ECE 351
Verilog and FPGA Design

Week 3_2: **Parameters**
SystemVerilog packages
User-defined types

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Portland State UNIVERSITY

1

Homework #1

Write-up: ..\..\assignments\hw1\ece351s21_hw1_release_R1_0\ece351_hw1.pdf

Question 1: ..\..\assignments\hw1\ece351sp1_hw1a.txt

Stimulus: ..\..\assignments\hw1\ece351s21_hw1_release_R1_0\hdl\stimulus.sv

Testbench: ..\..\assignments\hw1\ece351s21_hw1_release_R1_0\hdl\tb_CLA4Bit.sv

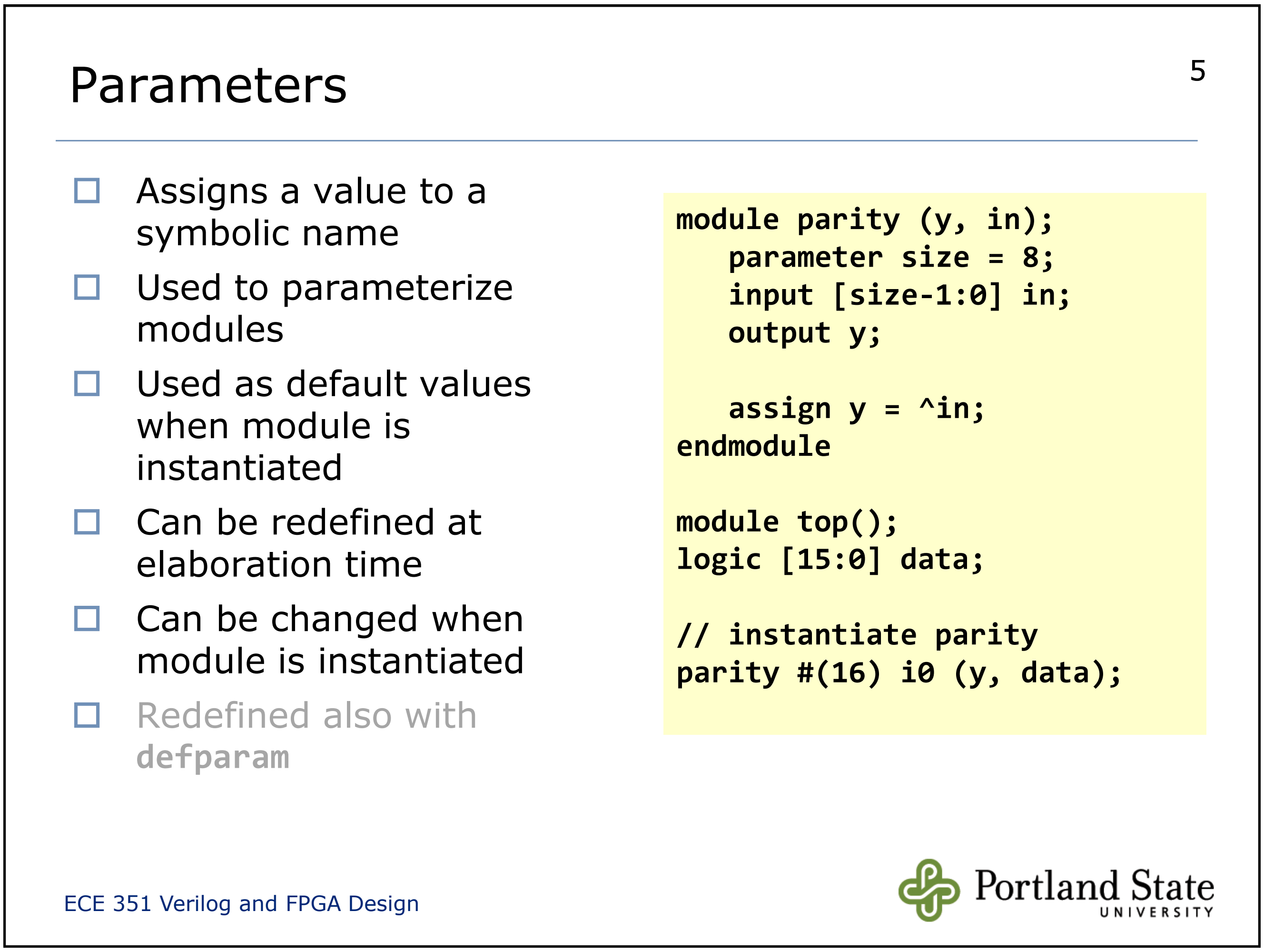
Ex: RTL model of 1-bit full adder

```
// single bit adder cell.  No carry out generated because the
// carry circuitry is implemented separately.  Almost too simple
// to make into a module but suggested for modularity

module cla_fa_1bit (
    input logic A, B,
    input logic CI,
    output logic S
);

    assign S = A ^ B ^ CI;

endmodule: cla_fa_1bit
```

Parameters

Source material drawn from:

- Mark F. And Roy K. ECE 571 lecture slides
- Roy's ECE 351 and Verilog workshop lecture slides
- *SystemVerilog for Design, 2nd Edition* by Stuart Sutherland
- RTL Modeling with SystemVerilog by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas



Parameters

- Assigns a value to a symbolic name
- Used to parameterize modules
- Used as default values when module is instantiated
- Can be redefined at elaboration time
- Can be changed when module is instantiated
- Redefined also with `defparam`

```
module parity (y, in);
   parameter size = 8;
   input [size-1:0] in;
   output y;

   assign y = ^in;
endmodule

module top();
logic [15:0] data;

// instantiate parity
parity #(16) i0 (y, data);
```

Portland State UNIVERSITY

Parameters (cont'd)

- Parameters included in module port list:

```
module adder #(parameter MSB = 32, LSB = 0)
             (output logic [MSB:LSB] sum;
              output logic co;
              input wire [MSB:LSB] a, b;
              input wire           ci;
              ...
```

- Parameters can (and should be) be sized and typed

```
parameter [31:0] A;        // unsigned 32-bit parameter
parameter signed [63:0] B; // signed 64-bit parameter
```

Portland State UNIVERSITY

Local parameters

- True constant
  - Cannot be redefined at elaboration time
  - Cannot be redefined outside the module

```
module paramram(d, a, wr, q);
   parameter D_WIDTH = 8, A_WIDTH = 8;
   localparam DEPTH = (2**A_WIDTH)-1;
   input [D_WIDTH-1:0] d;
   input [A_WIDTH-1:0] a;
   input wr;
   output wire [D_WIDTH-1:0] q;

   logic [D_WIDTH-1:0] mem [DEPTH:0]; // declare memory array

   always @(wr) mem[a] = d; // write
   assign q = mem[a]; //read
endmodule
```

Portland State UNIVERSITY

`define

- `define
  - defines a global text substitution
  - Fixed at elaboration time – cannot be redefined

```
`define size 8

Logic [`size - 1:0] a, b;

always @(posedge clk)
   a <= b;
```

This is NOT the same as a parameter…if you're familiar w/ C it'd be better to think this way:

- parameter <-> const
- `define <-> #define

Portland State UNIVERSITY

SystemVerilog Packages

Source material drawn from:
- Mark Faust and Roy Kravitz ECE 571 lecture slides
- *SystemVerilog for Design, 2nd Edition* by Stuart Sutherland

Declaration spaces

Limitations of Verilog 2001:

- Variables, nets, tasks, functions must be declared in a module
  - Modeling: must be used only within the module where declared
  - Verification: can use hierarchical references into other modules
    - Don’t represent hardware behavior
    - Aren’t synthesizable
- No place for global declarations (ex: global functions and tasks)
  - Declarations used in multiple blocks must be declared in each block
  - Results in redundancy (and therefore source of errors when changes made)
  - `` `include `` files only a partial solution

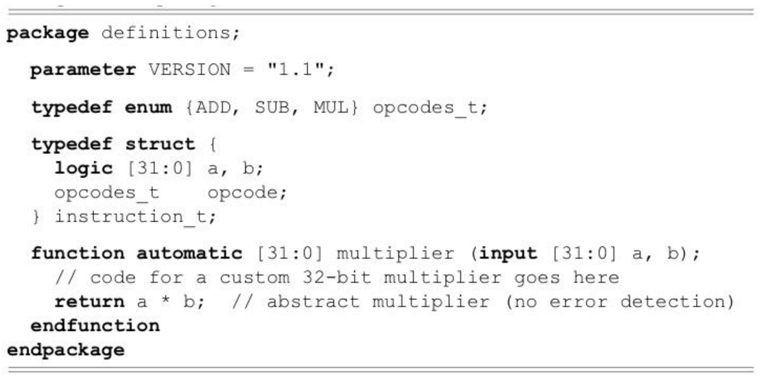
SystemVerilog specification includes packages:

- Concept borrowed from VHDL (Java, OOP languages, …)
- `package..endpackage` keywords

Portland State UNIVERSITY

Packages

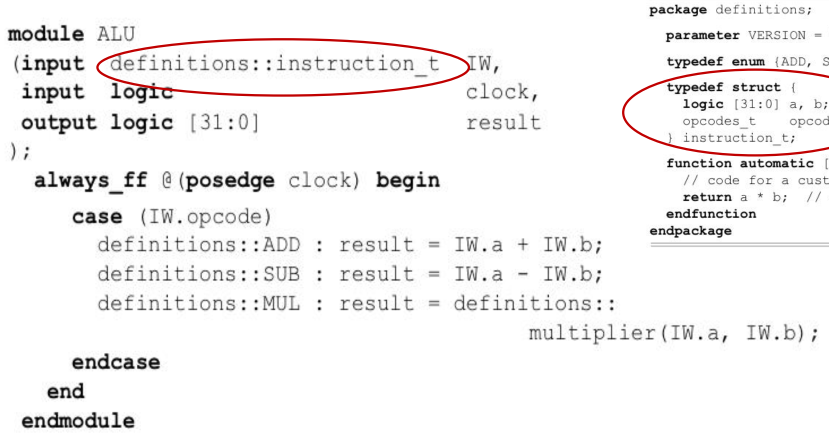
- A SystemVerilog package can include the following synthesizable constructs:
  - parameter and localparam constant definitions
  - const variable definitions
  - typedef user-defined types
  - Fully automatic task and function definitions
  - import statements from other packages
  - Operator overload definitions



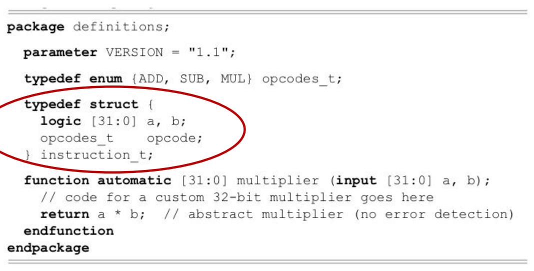
```
package definitions;
  parameter VERSION = "1.1";
  typedef enum {ADD, SUB, MUL} opcodes_t;
  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;
  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

Referencing package contents

- Modules and interfaces can reference definitions and declarations in a package
  - Direct reference using a scope resolution operator
  - Import specific package items into the module or interface
  - Wildcard import package items into the module or interface



```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  always_ff @(posedge clock) begin
     case (IW.opcode)
       definitions::ADD : result = IW.a + IW.b;
       definitions::SUB : result = IW.a - IW.b;
       definitions::MUL : result = definitions::
                                     multiplier(IW.a, IW.b);
     endcase
   end
 endmodule
```



```
package definitions;
  parameter VERSION = "1.1";
  typedef enum {ADD, SUB, MUL} opcodes_t;
  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;
  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

Portland State UNIVERSITY

Importing SystemVerilog packages

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

```
module ALU
 (input  definitions::instruction_t  IW,
  input  logic                       clock,
  output logic [31:0]                result
 );
   import definitions::ADD;
   import definitions::SUB;
   import definitions::MUL;
   import definitions::multiplier;

   always_comb begin
     case (IW.opcode)
       ADD : result = IW.a + IW.b;
       SUB : result = IW.a - IW.b;
       MUL : result = multiplier(IW.a, IW.b);
     endcase
   end
endmodule
```

Caution: Importing an enumerated type definition doesn't import the labels used in the definition

Importing SystemVerilog packages(cont’d)

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

Why?

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  import definitions::*;  // wildcard import

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

imports all definitions

Portland State UNIVERSITY

Importing SystemVerilog packages(cont'd)

…Employ C Programming trick

```
`ifndef DEFS_DONE  // if the already-compiled flag is not set...
  `define DEFS_DONE  // set the flag
  package definitions;

    parameter VERSION = "1.1";

    typedef enum {ADD, SUB, MUL} opcodes_t;

    typedef struct {
      logic [31:0] a, b;
      opcodes_t    opcode;
    } instruction_t;

    function automatic [31:0] multiplier (input [31:0] a, b);
      // code for a custom 32-bit multiplier goes here
      return a * b;  // abstract multiplier (no error detection)
    endfunction
  endpackage

  import definitions::*;  // import package into $unit

`endif
```

```
'include "definitions.pkg"
```

Portland State UNIVERSITY

Importing SystemVerilog packages(cont’d)

```
`include "definitions.pkg" // compile the package file

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a,
    endcase
  end
endmodule
```

```
`include "definitions.pkg" // compile the package file

module test;
  instruction_t test_word;
  logic [31:0]  alu_out;
  logic         clock = 0;

  ALU dut (.IW(test_word), .result(alu_out), .clock(clock));

  always #10 clock = ~clock;

  initial begin
    @(negedge clock)
    test_word.a = 5;
    test_word.b = 7;
    test_word.opcode = ADD;
    @(negedge clock)
    $display("alu_out = %d (expected 12)", alu_out);
    $finish;
  end
endmodule
```

Portland State UNIVERSITY

SystemVerilog package “gotchas”

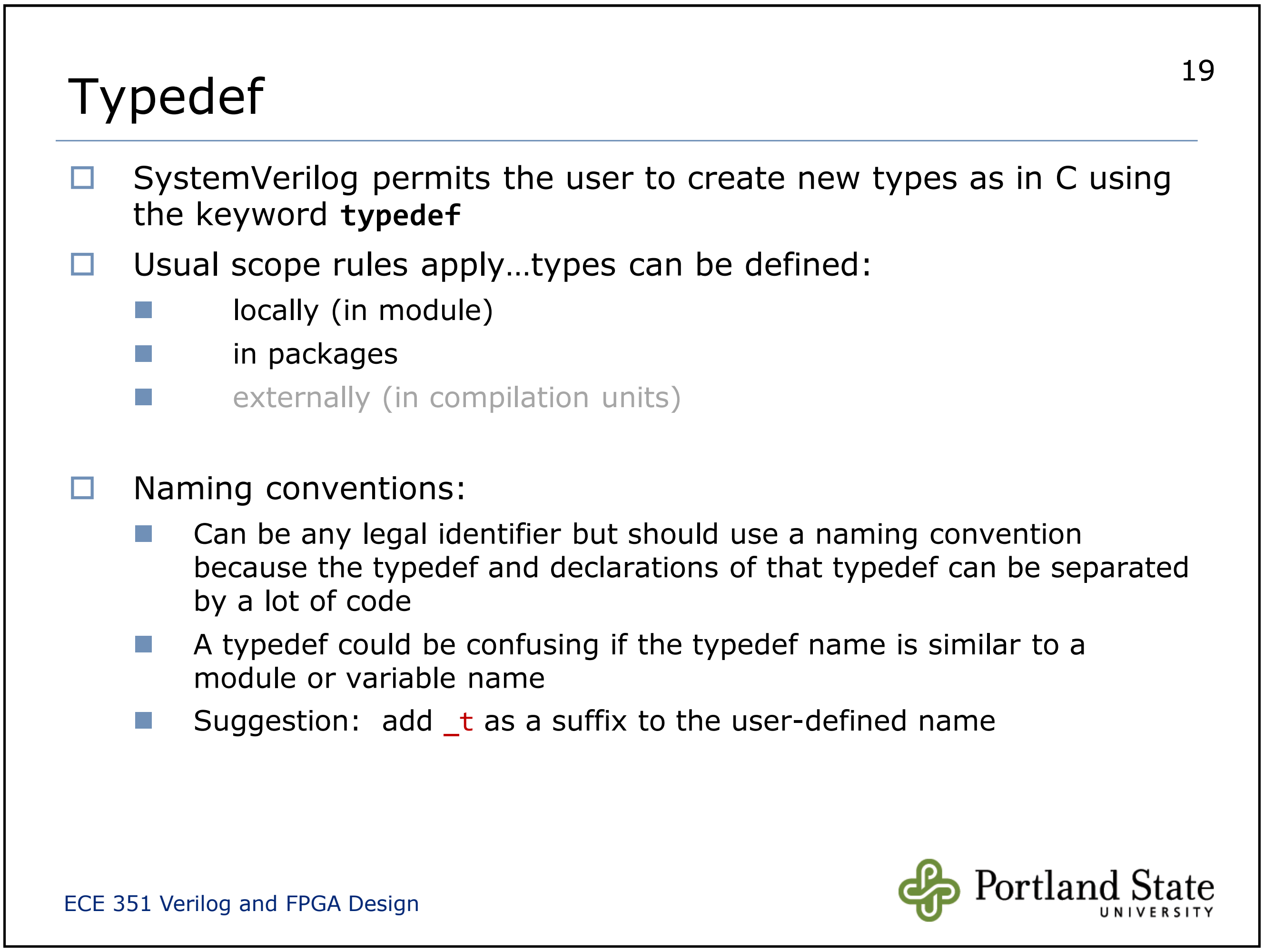
- Variables in packages
  - In simulation, value of variable will be shared among all modules importing the variable
  - Not synthesizable (must use module ports to communicate)
- To be synthesizable, all tasks and functions must be declared automatic and not contain static variables
  - Storage for automatics allocated at time called
  - Each module referencing a task/function’s sees unique copy
    - No sharing of storage
  - Tasks and functions defined in a package will be duplicated and treated as though defined in any module that references them

Portland State UNIVERSITY

User defined types - Typedefs

Source material drawn from:

- Mark F. and Roy K. ECE 571 lecture slides
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas



Typedef

- SystemVerilog permits the user to create new types as in C using the keyword **`typedef`**
- Usual scope rules apply…types can be defined:
  - locally (in module)
  - in packages
  - externally (in compilation units)
- Naming conventions:
  - Can be any legal identifier but should use a naming convention because the typedef and declarations of that typedef can be separated by a lot of code
  - A typedef could be confusing if the typedef name is similar to a module or variable name
  - Suggestion: add `_t` as a suffix to the user-defined name

Typedef example

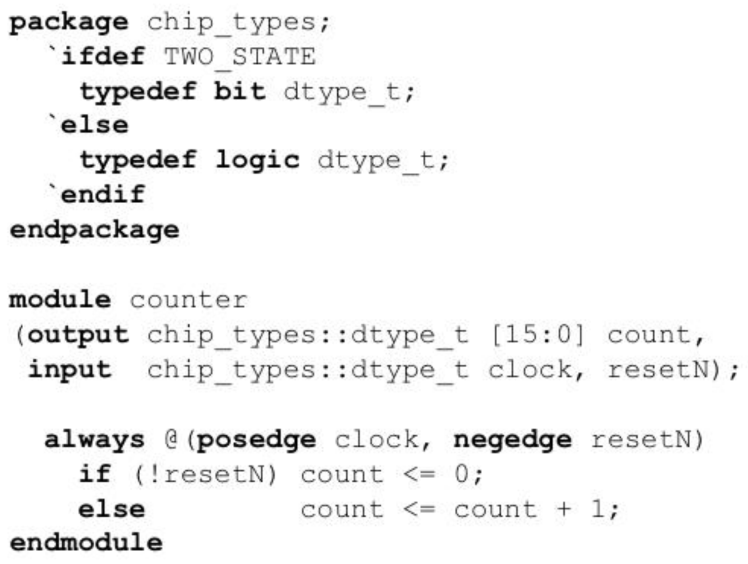
```
typedef int unsigned uint_t;
typedef logic [3:0] nibble_t;

uint_t a, b;                 // two variables of type uint_t

nibble_t opA, opB;           // two variables of the nibble_t type
wire nibble_t [7:0] data;    // 32-bit net comprising 8 nibble_t
```

Portland State UNIVERSITY

Source: Sutherland; p.133 (Kindle Editon)

Sharing types

To share a user-defined type put the **`typedef`** in a package



```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

module counter
(output chip_types::dtype_t [15:0] count,
 input  chip_types::dtype_t clock, resetN);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;
    else         count <= count + 1;
endmodule
```

User Defined Types – Enumerated Types

Source material drawn from:
- Mark F. and Roy K. ECE 571 lecture slides
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

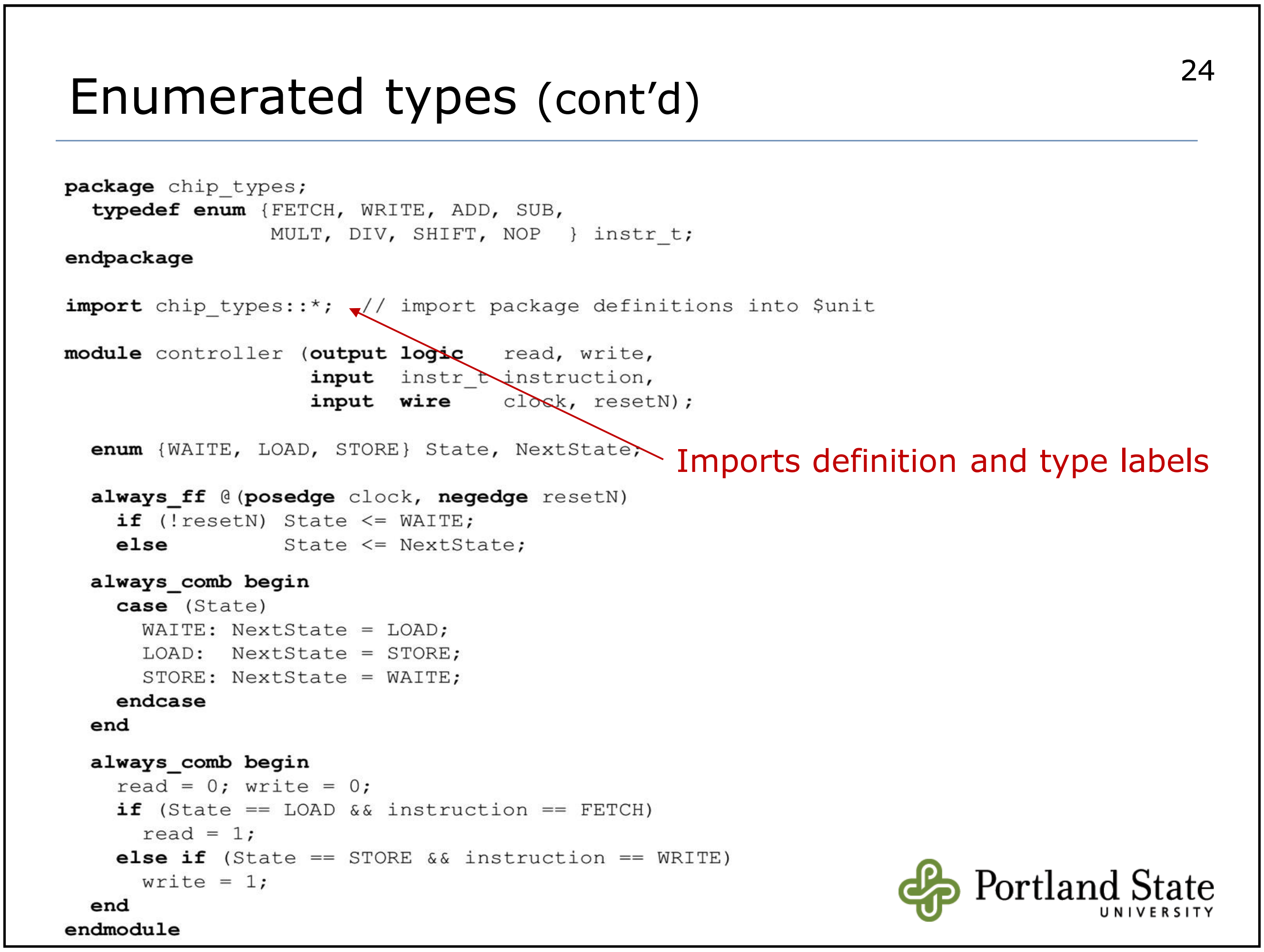
Enumerated types

- Verilog 2001 required use of **`` `define ``** or **`parameter`** (no error checking on assignments) to name states, opcodes, etc.
- SystemVerilog allows you to declare a type with an explicit list of valid values: `enum {red,green,blue} RGB;`

```
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg        read, write,
                   input  wire [2:0] instruction,
                   input  wire       clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;
```

```
  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always @(State) begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always @(State, instruction) begin
    read = 0; write = 0;
    if (State == LOAD && instruction == `FETCH)
      read = 1;
    else if (State == STORE && instruction == `WRITE)
      write = 1;
  end
endmodule
```

Portland State UNIVERSITY



Enumerated types (cont'd)

```
package chip_types;
  typedef enum {FETCH, WRITE, ADD, SUB,
                MULT, DIV, SHIFT, NOP  } instr_t;
endpackage

import chip_types::*;  // import package definitions into $unit

module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  wire    clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always_comb begin
    case (State)
      WAITE: NextState = LOAD;
      LOAD:  NextState = STORE;
      STORE: NextState = WAITE;
    endcase
  end

  always_comb begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)
      read = 1;
    else if (State == STORE && instruction == WRITE)
      write = 1;
  end
endmodule
```

Imports definition and type labels

Portland State UNIVERSITY

Enumerated types (cont'd)

- SystemVerilog provides shorthand for specifying ranges of labels

```
enum {RESET, S[5], W[6:9]} state;
```

Creates an enumerated list with the labels `RESET`, `S0..S4`, `W6..W9`

- Labels must be unique within their scope (e.g. module, begin…end block, compilation unit, interfaces…)

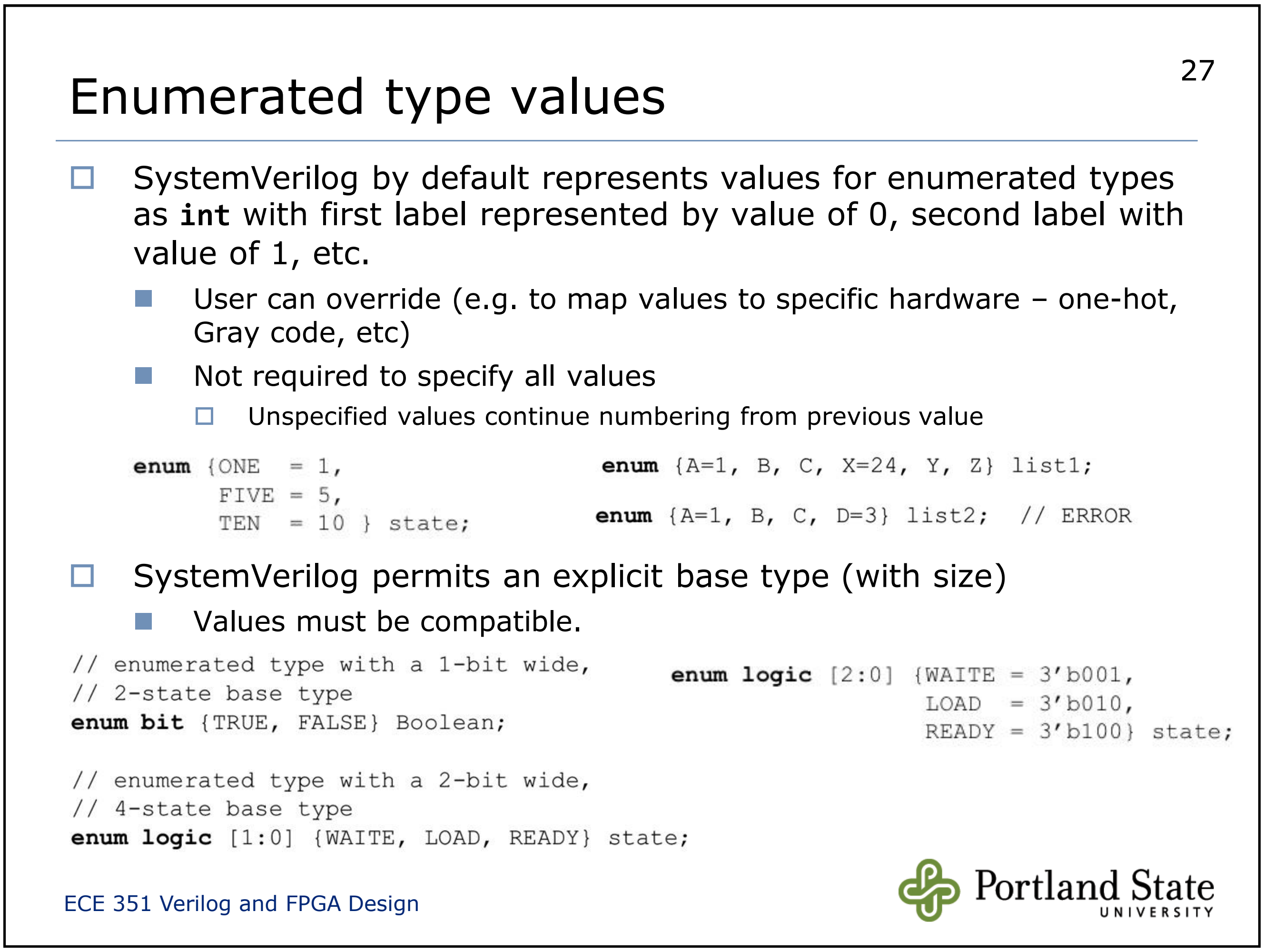
```
module FSM (...);
  ...
  always @(posedge clock)
    begin: fsm1
      enum {STOP, GO} fsm1_state;
      ...
    end

  always @(posedge clock)
    begin: fsm2
      enum {WAITE, GO, DONE} fsm2_state;
      ...
    end
  ...
```

Same label, different scope

```
module FSM (...);
  enum {GO, STOP} fsm1_state;
  ...
  enum {WAITE, GO, DONE} fsm2_state;   // ERROR
  ...
```

Same label, same scope

Portland State UNIVERSITY



Enumerated type values

- SystemVerilog by default represents values for enumerated types as **int** with first label represented by value of 0, second label with value of 1, etc.
  - User can override (e.g. to map values to specific hardware – one-hot, Gray code, etc)
  - Not required to specify all values
    - Unspecified values continue numbering from previous value

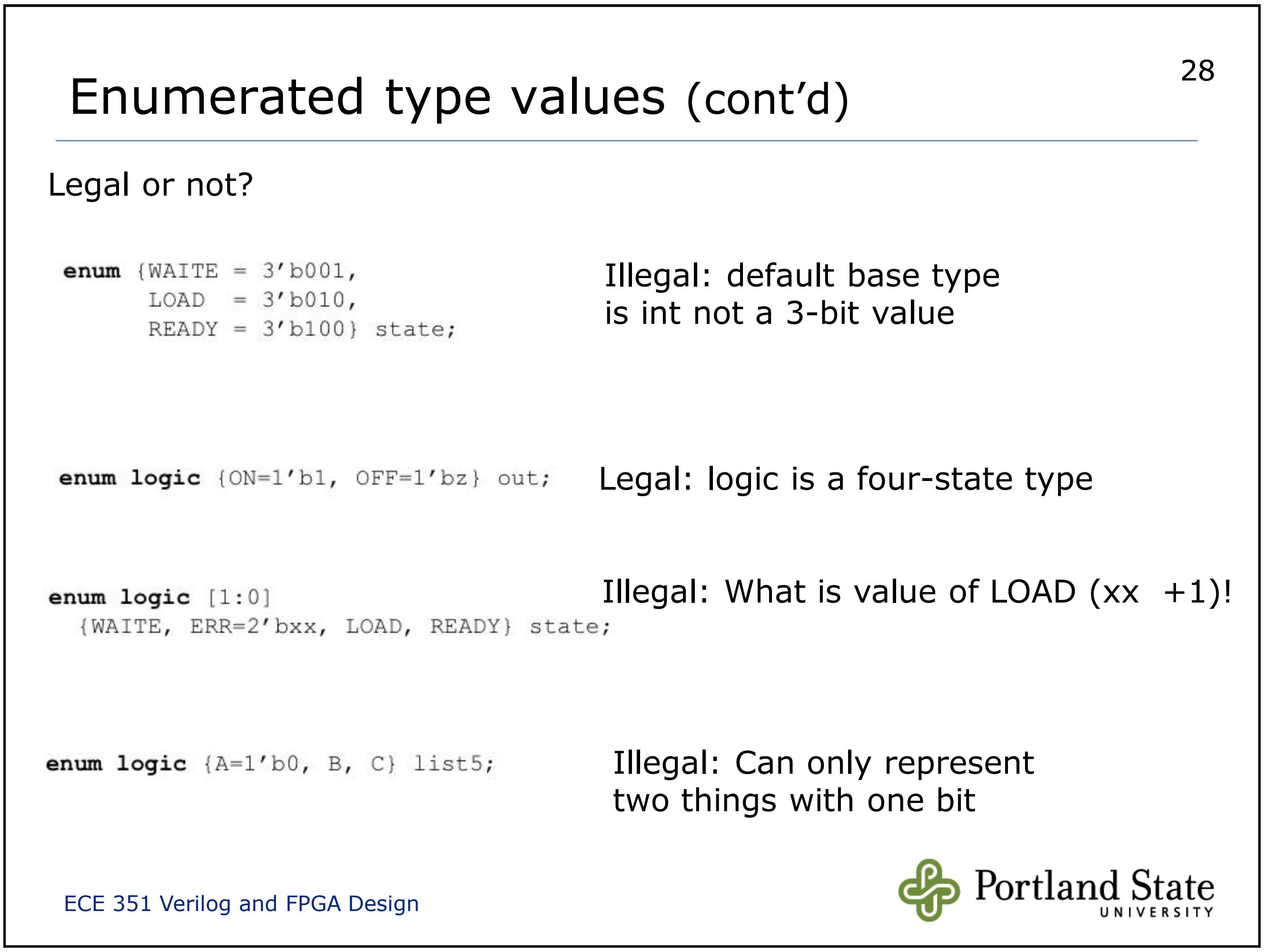
```
enum {ONE  = 1,
      FIVE = 5,
      TEN  = 10 } state;
```

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

enum {A=1, B, C, D=3} list2;  // ERROR
```

- SystemVerilog permits an explicit base type (with size)
  - Values must be compatible.

```
// enumerated type with a 1-bit wide,
// 2-state base type
enum bit {TRUE, FALSE} Boolean;

// enumerated type with a 2-bit wide,
// 4-state base type
enum logic [1:0] {WAITE, LOAD, READY} state;
```

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



Enumerated type values (cont’d)

Legal or not?

```
enum {WAITE = 3’b001,
      LOAD  = 3’b010,
      READY = 3’b100} state;
```

Illegal: default base type
is int not a 3-bit value

```
enum logic {ON=1’b1, OFF=1’bz} out;
```

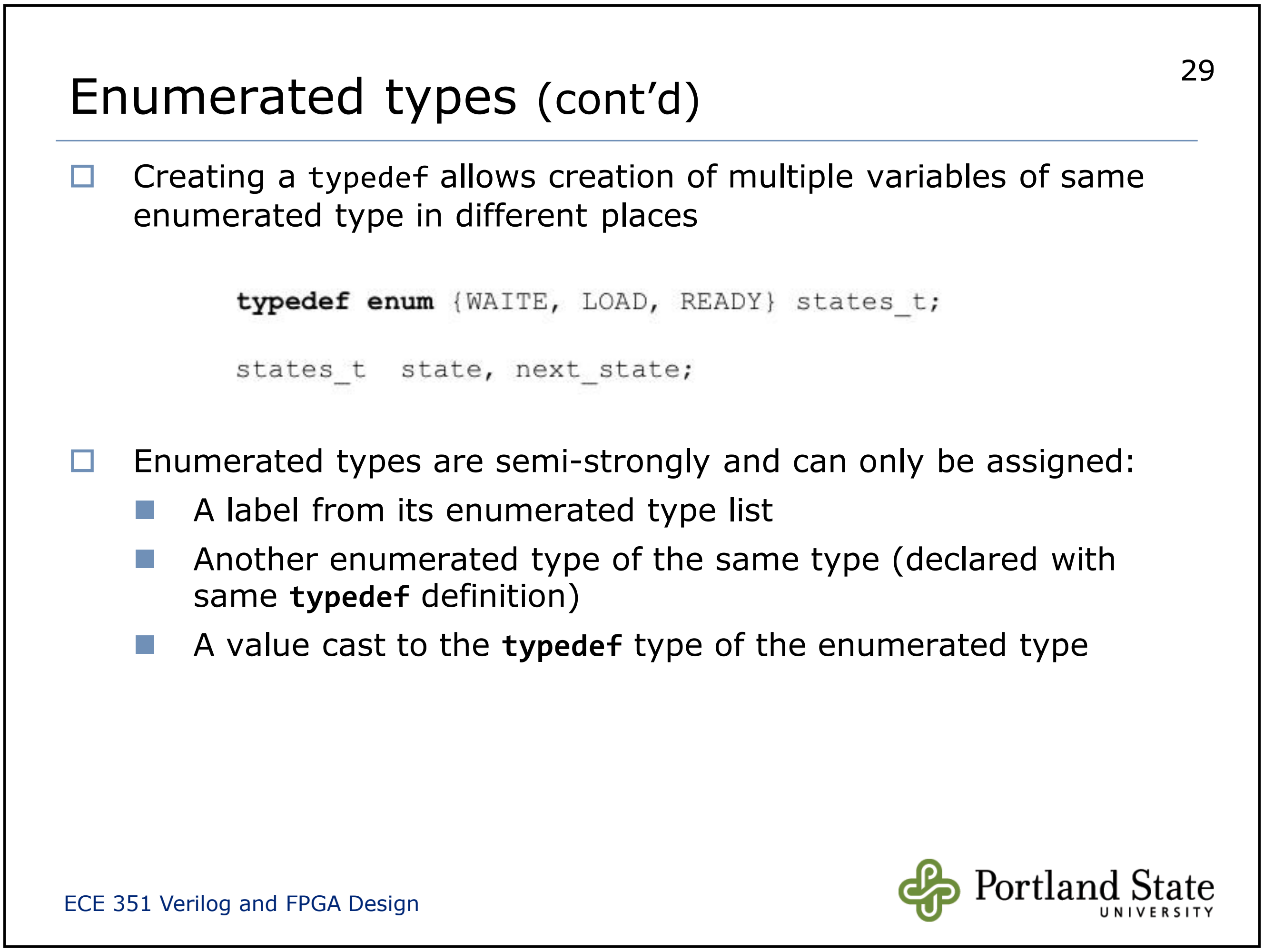
Legal: logic is a four-state type

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

Illegal: What is value of LOAD (xx +1)!

```
enum logic {A=1’b0, B, C} list5;
```

Illegal: Can only represent
two things with one bit



Enumerated types (cont'd)

- Creating a `typedef` allows creation of multiple variables of same enumerated type in different places

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;
```

- Enumerated types are semi-strongly and can only be assigned:
  - A label from its enumerated type list
  - Another enumerated type of the same type (declared with same **`typedef`** definition)
  - A value cast to the **`typedef`** type of the enumerated type

Portland State UNIVERSITY

Enumerated types (cont’d)

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;
int foo;
```

```
state = next_state;      // Legal: both variables of same enumerated type
foo = state + 1;         // Legal: foo is of type int, underlying enum is int
state = foo + 1;         // Illegal: foo is type int, foo+1 is int not enum type
state = state + 1;       // Illegal: (state + 1) type is int not enum type
state++;                 // Illegal: (state + 1) type is int not enum type
next_state += state;     // Illegal: (state + 1) type is int not enum type
```

Portland State UNIVERSITY

System tasks & methods for enumerated types

SystemVerilog provides several special system functions called methods to iterate through the values in an enumerated type list

```
<enum_var>.first      // Return value of first member in enumerated
                      // list of var
<enum_var>.last       // Return value of last member in enumerated
                      // list of var
<enum_var>.next(<N>)  // Return value of next member in enumerated
                      // list.  If N provided return Nth next member
<enum_var>.prev(<N>)  // Return value of previous member in enumerated
                      // list.If N provided return Nth previous member
<enum_var>.num        // Return the number of labels in the enumerated
                      // list of var
<enum_var>.name       // Return string representation of label for
                      // value
```

Parameterized Types

```
module adder #(parameter type dtype = logic [0:0]) // default is 1-bit size
(
        input dtype a, b,
        output dtype sum
);

assign sum = a + b;

endmodule

module top (
        input logic [15:0] a, b,
        input logic [31:0] c, d,
        output logic [15:0] r1,
        output logic [31:0] r2
);

adder #(.dtype(logic [15:0])) i1 (a, b, r1); // 16 bit adder
adder #(.dtype(logic signed [31:0])) i2 (c, d, r2); // 32-bit signed adder

endmodule
```

Next Time

- Topics:
  - User-defined types (wrap-up)
  - RTL expression operators
- You should:
  - Read Sutherland Ch 5
- Homework, projects and quizzes
  - Homework #1 has been posted. Should be submitted to D2L by 10:00 PM on Mon, 26-Apr
    - No late assignments accepted after Noon on Thu, 30-Apr
  - Next in-class exercise currently scheduled for Thu, 22-Apr