שפת תכנון חומרה
ורילוג - Verilog

Verilog – Dataflow Modeling

Dr. Avihai Aharon

Objectives

- **Describe the continuous assignment ("assign") statement, restrictions on the assign statement, and the implicit continuous assignment statement**
- **Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statement**
- **Define expressions, operators, and operands**
- **List operator types for all possible operations – arithmetic, logical, relational, equality, bit-wise, reduction, shift, concatenation, and conditional**

Continuous Assignment

- A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net
  - ✓ A continuous assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction
  - ✓ A continuous assignment statement starts with the keyword "assign"

- The simplest syntax of an assign statement is as follows

```
<continuous_assign>
        ::= assign <drive_strength>? <delay>? <list_of_assignments>
```

- ✓ Drive strength is optional and can be specified in terms of strength levels (the default value for drive strength is strong1 and strong0)
- ✓ The delay value is also optional and can be used to specify delay on the assign statement

Continuous Assignment

- **Continuous assignments** have the following characteristics
  - The left hand side of an assignment must always be scalar or vector net or a concatenation of scalar and vector nets (cannot be a register !!!)
  - Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side (RHS) operands changes and the value is assigned to the left-hand-side (LHS) net
  - The operands on the RHS can be registers or nets or function calls. Registers or nets can be scalars or vectors
  - Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value.

```
assign out = i1 & i2;
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0]
```

Implicit Continuous Assignment

- Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared
  - ✓ Can be only one implicit declaration assignment per net because a net is declared only once

```
// Regular continuous assignment
wire out;
assign out = i1 & i2;

// Same effect is achieved by an implicit continuous assignment
wire out = i1 & i2;
```

Continuous Assignment: Delays

- **Delay values control the time between the change in a RHS operand and when the new value is assigned to the LHS**
- **There are three ways of specifying delays in continuous assignment statements**
  - ✓ **Regular assignment delay**
  - ✓ **Implicit continuous assignment delay**
  - ✓ **Net declaration delay**

Regular Assignment Delay

- The first method is to assign a delay value in a continuous assignment statement

- The delay value is specified after the keyword assign
  - Any change in values of in1 and in2 will result in a delay of 10 time units before re-computation of the expression in1 & in2, and the result will be assigned to out
  - If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered (internal delay)
  - An input pulse that is shorter than the delay of the assignment statement does not propagate to the output

```
assign #10 out = i1 & i2; // Delay in a continuous assignment
```

Implicit Continuous Assignment Delay

- **An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net**
  - ✓ **The declaration below has the same effect as defining wire out and declaring a continuous assignment on out**

```
// Implicit continuous assignment delay
wire #10 out = i1 & i2;

// same as
wire out;
assign #10 out = i1 & i2;
```

Net Declaration Delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net
  - If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly
  - Net declaration delays can also be used in gate-level modeling

```
// Net Delays
wire #10 out;
assign out = i1 & i2;

// The above statement has the same effect as the following
wire out;
assign #10 out = i1 & i2;
```

Expressions, Operators and Operands

- Dataflow modeling describes the design in terms of expressions instead of primitive gates

- The basis of dataflow modeling formed by
  - ✓ Expressions – constructs that combine operators and operands to produce a result
  - ✓ Operands – can be constants, integers, real numbers, nets, registers, times, bit-select, part-select, memories or function calls
  - ✓ Operators – acts on the operands to produce desired results. Verilog provides various types of operators.

Verilog Operator Types (1)

Operators Type	*Operator Symbol*	*Operation Performed*	*Number of Operands*
Arithmetic	*	Multiply	Two
	/	Divide	Two
	+	Add	Two
	-	Subtract	Two
	%	modulus	Two
Logical	!	Logical negation	One
	&&	Logical and	Two
	||	Logical or	Two
Relational	>	Greater than	Two
	<	Less than	Two
	>=	Greater than or equal to	Two
	<=	Less than or equal to	Two

Verilog Operator Types (2)

<table>
<tr><th>Operators Type</th><th>Operator Symbol</th><th>Operation Performed</th><th>Number of Operands</th></tr>
<tr><td>Equality</td><td>==</td><td>Equality</td><td>Two</td></tr>
<tr><td></td><td>!=</td><td>Inequality</td><td>Two</td></tr>
<tr><td></td><td>===</td><td>Case equality</td><td>Two</td></tr>
<tr><td></td><td>!==</td><td>Case inequality</td><td>Two</td></tr>
<tr><td>Bitwise</td><td>~</td><td>Bitwise negation</td><td>One</td></tr>
<tr><td></td><td>&</td><td>Bitwise and</td><td>Two</td></tr>
<tr><td></td><td>|</td><td>Bitwise or</td><td>Two</td></tr>
<tr><td></td><td>^</td><td>Bitwise xor</td><td>Two</td></tr>
<tr><td></td><td>^~ or ~^</td><td>Bitwise xnor</td><td>Two</td></tr>
<tr><td>Reduction</td><td>& / ~&</td><td>Reduction and / nand</td><td>One</td></tr>
<tr><td></td><td>| / ~|</td><td>Redaction or / nor</td><td>One</td></tr>
<tr><td></td><td>^ / ^~ or ~^</td><td>Reduction xor / xnor</td><td>One</td></tr>
</table>

Verilog Operator Types (3)

Operators Type	*Operator Symbol*	*Operation Performed*	*Number of Operands*
Shift	<<	Right shift	Two
	>>	Left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Operators in Verilog: Arithmetic Operators

- There are two types of arithmetic operators: binary and unary
- Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), and modulus (%)
  - If any operand bit has a value "x", then the result of the entire expression is "x" (this seems intuitive because if an operand value is not known precisely, the result should be an unknown
  - Modulus operators produce the reminder from the division of two numbers
- The operators "+" and "-" can also work as unary operators
  - They are used to specify the positive or negative sign of the operands and have higher precedence than the binary "+" or "-" operators
  - Negative numbers are represented as 2's complement internally in Verilog (negative numbers of the type <sss>'<base> <nnn> should be avoided!!!)
  - Use negative numbers only of the type integer or real in expressions!!!

Operators in Verilog: Logical Operators

- **Logical operators are logical -and (&&), -or (||) and -not (!)**
  - ✓ **Operators "&&" and "||" are binary operators.**
  - ✓ **Operator "!" Is a unary operator**
- **Logical operators follow these conditions**
  - ✓ **Logical operators always evaluate a 1-bit value, "0" (false), "1" (true), or "x" (ambiguous)**
  - ✓ **In an operand is not equal to zero, it is equivalent to a logical "1" (true condition). If it is equal to zero, it is equivalent to a logical "0" (false condition). If any operand bit is "x" or "z", it is equivalent to "x" (ambiguous condition) and it is normally treated by simulators as a false condition**
  - ✓ **Logical operators take variables or expressions as operands**
- **Use of parentheses to group logical operands is highly recommended to improve readability (also user does not have to remember the precedence of operators)**

Operators in Verilog: Bitwise Operators

- ***Bitwise*** **operators are** ***negation*** **(~),** ***and*** **(&),** ***or*** **(|),** ***xor*** **(^), and** ***xnor*** **(^~, ~^)**
  - ✓ ***A "z" is treated as an "x" in bitwise operation***
  - ✓ ***Logic tables for the bitwise computation***

and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

not	Result
0	1
1	0
x	x

Operators in Verilog: Reduction Operators

- Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (^~, ~^)
  - ✓ Reduction operators take only one operand
  - ✓ Reduction operators perform a bitwise operation on a single vector operand and yield 1-bit result
  - ✓ The logic tables are the same as shown for bitwise operators
  - ✓ The difference is that bitwise operation are on bits from two different operands, whereas reduction operations are on the bits of the same operand
  - ✓ Reduction operators work bit by bit from right to left
  - ✓ Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively

Operators in Verilog: Unary Operators

❖ **a = 1011** **and** **b = 0000**

Bit-Wise Negation	Logical Negation	Unary Reduction
~ a = 0100	*! a = 0*	*& a = 0 & b = 0*
~ b = 1111	*! b = 1*	*| a = 1 | b = 0*
		^ a = 1 ^ b = 0

- ✓ **The bit-wise negation operator inverts the value of each bit of the operand, to produce result with the same number of bits as the operand**
- ✓ **The logical negation operator reduces an operand to its logical inverse. If an operand contains all zeroes, it is false (logic "0"). If it is non-zero, it is true (logic "1"). If it is unknown, its logical value is ambiguous**
- ✓ **Unary reduction operators and the logical negation operators on all bits of a single operand to produce a single-bit result**

Operators in Verilog: Binary Operators

- a = 1010
  b = 0011
  c = 0000

Bit-Wise	Logical
a | b = 1011	*a || b = 1*
a & b = 0010	*a && b = 1*
a | c = 1010	*a || c = 1*
a & c = 0000	*a && c = 0*

- ✓ The **bit-wise binary** operator performs bit-wise manipulation on two operands. They compare each bit in one operand with its corresponding bit in the other operand to calculate each bit for the result
- ✓ **Logical binary** operators operate on logic values. If an operand contains all zeros, it is false (logic "0"). If it is non-zero, it is true (logic "1"). If it is unknown, its logical value is ambiguous
- ✓ Because the operands can be declared to be of **different size**, the smaller operand is zero-extended to the size of the larger operand during the operation

Operators in Verilog: Relation Operators

- **Relation operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=)**
  - **If relational operators are used in an expression, the expression returns a logical value of "1" (true condition) if the expression is true and logical value of "0" (false condition) if the expression is false**
  - **If there are any unknown "x" or "z" bits in the operands, the expression takes the value "x" (ambiguous condition)**

Operators in Verilog: Equality Operators

- **Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==)**
  - ✓ **When used in an expression, equality operators return logical value "1" if true, "0" if false**
  - ✓ **These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length**

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
a = 2'b1x;
b = 2'b1x;

if (a == b)   $display("a is equal to b");
else          $display("a is not equal to b");
```

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
a = 2'b1x;
b = 2'b1x;

if (a === b)  $display("a is identical to b");
else          $display("a is not identical to b");
```

Operators in Verilog: Shift Operators

- **Shift operators are right shift (>>) and left shift (<<)**
  - **These operators shift a vector operand to the right or the left by a specified number of bits**
  - **The operands are the vector and the number of bits to shift**
  - **When the bits are shifted, the vacant bit positions are filled with zeros**
  - **Shift operations do not wrap around**

```
// X = 4'b1100

Y = X >> 1;     // Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position
Y = X << 1;     // Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position
Y = X << 2;     // Y is 4'b0000. Shift left 2 bits
```

Operators in Verilog: Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple operands
  - ✓ The operands must be sized (unsized operands are not allowed)
  - ✓ Concatenations are expressed as operands within braces, with commas separating the operands
  - ✓ Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constant

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B, C};                    // Result Y is 4'b0010
Y = {A, B, C, D, 3'b001};      // Result Y is 11'b10010110001
Y = {A, B[0], C[1]};           // Result Y is 3'b101
```

Operators in Verilog: Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant
  - ✓ A replication constant specifies how many times to replicate the number inside the brackets ({ })

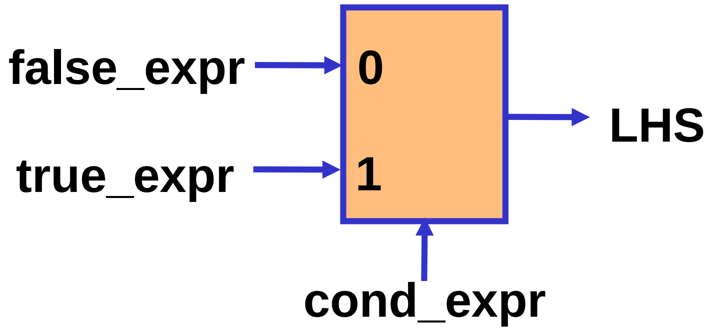
```
reg A;
reg [1:0] B, C;
reg [2:0] D;
wire Y[9:0];
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A}};                    // Result Y is 4'b1111
Y = { 4{A}, 2{B}};              // Result Y is 8'b11110000
Y = { 4{A}, 2{B}, C};           // Result Y is 10'b1111000010
```

Operators in Verilog: Conditional Operator

- The conditional operator (?:) takes three operands:

  <LHS> = <condition_expr> ? <true_expr> : <false expr>

  - ✓ The condition expression (condition_expr) is first evaluated.
  - ✓ If the result is true (logical “1”), then the true_expr is evaluated
  - ✓ If the result is false (logical “0”), then the false_expr is evaluated
  - ✓ If the result is “x” (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an “x” is the bits are different and the value of the bits if they are the same



Operators in Verilog: Conditional Operator

- ✓ **The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression**

```
// Model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 35'bz;
 // Model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

- ✓ **Conditional operators can be nested. Each true_expr or false_expr can itself be a conditional operation**

```
// Check that A==3 and control are the two select signals of
// 4-to-1 mux with n, m, x, y as the inputs and out as the
// output signal
assign out = (A == 3) ? ( control ? x : y ) : ( control ? m : n );
```

Operators in Verilog: Operator Precedence

<table>
<tr><th>Type of Operators</th><th>Examples</th><th>Degree</th></tr>
<tr><td>Concatenate & Replicate</td><td>{ } {{ }}</td><td>(highest)</td></tr>
<tr><td>Unary</td><td>! ~ & | ^</td><td></td></tr>
<tr><td>Arithmetic</td><td>* / %</td><td></td></tr>
<tr><td></td><td>+ -</td><td></td></tr>
<tr><td>Logical shift</td><td><< >></td><td>Precedence</td></tr>
<tr><td>Relational</td><td>< > >= <=</td><td></td></tr>
<tr><td>Equality</td><td>== === != !==</td><td></td></tr>
<tr><td>Binary (bit-wise)</td><td>& | ^ ~^</td><td></td></tr>
<tr><td>Binary (logical)</td><td>&& ||</td><td></td></tr>
<tr><td>Conditional</td><td>?:</td><td>(lowest)</td></tr>
</table>

Dataflow Modeling Summary

- **Continuous assignment is one of the main constructs used in dataflow modeling. A continuous assignment is always active and the assignment expression is evaluated as soon as one of the RHS variables changes. The LHS of a continuous assignment must be a net. Any logic function can be realized with continuous assignments**
- **Delay values control the time between the change in a RHS variable and when the new value is assigned to the LHS. Delays on a net can be defined in the assign statement, implicit continuous assignment, or net declaration**
- **Assignment statements contain expressions, operators, and operands**
- **The operator types are arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, replication, and conditional. Unary operators require one operand, binary operators require two operands, and ternary require three operands. The concatenation operator can take any number of operands**
- **The conditional operator behaves like multiplexer in hardware or like the if-then-else statement in programming language**