports, and hierarchy
Literals, nets, and vars

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science
Portland State University

Logic Simulation w/ QuestaSim

Example: ..\examples\ripple_carry_counter.pdf

QuestaSim Tutorial: ..\docs\Questa® SIM Tutorial.pdf

Using QuestaSim at PSU: ..\docs\UsingMentorQuestaAtPSU_R2_0.pdf

Portland State University

Review: Some Definitions

- **Module** – the basic building block in Verilog
  - Can be an element or a collection of lower-level design blocks
  - Can provide abstraction…hides the details of the implementation (i.e. possible to modify block internals without affecting the rest of the design)
- **Instance** – Module is a template, instance is the actual object
- **Simulation** – The act of applying inputs to and monitoring the outputs from a Verilog model
- There are two distinct components of a simulation
  - **Design Block** – the implementation of the desired functionality
  - **Stimulus Block (Test Bench or Testbench)** – The inputs applied to the design block

QuestaSim Demo

Example: ..\examples\ripple_carry_counter.pdf

Steps to run QuestaSim remotely (Windows)

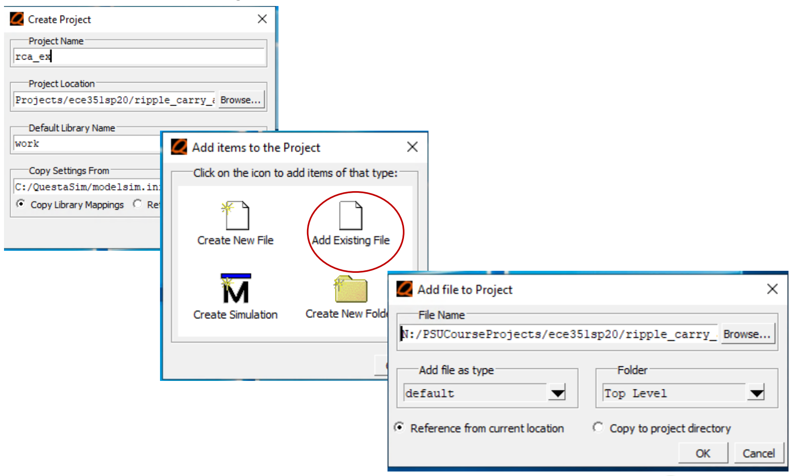
- ☐ (optional) Use windows `subst` or `visual subst` to map your shared drive to your PC (this is the `N:` drive on my laptop)
- ☐ Start a VPN session using either `OpenVPN` or Cisco `AnyConnect`
- ☐ Start a Remote Desktop session to `ts.cecs.pdx.edu` or use the Web-based connection at https://rdp.cecs.pdx.edu/
- ☐ Open QuestaSim, create and populate (add your simulation files) the project
- ☐ Compile all of the individual files, starting w/ files that are accessed from other files (ex: packages)
- ☐ Fix, recompile, … until all of the files compile with no errors
- ☐ Select the `Library` tab and open the `work` folder. Select your testbench file, right click and select `simulate`
- ☐ Fix any Load errors. When the simulation loads cleanly (a bunch of windows will open) type `log -r *` into the console window
- ☐ Add variables to your Wave window and save. This will save yur wave settings in a file called wave.do
- ☐ Type run –all (or use the GUI) to run your simulation

Portland State UNIVERSITY

Using the QuestaSim GUI

Create a new QuestaSim Project and add the source files

`File/New Project…`



Create Project

Project Name: rca_ex

Project Location: Projects/ece351sp20/ripple_carry_

Default Library Name: work

Copy Settings From: C:/QuestaSim/modelsim.in

(● Copy Library Mappings)

Add items to the Project

Click on the icon to add items of that type:

Add file to Project

File Name: N:/PSUCourseProjects/ece351sp20/ripple_carry_

Add file as type: default

Folder: Top Level

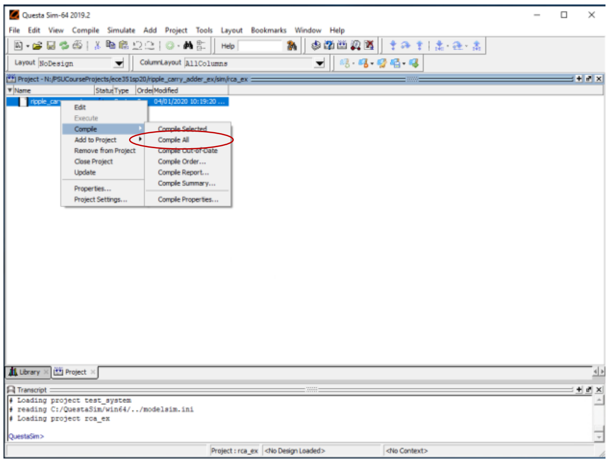
(● Reference from current location) ○ Copy to project directory

Portland State UNIVERSITY

Using the QuestaSim GUI (cont'd)

Compile the source files

`Compile/Compile All`

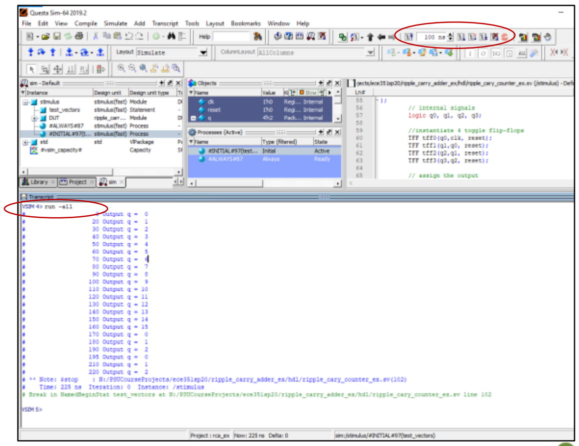


```
# Loading project test_system
# reading C:/QuestaSim/win64/../modelsim.ini
# Loading project rca_ex

QuestaSim>
```

Using the QuestaSim GUI (cont'd)

Load the simulation model

`Select Library tab, open work directory, select top module, right click and select Simulate`

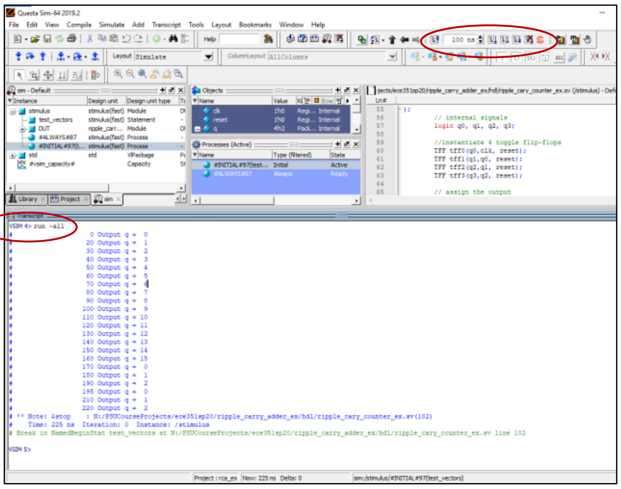

Questa Sim-64 2019.2
File Edit View Compile Simulate Add Transcript Tools Layout Bookmarks Window Help
100 ns
Layout Simulate
ColumnLayout AllColumns
sim - Default
Instance Design unit Design unit type
stimulus stimulus(fast) Module
test_vectors stimulus(fast) Statement
DUT ripple_carr... Module
#ALWAYS#87 stimulus(fast) Process
#INITIAL#97(t... stimulus(fast) Process
std std VPackage
#vsim_capacity# Capacity
Objects
Name Value
clk 1'h0 Regi... Internal
reset 1'h0 Regi... Internal
q 4'h2 Pack... Internal
Processes (Active)
Name Type (filtered) State
#INITIAL#97(test... Initial Active
#ALWAYS#87 Always Ready
jects/ece351sp20/ripple_carry_adder_ex/hdl/ripple_cary_counter_ex.sv (/stimulus)
Library
Project
sim
Transcript
Project : rca_ex
Now: 225 ns Delta: 0
sim:/stimulus/#INITIAL#97(test_vectors)

```
);

        // internal signals
        logic q0, q1, q2, q3;

        //instantiate 4 toggle flip-flops
        TFF tff0(q0,clk, reset);
        TFF tff1(q1,q0, reset);
        TFF tff2(q2,q1, reset);
        TFF tff3(q3,q2, reset);

        // assign the output
```

```
VSIM 4> run -all
#                   10 Output q =  0
#                   20 Output q =  1
#                   30 Output q =  2
#                   40 Output q =  3
#                   50 Output q =  4
#                   60 Output q =  5
#                   70 Output q =  6
#                   80 Output q =  7
#                   90 Output q =  8
#                  100 Output q =  9
#                  110 Output q = 10
#                  120 Output q = 11
#                  130 Output q = 12
#                  140 Output q = 13
#                  150 Output q = 14
#                  160 Output q = 15
#                  170 Output q =  0
#                  180 Output q =  1
#                  190 Output q =  2
#                  195 Output q =  0
#                  210 Output q =  1
#                  220 Output q =  2
# ** Note: $stop    : N:/PSUCourseProjects/ece351sp20/ripple_carry_adder_ex/hdl/ripple_cary_counter_ex.sv(102)
#    Time: 225 ns  Iteration: 0  Instance: /stimulus
# Break in NamedBeginStat test_vectors at N:/PSUCourseProjects/ece351sp20/ripple_carry_adder_ex/hdl/ripple_cary_counter_ex.sv line 102

VSIM 5>
```

Using the QuestaSim GUI (cont'd)

Run the simulation



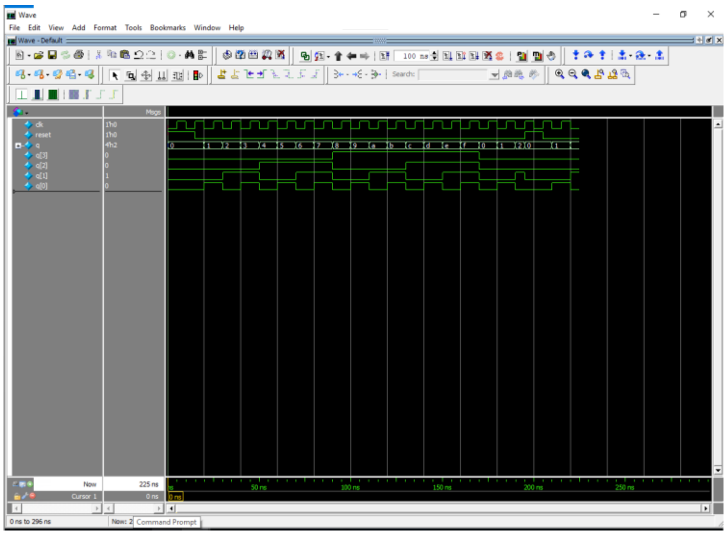
```
    // internal signals
    logic q0, q1, q2, q3;

    //instantiate 4 toggle flip-flops
    TFF tff0(q0,clk, reset);
    TFF tff1(q1,q0, reset);
    TFF tff2(q2,q1, reset);
    TFF tff3(q3,q2, reset);

    // assign the output
```

```
VSIM 4> run -all
#                      0 Output q =  0
#                     20 Output q =  1
#                     30 Output q =  2
#                     40 Output q =  3
#                     50 Output q =  4
#                     60 Output q =  5
#                     70 Output q =  6
#                     80 Output q =  7
#                     90 Output q =  8
#                    100 Output q =  9
#                    110 Output q = 10
#                    120 Output q = 11
#                    130 Output q = 12
#                    140 Output q = 13
#                    150 Output q = 14
#                    160 Output q = 15
#                    170 Output q =  0
#                    180 Output q =  1
#                    190 Output q =  2
#                    195 Output q =  0
#                    210 Output q =  1
#                    220 Output q =  2
# ** Note: $stop    : N:/PSUCourseProjects/ece351sp20/ripple_carry_adder_ex/hdl/ripple_cary_counter_ex.sv(102)
#    Time: 225 ns  Iteration: 0  Instance: /stimulus
# Break in NamedBeginStat test_vectors at N:/PSUCourseProjects/ece351sp20/ripple_carry_adder_ex/hdl/ripple_cary_counter_ex.sv line 102

VSIM 5>
```

Project : rca_ex   Now: 225 ns Delta: 0   sim:/stimulus/#INITIAL#97(test_vectors)

Portland State UNIVERSITY

Using the QuestaSim GUI (cont'd)



Portland State UNIVERSITY

SystemVerilog Language Rules

Source material drawn from:
- *Roy's lecture notes*
- *RTL Modeling with SystemVerilog for Simulation and Stimulus* by Stuart Sutherland

Lexical Conventions

- SystemVerilog syntax is similar to C
  - Programs consist of a stream of tokens (comments, delimeters, numbers, strings, identifiers and keywords)
  - Case-sensitive (`Roy` is not the same as `roy` or `ROY`)
- Whitespace
  - Blank spaces (`\b`), tabs (`\t`), and newlines (`\n`) are whitespace
  - Whitespace is ignored except when it separates tokens or when it is part of a string
- Comments
  - Follows the C conventions
  - `//` - single line comment. Everything from `//` to end of line is ignored
  - `/*…*/` - Multiple line comment. Starts with `/*` and ends with `*/`
  - Cannot nest multiple line comments but you can embed single line comments in a multiple line comment

13

Lexical Conventions (cont'd)

- Three types of operators
  - Unary – precedes the operator (ex: `a = ~b`)
  - Binary – appears between two operators (ex: `a = b & c`)
  - Ternary – two separate operators that separate 3 operators (ex: `a = b ? c : d`)
- Identifiers and Keywords
  - Made up of alphanumeric characters, the underscore (_) or dollar sign ($)
  - Are case-sensitive and cannot start with a digit or $
  - Keywords are special identifiers reserved to define the language constructs. All are lowercase

14

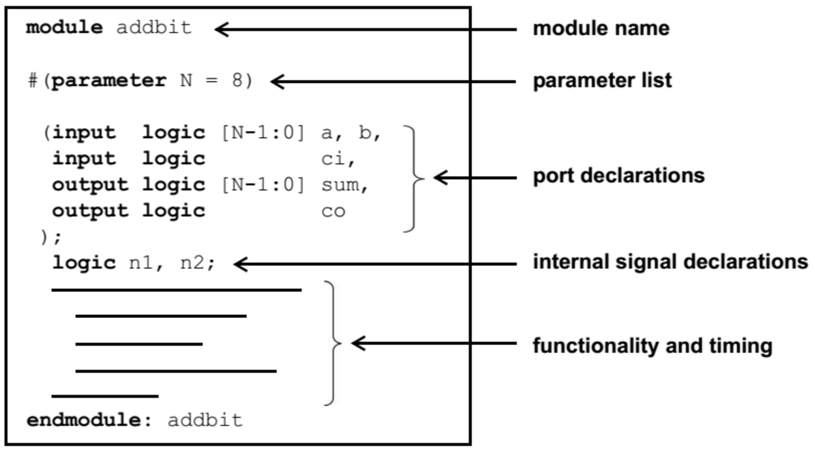
Logic Value System

- Four valued logic: 0, 1, z, x
- Each port, net, or register can be in one of the four values:
  - 0 - Logic 0, GND, …
  - 1 - Logic 1, VDD, …
  - x - Don't Know or undefined
    - Simulated circuit could have a value but simulator can't determine what it is.
    - Physical circuit will have a value but you don't know what it is and it could vary from chip to chip, be dependent on temperature and/or other undesirable behaviors
    - Ex: uninitialized register, OR gate w/ 'x' on input(s)
  - z - High impedance
    - Must be assigned under control
    - Use the form: **`assign y = en ? a : 1'bz;`**

Modules and Ports

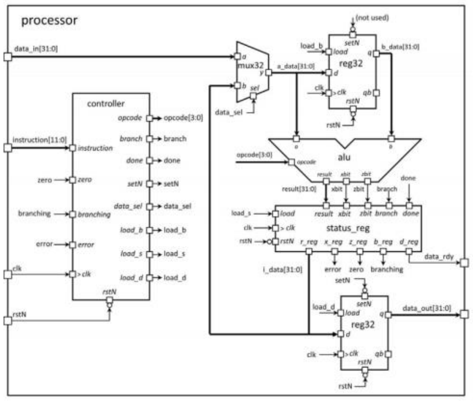
- A **Module** is the basic building block in Verilog
  - Begins with the `module <name>` and ends with the keyword `endmodule`. All other parts are optional
  - A module definition is a template…you must add one or more instances of the module to include the logic in your design
  - You can define more than one module in a single file but the convention is one file per module (with a .v extension)
  - Modules in a design can be defined in any order in one or more files
- A **Port** provides the interface by which a module can communicate with its environment
  - The ports list is optional but in an ASIC or FPGA design the ports in the top level of the synthesized design may map to pins on the device
  - Three types of ports declarations – `input`, `output`, `inout`

Modules



```
module addbit

#(parameter N = 8)

 (input  logic [N-1:0] a, b,
  input  logic         ci,
  output logic [N-1:0] sum,
  output logic         co
 );
  logic n1, n2;

endmodule: addbit
```

Module Hierarchy



```
module processor (/* port declarations */);

  ...// internal net declarations

  controller cntlr (/* port connections */);
  mux32 mux (/* port connections */);
  alu alu (/* port connections */);
  status_reg s_reg (/* port connections */);
  reg32 b_reg (/* port connections */);
  reg32 d_reg (/* port connections */);
endmodule: processor
```

The syntax of a module instantiation is:

```
module_name #(parameter_values) instance_name (connections_to_ports);
```

Portland State UNIVERSITY

Ports

Two ways to connect to a module's ports:

- ☐ Ordered port connections

```
dff d1 (out, /*not used*/, in, clock, reset);
```

- ☐ Named port connections

```
dff d1 (.q(out), .qb(/*not used*/),
        .d(in), .clk(clock), .rst(reset) );
```

```
instruction_decode decoder (
  .alu_opcode(alu_opcode),
  .alu_a_sel(alu_a_sel),
  .alu_b_sel(alu_b_sel),
  .w_reg_enable(w_reg_enable),
  .reg_file_sel(reg_file_sel),
  .zero_enable(zero_enable),
  .carry_enable(carry_enable),
  .polarity(polarity),
  .option(isoption),
  .tris(istris),
  .instruct_reg(instruct_reg)
);
```

Named Port connections

```
instruction_decode decoder (
  alu_opcode, alu_a_sel, alu_b_sel,
  w_reg_enable, reg_file_sel,
  zero_enable, carry_enable,
  polarity, isoption, istris,
  instruct_reg
);
```

Ordered Port connections

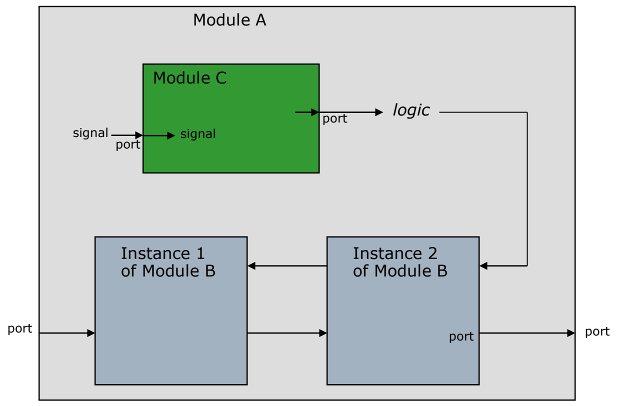
Portland State UNIVERSITY

Ports (cont'd)

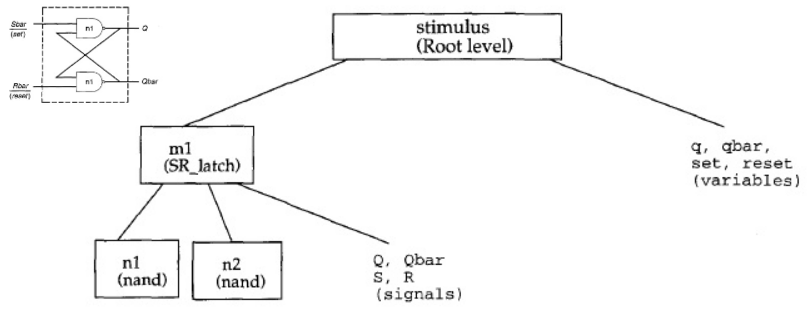
- ☐ Modules often uses a net name identical to the instantiated module's port name when connecting to the port:
- ☐ Explicit port <-> net mapping
  - Memory M(.address(address),.data(data),.control(ctrl));
- ☐ SystemVerilog provides shortcuts for making port connections
  - .name ("dot name")
  - .* ("dot star")
  - Interfaces
- ☐ SystemVerilog allows .name:
  - Memory M(.address, .data, .control(ctrl));
  - The net must match in name, type, and size or an error results
- ☐ .*
  - Causes all ports of the instantiated module to be connected to a net with the same name, type, and size if it exists

Module Hierarchy

- ☐ Designs inside designs
- ☐ Higher-level designs instantiate lower-level design



Hierarchical Names



```
stimulus                      stimulus.q
stimulus.qbar                 stimulus.set
stimulus.reset                stimulus.m1
stimulus.m1.Q                 stimulus.m1.Qbar
stimulus.m1.S                 stimulus.m1.R
stimulus.n1                   stimulus.n2
```

Literals, nets, and vars

Source material drawn from:
- Alex P., Mark F. and Roy K. ECE 571 lecture slides
- Roy K. ECE 351 and Verilog Workshop lecture slides
- *Logic Design and Verification Using SystemVerilog (Revised)* by Donald Thomas
- *RTL Modeling with SystemVerilog for simulation and Synthesis* by Stuart Sutherland

Literals

- **sized** or **unsized**: <size>’[s]<radix><number>
  - **Sized** example: 3’b010 = 3-bit wide binary number
    - The prefix (3) indicates the size of number
  - **Unsized** example: 123 = 32-bit wide decimal number by default
  - **Signed** example: -8’sd39 = 8-bit wide signed number
    - **Defaults**
      - No specified <base format> defaults to **decimal**
      - No specified <size> defaults to **32-bit** number
      - No “s” defaults to unsigned
- Radix:
  - Decimal (’d or ’D) 16’d255 = 16-bit wide decimal number
  - Hexadecimal (’h or ’H) 8’h9a = 8-bit wide hexadecimal number
  - Binary (’b or ’B) ’b1010 = 32-bit wide binary number
  - Octal (’o or ’O) ’o21 = 32-bit wide octal number

Literals (cont’d)

- ‘_’ (underscore): used for readability (ignored)
  - Example: `32’h21_65_bc_fe` = 32-bit hexadecimal number
- ‘x’ or ‘X’ (unknown value)
  - Example: `12’hxxx` = 12-bit hexadecimal number, unknown state
- ‘z’ or ‘Z’ (high impedance value)
  - Example: `1’bz` = 1-bit high impedance number
- ‘?’ can used in place of ‘z’ or ‘Z’ to represent hi-impedance and don’t care

Mismatched size and values

- Legal to specify a literal integer with a bit-width != number of bits required:
  - 4’hFACE // 4-bit width, 16-bit unsigned value
  - 16’sh8 // 16-bit size, 4 bit signed value, MSB set
  - 32’bz // 32-bit width, 1-bit unsigned valued
- SystemVerilog will adjust to match according to these rules:
  - When size < #bits than value, left-most bits truncated
  - When size > #bits than value, value is left-extended
    - if left-most bit of value is 0 or 1, additional bits filled w/ 0
    - If left-most bit of value is Z, upper bits are filled w/ Z
    - If left-most bit of value is X, upper bits are filled w/ X
  - e.g. Value is not sign extended even is literal integer is specified as signed. Sign extension occurs when signed literal value is used in operations and assignment statements

Mismatched size and values

Your turn:

- 4’hFACE //
- 16’sh8 //
- 32’bz //

- 4’hFACE // truncates to 4’hE
- 16’sh8 // extends to 16’hsh0008
- 32’bz // extends to 32’hZZZZZZZZ

Vector fill and floating-point literal values

- SystemVerilog provides a special form of unsized literals:
  - Sets all bits of vector of any size to 0, 1, X, or Z
  - Vector size is automatically determined based on context
    - '0 fills all bits on the LHS w/ 0
    - '1 fills all bits on the LHS w/ 1
    - 'z or 'Z fills all bits on the LHS w/ z
    - 'x or 'X fills all bits on the LHS w/ x
  - Important construct for modeling scalable designs (i.e. different vector widths for different design configurations)
- Floating-point literal values (real numbers):
  - Represented using 64-bit double precision floating point values
    - Examples: 3.1567, 5.0, 0.5
  - Real numbers not typically supported by synthesis tools

Vectors and Part Select

- Verilog net and variable data types more than one bit wide are called *vectors*
  - Examples:

```
wire [255:0] bus;           // 256-bit wide bus
tri [0:127] backward_bus;   // bit 0 is MSB
logic[31:24] whatever;      // don't need bit 0
```

- Select part of a vector
  - Examples:

```
my_bit = bus[152];
my_bit = bus[i];               // selects ith bit of bus
my_byte = bus[87:80];
my_byte = bus[80+:8];          // selects bits [87:80]
my_byte = bus[87-:16];         // selects bits [87:72]
my_byte = bus[byteNum*8-:8];   // variable part select
my_byte = bus[120:127];        // Illegal!
```

Vector Part Select

- Part select can include expressions of variables

```
my_byte = word[n*8+7:n*8]; //selects nth byte of word
```

- Part select increment and decrement function
  `[`*<starting_bit>*`+:`*<width>*`]` increments from starting_bit
  `[`*<starting_bit>*`-:`*<width>*`]` decrements from starting_bit

- Examples:

```
logic [127:0] vector_le;
logic [0:127] vector_be;
my_byte = vector_le[31-:8];  //selects vector_le[31:24]
my_byte = vector_le[24+:8];  //selects vector_le[31:24]
my_byte = vector_be[31-:8];  //selects vector_be[24:31]
my_byte = vector_be[24+:8];  //selects vector_be[24:31]
my_byte = vector_le[n*8+:8]; //selects nth byte of vector_le
my_byte = vector_be[n*8+:8]; //selects nth byte of vector_be
```

Source: Verilog 2001 A guide to the New Features of the Verilog Hardware Description Language by Stuart Sutherland

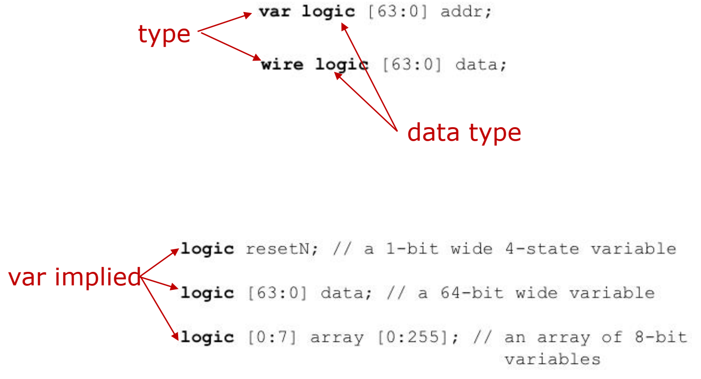
Types and Data types

- SystemVerilog introduces notion of type and data type
  - type indicates whether identifier is a net or a variable
  - data type indicates the value system (0/1 for two-state logic, 0/1/X/Z for four-state logic)
- Data types are used in RTL modeling to indicate desired silicon behavior
  - ex: should an adder performed signed or unsigned arithmetic
- Type:
  - Nets are used to connect design blocks together
    - Transfers data values from a source (driver) to a destination (receiver)
    - SystemVerilog supports several net types
    - Net types are always 4-state data types
  - Variables provide temporary storage for simulation
    - Required on LHS of procedural block assignments
    - Can be either 2-state or 4-state data types

Types and Data types (cont'd)

- type – `wire`, `wand`, … (net) or `var`
  - keyword for general purpose variable is `var`
  - When type is omitted from declaration `var` is implied
- data type – `logic` (4-state), `bit` (2-state), `reg` (4-state) , …
  - When data type is omitted logic is implied
  - `reg` maintains compatibility w/ Verilog but use logic for your designs

Types and Data types (cont'd)



```
var logic [63:0] addr;

wire logic [63:0] data;
```

```
logic resetN; // a 1-bit wide 4-state variable

logic [63:0] data; // a 64-bit wide variable

logic [0:7] array [0:255]; // an array of 8-bit
                              variables
```

Synthesizable variable data types

Type	Representation
`reg`	An obsolete general purpose 4-state variable of a user-defined vector size; equivalent to **`var logic`**
`logic`	Usually infers a general purpose **`var logic`** 4-state variable of a user-defined vector size, except on module input/inout ports, where **`wire logic`** is inferred
`integer`	A 32-bit 4-state variable; equivalent to **`var logic`** `[31:0]`
`bit`	A general purpose 2-state **`var`** variable with a user-defined vector size; defaults to a 1-bit size if no size is specified
`int`	A 32-bit 2-state variable; equivalent to **`var bit`** `[31:0]`; synthesis compilers treat **`int`** as the 4-state **`integer`** type
`byte`	An 8-bit 2-state variable; equivalent to **`var bit`** `[7:0]`
`shortint`	A 16-bit 2-state variable; equivalent to **`var bit`** `[15:0]`
`longint`	A 64-bit 2-state variable; equivalent to **`var bit`** `[63:0]`

Source: Sutherland, Table 3-1

SystemVerilog 2-State Variables

- Useful for modeling at higher levels of abstraction than RTL (e.g. system, transaction)
  - Higher level models seldom require z, x which take up more memory and are slower to simulate
- Most of the 2-state types are signed by default
  - Can override w/ the **unsigned** keyword

```
int s_int; // signed 32-bit variable
int unsigned u_int; // unsigned 32-bit variable
```

- Common uses:
  - Bus functional models
  - Interfacing Verilog models to C/C++ models using SystemVerilog Direct Programming I/F
  - Loop variables (aren't synthesized, don't require z, x states)

SystemVerilog 2-State Variables (cont'd)

- 2-state types:

bit	1-bit 2-state integer
byte	8-bit 2-state integer (similar to C **char**)
shortint	16-bit 2-state integer (similar to C **short**)
int	32-bit 2-state integer (similar to C **int**)
longint	64-bit 2-state integer (similar to C **longlong**)

- Synthesis treats 4-state and 2-state variables the same way
- 2-state data types are initialized to 0 in simulation but not in synthesized hardware
- 4-state variables can be assigned to 2-value data types but be careful…

SystemVerilog 2-State Variables (cont'd)

4-state variables can be assigned to 2-value data types but use care when connecting 2-state variables in testbench to design under test, especially outputs.

- 4-state variables driving x or z to a 2-state variable will be (silently) converted to 2-state value (0) and your testbench will never know
- Use `$isunknown()` which returns 1 if any bit of the expression is x or z.

```
if ($isunknown(iport) == 1)
  $display("@%0t: 4-state value detected on iport %b",
           $time, iport);
```

- Use `$countbits()` to count number of bits having a specified set of values (0, 1, x, z)

```
assert (!$isunknown(data)
  else $error (“data has %0d bits with x or z”, $countbits (data, 'x, 'z));
```

Variable Assignment Rules

- A variable can receive a value from any one of these ways:
  - Assigned a value from single `always_comb`, `always_ff`, or `always_latch` procedural block
  - Assigned a value from single continuous assignment statement
  - As the result of an assignment operator such as ++
  - As an input to a module, task or function
  - As a connection to an output port of a module instance, task or function instance or primitive (gate level) instance
- A variable can only be assigned by a single source, however multiple assignments to a variable in the same procedural block is treated as a single driver

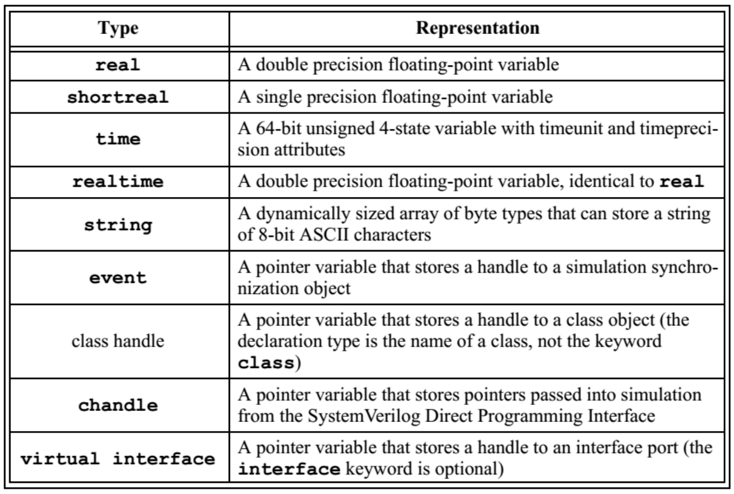
```
logic [15:0] q;  //16-bit 4-state unsigned variable

always_ff @(posedge clk)
  if (!rstN)
    q <= ‘0;
  else
    q <= d
```

Variable Assignment Rules (cont’d)

var(iables) cannot be driven by multiple sources but nets can

```
module add_and_increment (output logic [63:0] sum,
                          output logic        carry,
                          input  logic [63:0] a, b );
  always @(a, b)
    sum = a + b;          // procedural assignment to sum

  assign sum = sum + 1;   // ERROR! sum is already being
                          // assigned a value

  look_ahead i1 (carry, a, b);  // module instance drives carry

  overflow_check i2 (carry, a, b); // ERROR! 2nd driver of carry
endmodule

module look_ahead (output wire         carry,
                   input  logic [63:0] a, b);
  ...
endmodule

module overflow_check (output wire         carry,
                       input  logic [63:0] a, b);
  ...
endmodule
```

Non-synthesizable variable data types



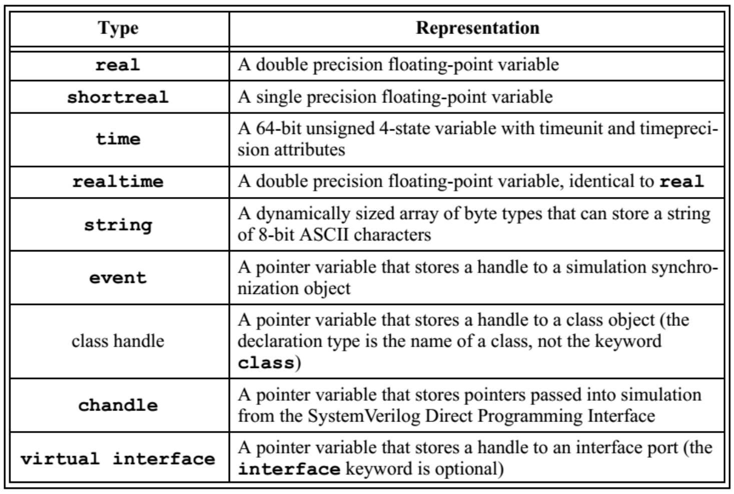
Type	Representation
`real`	A double precision floating-point variable
`shortreal`	A single precision floating-point variable
`time`	A 64-bit unsigned 4-state variable with timeunit and timeprecision attributes
`realtime`	A double precision floating-point variable, identical to **`real`**
`string`	A dynamically sized array of byte types that can store a string of 8-bit ASCII characters
`event`	A pointer variable that stores a handle to a simulation synchronization object
class handle	A pointer variable that stores a handle to a class object (the declaration type is the name of a class, not the keyword **`class`**)
`chandle`	A pointer variable that stores pointers passed into simulation from the SystemVerilog Direct Programming Interface
`virtual interface`	A pointer variable that stores a handle to an interface port (the **`interface`** keyword is optional)

Source: Sutherland, Table 3-2

Net types

- Used to connect design elements together
  - ex: connecting the output port of one module to the input port of another module
- Nets differ from variables in these significant ways:
  - Nets reflect the current value of the driver(s) on the net (i.e. no temporary storage)
  - Nets can resolve the results of multiple drivers
  - Nets reflect both a driver value (0, 1, Z, or X) and a driver strength
    - Strength is represented in steps from 0 (weak, low drive) to 7 (strong, high drive)
    - Default drive strength is 6 (strong)
    - Strengths are NOT used in RTL modeling (dependent on target technology)

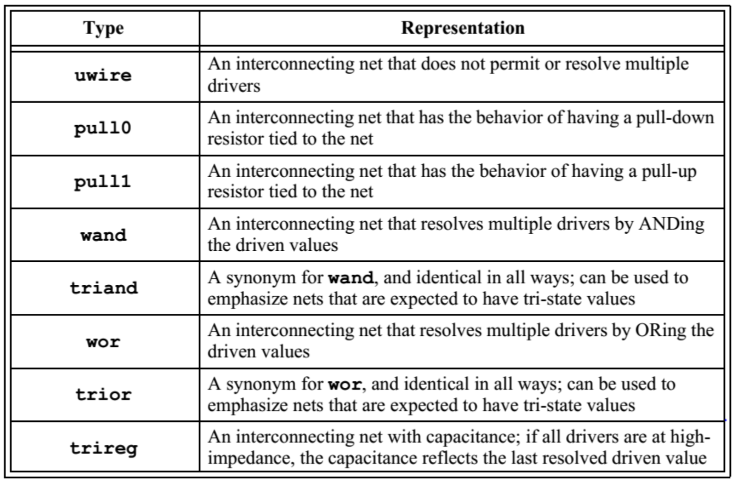
Synthesizable Net Types



Type	Representation
real	A double precision floating-point variable
shortreal	A single precision floating-point variable
time	A 64-bit unsigned 4-state variable with timeunit and timeprecision attributes
realtime	A double precision floating-point variable, identical to **real**
string	A dynamically sized array of byte types that can store a string of 8-bit ASCII characters
event	A pointer variable that stores a handle to a simulation synchronization object
class handle	A pointer variable that stores a handle to a class object (the declaration type is the name of a class, not the keyword **class**)
chandle	A pointer variable that stores pointers passed into simulation from the SystemVerilog Direct Programming Interface
virtual interface	A pointer variable that stores a handle to an interface port (the **interface** keyword is optional)

Source: Sutherland, Table 3-3

"Generally" Non-synthesizable Net Types



Type	Representation
uwire	An interconnecting net that does not permit or resolve multiple drivers
pull0	An interconnecting net that has the behavior of having a pull-down resistor tied to the net
pull1	An interconnecting net that has the behavior of having a pull-up resistor tied to the net
wand	An interconnecting net that resolves multiple drivers by ANDing the driven values
triand	A synonym for **wand**, and identical in all ways; can be used to emphasize nets that are expected to have tri-state values
wor	An interconnecting net that resolves multiple drivers by ORing the driven values
trior	A synonym for **wor**, and identical in all ways; can be used to emphasize nets that are expected to have tri-state values
trireg	An interconnecting net with capacitance; if all drivers are at high-impedance, the capacitance reflects the last resolved driven value

Source: Sutherland, Table 3-4

Net assignment and connection rules

- Nets can receive a value from two types of sources:
  - As a connection to an output or inout port
  - As the LHS of a continuous assignment (and assign statement)
- Any changes on the RHS of an assignment cause the LHS to be re-evaluated and LHS updated
  - LHS can be a variable or a net

```
wire [15:0] sum;
assign sum = a + b;
```

  - An implicit continuous assignment combines the declaration of a net and the assignment

```
wire [15:0] sum = a + b;
```

- Rules for resolving port/connection mismatches:
  - port size < net or variable size – leftmost bits truncated
  - Port size > net or variable size – value of net is left-extended
    - If either port or net/variable is unsized – zero-extended
    - If both port and net/variable are sighed – sign-extended

Next Time

- Topics:
  - Packed and unpacked arrays
  - Parameters and constants
  - User-defined types (time permitting)
- You should:
  - Read Sutherland Ch 3
- Homework, projects and quizzes
  - Homework #1 will be assigned Tue, 14-Apr