ECE 351
Verilog and FPGA Design

Lecture 7: User-defined types (wrap-up)
Packages
RTL expression operations

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

Review: 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
```

Review: 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

Using Typedefs w/ enum(s)

- 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

Using Typedefs w/ enum(s) (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
```

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
```

User Defined Types – Struct(ures) and Unions

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

Struct(ure)s

```
typedef enum {Request, Response, Broadcast} PacketType;

typedef struct {
       int ID;
       PacketType Type;
       int CheckSum;
       byte Data[1024];
       } Packet_t;

Packet_t SamplePacket;


SamplePacket.ID = 0;
SamplePacket.Type = Request;
.
.
.
```

Packed Structs

- Packed structs give you more control over how bits are laid out in memory

```
typedef struct {bit [7:0] r, g, b;} pixel_s;
pixel_s my_pixel;
```

Consumes 3 words

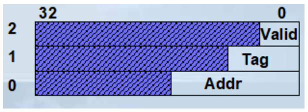
```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
pixel_p_s my_pixel;
```

Consumes 3 bytes

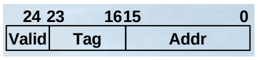
Packed Structs (cont'd)

```
reg [24:0] Entry;
`define Valid 24
`define Tag 23:16
`define Addr 15:0
iTag   = Entry[`Tag];
iAddr  = Entry[`Addr];
iValid = Entry[`Valid]
```

```
struct packed {
  bit         Valid;
  byte        Tag;
  bit [15:0]  Addr;
} Entry;
iTag   = Entry.Tag;
iAddr  = Entry.Addr;
iValid = Entry.Valid
```



unpacked



packed

Initializing Structures

```
initial begin
  typedef struct {int a;
                  byte b;
                  shortint c;
                  int d;} my_struct_s;
  my_struct_s st = '{32'haaaa_aaaad,
                     8'hbb,
                     16'hcccc,
                     32'hdddd_dddd};

  $display("str = %x %x %x %x ", st.a, st.b, st.c, st.d);
end
```

Unions

- Stores several types (mutually exclusive fields) in the same bits

```
typedef union { int i; real f; } num_u;
num_u un;
un.f = 0.0; // set value in floating point format
```

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
```

Summary

- `typedef`, `enum`, `struct`, `union` were added to SystemVerilog (i.e. did not exist in Verilog 2001
- Modeled after same constructs in C
- User-defined types (`typedef`)
  - Allow users to define new types built from predefined types or other user-defined types
  - Can be used as module ports and passed in or out of tasks and functions
- Enumerated types (`enum`)
  - Allow the declaration of variables with a define set of values (represented by abstract labels instead of hardware-centrid logic values
  - Allow modeling at a more abstract level than Verilog 2001
  - Values of labels can be specified (ex: one-hot encoding)
  - Default type is int…should always size and type your enums
  - Are strongly typed compared to other SystemVerilog constructs
  - Have a set of functions to get at elements of the enum, etc.

Summary (cont'd)

- Structures (struct)
  - Make it possible to bundle several variables together and work w/ the complete bundle while still being able to work w/ the individual variables
  - Structs can be copied, assigned list of values, passed through module ports, and passed in and out of functions and tasks
  - Can be packed or unpacked…better to use packed when you can
- Unions (`union`)
  - Give high-level coding style for modeling shared resources in a design
  - Ex: a register that can store different types of data at different times
  - Can be packed or unpacked. Use packed if based on packed struct(s)

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

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
```

Portland State UNIVERSITY

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
```

Importing SystemVerilog packages

```
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
```

```
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
```

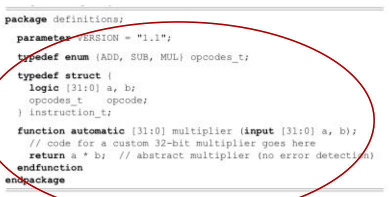
Caution: Importing an enumerated type definition doesn't import the labels used in the definition

Importing SystemVerilog packages(cont'd)

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



```
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(cont'd)

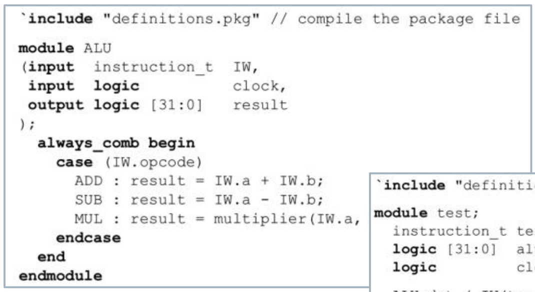
…Employ C Programming trick

```
`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
```

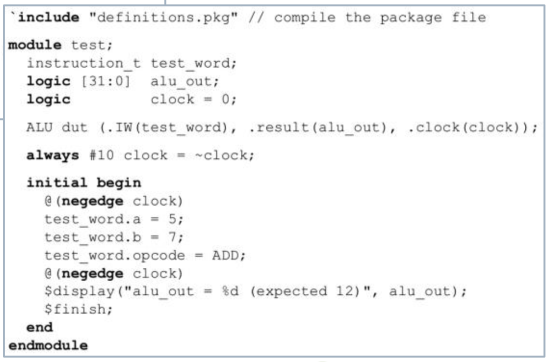
```
'include "definitions.pkg"
```

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
```

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

RTL Expression Operators

Source material drawn from:
- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Expressions Operators and Operands

- RTL and behavioral modeling describe a design in expressions instead of primitive gates
- Expressions are constructs that combine operators and operands to produce a result
  - ex: `a ^ b;  addr1[20:17] + addr2[20:17]; in1 | in2;`
- Operands can be any of the SystemVerilog data types
  - constants, integers, real numbers, wires, logic, vectors or parts of vectors, function calls
- Operators act on operands to produce desired results
  - `count + 1; a && b; !P1; reg[3:0] >> 2;`
- 2-state and 4-state expressions
  - When any of the operand is a 4-state expression the result of the operation will be a 4-state expression
  - All operands must be 2-state expressions for an expression to have a 2-state result
  - Recommended Guideline: Only use 4-state types for RTL modeling

X-optimization and X-pessimism

- Most SystemVerilog operators are X-optimistic…will a produce a known result even if there are X or Z values in the operands

```
logic [3:0] a, b, c, result;
assign a = 4’b01zx;      // some bits are X or Z
assign b = 4’b0000;      // all bits are zero
assign c = 4’b1111;      // all bits are one

assign result = a & b;   // result is 4’b0000 because & models a
                         // AND gate and 0 ANDed w/ any value is 0
assign result = a & c;   // result is 4’b01xx because 1 ANDed w/
                         // any value could result in either a 0
                         //  or a 1
```

X-optimization and X-pessimism (cont’d)

- Arithmetic and relational operators are X-pessimistic…will produce an X result if any operand has any bit w/ an X or Z value

```
logic [3:0] a, b, c, result;
assign a = 4’b000x;      // some bits are X or Z
assign b = 4’b0101;      // all bits are zero
assign c = 4’b1111;      // value of 5 in decimal

assign result = a + b;   // result is 4’bxxxx because + adds
                         // numbers and 4’b000x is not a number
```

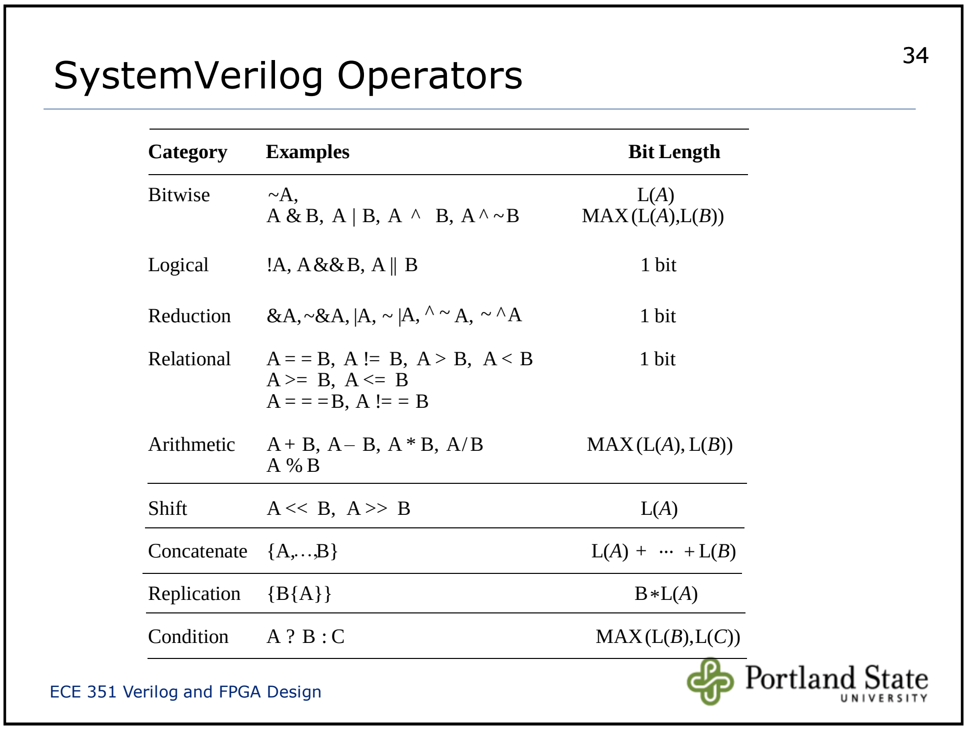
Vector sizes

- Self-determined operands – operators treat each operand independently even if the vectors are different sizes
  - ex: && does a logical which tests to see if both operands are true – each operand can be evaluate to be true or false independent of the vector size of other operand
- Context-determined operands – operators need to expand the operands to be the same vector size before the operation is performed
  - Operation will left-extend the shortest operand to be the same vector size as the largest operand
  - ex: & does a bitwise AND and returns a Boolean (1'b1 or 1'b0) result of each bit
  - Rules:
    - If leftmost bit is 0 or 1 and operand is unsigned than zero-extend
    - If leftmost bit is Z than operand is Z-extended
    - If leftmost bit is X than operand is X-extended

Vector sizes (cont'd)

- Signed and unsigned expressions
  - If all operands are signed than a signed operation ins performed
  - If any of the operands are unsigned than an unsigned operation is performed
- Integer (vector) and real (floating-point)
  - Operations can be performed on mix of integer and real expressions
  - Rule: If any of the operands is a real expression than the other operand is converted to a real-expression and a floating-point operation is performed

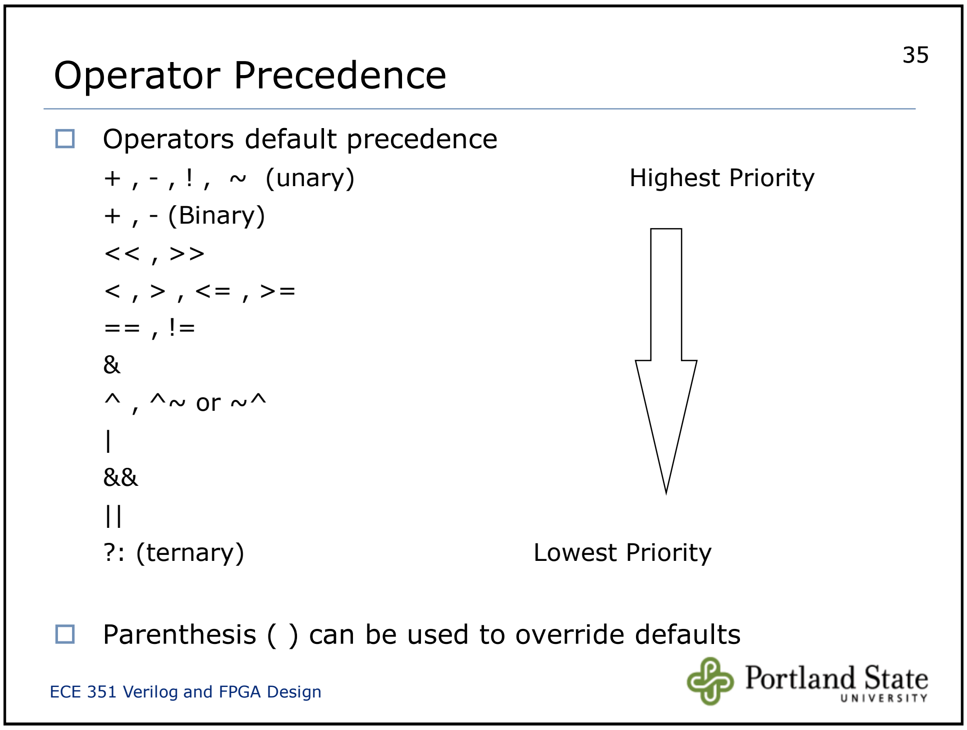
RTL Synthesis compilers typically do not support real (floating-point) expressions



SystemVerilog Operators

Category	Examples	Bit Length
Bitwise	~A, A & B, A | B, A ^ B, A ^ ~B	L(*A*) MAX (L(*A*),L(*B*))
Logical	!A, A && B, A || B	1 bit
Reduction	&A, ~&A, |A, ~ |A, ^ ~ A, ~ ^A	1 bit
Relational	A = = B, A != B, A > B, A < B A >= B, A <= B A = = =B, A != = B	1 bit
Arithmetic	A + B, A – B, A * B, A / B A % B	MAX (L(*A*), L(*B*))
Shift	A << B, A >> B	L(*A*)
Concatenate	{A,…,B}	L(*A*) + ··· + L(*B*)
Replication	{B{A}}	B∗L(*A*)
Condition	A ? B : C	MAX (L(*B*),L(*C*))

Portland State UNIVERSITY



Operator Precedence

- Operators default precedence

+ , - , ! , ~ (unary)	Highest Priority
+ , - (Binary)	
<< , >>	
< , > , <= , >=	
== , !=	
&	
^ , ^~ or ~^	
|	
&&	
||	
?: (ternary)	Lowest Priority

- Parenthesis ( ) can be used to override defaults

Portland State UNIVERSITY

Concatenation and replication

Operator	Example Usage	Description
{ }	{m,n}	Join m and n together as a vector
{r{ }}	{r{m,n}}	Join m and n together, and replicate r times; r must be a literal integer value. it cannot be a parameter.

Sutherland: Table 5.1 & Example 5.1

Concatenation and replication (cont'd)

- Converse of bit/part select:
  - `logic[9:0] CBus;`
  - `assign Cbus[7:0] = {Apart, 3'b000, Abit};`
- Can be used to replicate:
  - `Logic [9:0] CBus;`
  - `assign Cbus = { 6{Abit}, 3'b000};`
- The operands may be scalar nets or registers, vector nets or registers, bit-select, part-select or <u>sized</u> constants
  - <u>Must</u> be sized because the size of each operand must be known for computation of the size of the result
- Are synthesizable
- Do not directly add hardware but simply combine, append, or break apart vectors

Concatenation and replication (cont'd)

```
//
// 8-bit status register that stores multiple input values //
// -----------------------------------------------------------
// | int_en | unused | unused | zero | carry | neg | parity |
// -----------------------------------------------------------
// NOTE: not-used bits are set to a constant 1
//
module status_reg (
         input logic clk,                    // register clock
         input logic rstN,                   // active-low reset
         input logic int_en,                 // 1-bit interrupt enable
         input logic zero,                   // 1-bit result = 0 flag
         input logic carry,                  // 1-bit result overflow flag
         input logic neg,                    // 1-bit negative result flag
         input logic [1:0] parity,   // 2-bit parity bits
         output logic [7:0] status   // 8-bit status register output
);

always_ff @(posedge clk or negedge rstN)              // async reset
         if (!rstN)                                   // active-low reset
                  status <= {1'b0,2'b11,5'b0};        // reset
         else
                  status <= {int_en,2'b11,zero,carry,neg,parity};        // load
endmodule: status_reg
```

Sutherland: Table 5.1 & Example 5.1

Conditional Operator ?:

- Implies a multiplexer

```
module cond (sel, a, b, y);
   input sel, a, b;
   output y;
   assign y = sel ? a : b;
endmodule
```

- Can be nested:

```
input s1, s0;
input i3, i2, i1, i0;

//4-to-1 multiplexer
assign y = s1?(s0?i3:i2)
             :(s0?i1:i0);
```

- Can model a tri-state driver:

```
module tri_buf (en, a, y);
   input en, a;
   output y;
   assign y = en ? a : 1'bz;
endmodule
```

Bitwise Operators

Operator	**Operation**	**Examples** `ain = 3'b101, bin = 3'b110, cin = 3'b01x`
~	invert each bit	`~ain is 3'b010`
&	and each bit	`ain & bin is 3'b100, bin & cin is 3'b010`
|	or each bit	`ain | bin is 3'b111`
^	xor each bit	`ain ^ bin is 3'b011`
~^ or ^~	xnor each bit	`ain ^~ bin = 3'b100`

- Operates on each bit of the operand
- Result is the size of the largest operand
- Left-extended if sizes are different
- Bitwise operators are X-optimistic

Bitwise Operators (cont'd)

Table 5-5: Bitwise AND truth table

&	**0**	**1**	**x**	**z**
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 5-6: Bitwise OR truth table

|	**0**	**1**	**x**	**z**
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Table 5-7: Bitwise XOR truth table

^	**0**	**1**	**x**	**z**
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Table 5-8: Bitwise exclusive NOR truth table

^~ ~^	**0**	**1**	**x**	**z**
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Sutherland: Ch 5

Reduction operators

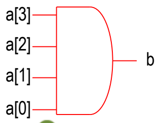
Table 5-9: Reduction operators for RTL modeling

Operator	Example Usage	Description
&	& m	AND all bits of m
~&	~& m	NAND all bits of m
|	| m	OR all bits of m
~|	~| m	NOR all bits of m
^	^ m	Exclusive-OR all bits of m
~^ **^~**	~^ m	Exclusive-NOR all bits of m

- Models gates yielding a single output bit
- Reduction operators are X-Optimistic

```
logic[3:0] a;
Logic b;
assign b = &a;
```

⇔

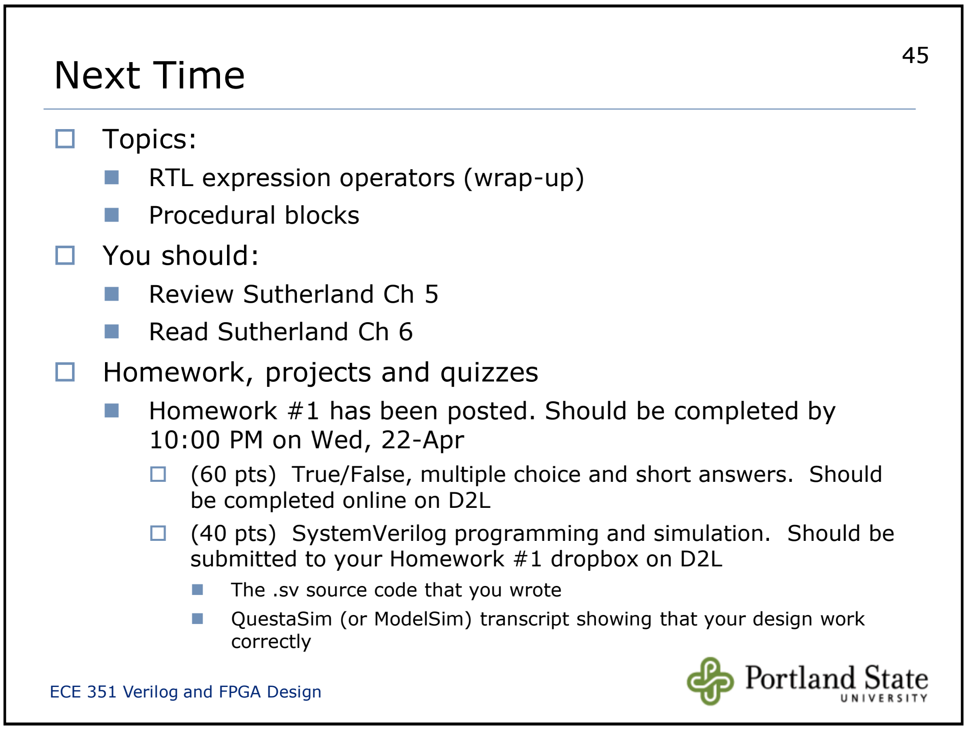


Sutherland: Ch 5

Reduction operators (cont'd)

```
// User-defined type definitions
package definitions_pkg;
  typedef struct {
    logic [3:0] data;
    logic       parity_bit;
  } data_t;
endpackage: definitions_pkg

// Parity checker using even parity (the combined data value
// plus parity bit should have an even number of bits set to 1
module parity_checker
 import definitions_pkg::*;
(input  data_t data_in,  // 5-bit structure input
 input  clk,             // clock input
 input  rstN,            // active-low asynchronous reset
 output logic  error     // set if parity error detected
);

  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) error <= 0;                 // active-low reset
    else       error <= ^{data_in.parity_bit, data_in.data};
      // reduction-XOR returns 1 if an odd number of bits are
      // set in the combined data and parity_bit
endmodule: parity_checker
```

Sutherland: Example 5.6



Next Time

- Topics:
  - RTL expression operators (wrap-up)
  - Procedural blocks
- You should:
  - Review Sutherland Ch 5
  - Read Sutherland Ch 6
- Homework, projects and quizzes
  - Homework #1 has been posted. Should be completed by 10:00 PM on Wed, 22-Apr
    - (60 pts) True/False, multiple choice and short answers. Should be completed online on D2L
    - (40 pts) SystemVerilog programming and simulation. Should be submitted to your Homework #1 dropbox on D2L
      - The .sv source code that you wrote
      - QuestaSim (or ModelSim) transcript showing that your design work correctly