Textbook: Verilog® HDL 2nd. Edition
Samir Palnitkar
Prentice-Hall, Inc.

INSTRUCTOR: CHING-LUNG SU
E-mail: kevinsu@yuntech.edu.tw
Chapter 6
Dataflow Modeling
Outline of Chapter 6

6.1 Continuous Assignments
6.2 Delays
6.3 Expressions, Operations, and Operands
6.4 Operator Types
6.5 Examples
6.1 Continuous Assignments

6.2 Delays

6.3 Expressions, Operations, and Operands

6.4 Operator Types

6.5 Examples
Continuous Assignments

- Continuous Assignments: The most basic statement in dataflow modeling, used to drive a value onto a net.
- Continuous assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.
- Continuous assignment statement starts with the keyword assign.
- The syntax of an assign statement is as follows:

```plaintext
continuous_assign ::= assign [ drive_strength ] [ delay3 ]
list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```
**Continuous Assignments** (Cont.)

- 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 assignments have the following characteristics in P-7:
The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.

Continuous assignments are always active. The assignment expression is evaluated at the right-hand-side operands changes and the value is assigned to the left-hand-side net.

The operands on the right-hand side can be registers or nets or function calls.

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. It is very useful in modeling timing behavior in real circuits.
Examples of Continuous Assignments

// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
6.1 Continuous Assignments

- **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.

- There can be only one implicit declaration assignment per net because a net is declared only once.

- Examples:

  //Regular continuous assignment
  wire out;
  assign out = in1 & in2;

  //Same effect is achieved by an implicit continuous assignment
  wire out = in1 & in2;
6.1 Continuous Assignments

- Implicit Net Declaration

- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name.

- If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

- Example:

  ```
  // Continuous assign. out is a net.
  wire i1, i2;
  assign out = i1 & i2; // Note that out was not declared as a wire
  // but an implicit wire declaration for out
  // is done by the simulator
  ```
6.1 Continuous Assignments
6.2 Delays
6.3 Expressions, Operations, and Operands
6.4 Operator Types
6.5 Examples
6.2 Delays

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side.

- Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.
6.2 Delays

- **Regular Assignment Delay**
  - The regular assignment delay value is specified after the keyword `assign`.
  - Any change in values of `in1` or `in2` will result in a delay of 10 time units before recomputation 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.
  - This property is called inertial delay.
  - An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```verilog
code
assign #10 out = in1 & in2;
// Delay in a continuous assign
```
6.2 Delays

- **Simulation Result of Regular Assignment Delay**

- When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- When in1 goes low at 60, out changes to low at 70.
- However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
- A pulse of width less than the specified assignment delay is not propagated to the output.
Implicit Continuous Assignment Delay

//implicit continuous assignment delay
wire #10 out = in1 & in2;

//same as
wire out;
assign #10 out = in1 & in2;
6.2 Delays

- **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.
  - Examples:

    ```vhdl
    //Net Delays
    wire # 10 out;
    assign out = in1 & in2;
    
    //The above statement has the same effect as the following.
    wire out;
    assign #10 out = in1 & in2;
    ```
6.3 Expressions, Operations, and Operands

6.1 Continuous Assignments
6.2 Delays
6.3 Expressions, Operations, and Operands
6.4 Operator Types
6.5 Examples
Dataflow modeling describes the design in terms of expressions instead of primitive gates.

Expressions, operators, and operands form the basis of dataflow modeling.
Expressions

Expressions are constructs that combine operators and operands to produce a result.

Examples:

// Examples of expressions. Combines operands and operators
a ^ b
in1 | in2
6.3 Expressions, Operations, and Operands

**Operands**

- Operands can be any one of the data types.
- Operands can be **constants, integers, real numbers, nets, registers, times, bit-select, part-select** (selected bits of the vector net or register vector), and **memories or function calls**.
6.3 Expressions, Operations, and Operands

- **Operands** (Cont.)

- **Examples for Operands:**

  integer count, final_count;
  final_count = count + 1; //count is an integer operand

  real a, b, c;
  c = a - b; //a and b are real operands

  reg [15:0] reg1, reg2;
  reg [3:0] reg_out;
  reg_out = reg1[3:0] ^ reg2[3:0]; //reg1[3:0] and reg2[3:0] are part-select register operands

  reg ret_value;
  ret_value = calculate_parity(A, B); //calculate_parity is a function type operand
Operators

Operators act on the operands to produce desired results.
Verilog provides various types of operators.
Operator types are discussed in detail in Section 6.4.

Examples:

\[ d1 \&\& d2 \] // \&\& is an operator on operands d1 and d2
\[ !a[0] \] // ! is an operator on operand a[0]
\[ B >> 1 \] // >> is an operator on operands B and 1
6.1 Continuous Assignments
6.2 Delays
6.3 Expressions, Operations, and Operands
6.4 Operator Types
6.5 Examples
6.4 Operator Types

- Verilog provides many different operator types.
- Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional.
- Some of these operators are similar to the operators used in the C programming language.
- Each operator type is denoted by a symbol.
### 6.4 Operator Types

#### Operator Types and Symbols

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
<th>Number of Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>multiply</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divide</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>add</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtract</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>power (exponent)</td>
<td>two</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>logical negation</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>logical and</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>greater than</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
<td>two</td>
</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>&amp;</td>
<td>bitwise and</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bitwise or</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>
</tbody>
</table>
### Operator Types and Symbols (Cont.)

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbols</th>
<th>Description</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>&amp;</td>
<td>reduction and reduction and</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>reduction nand</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>reduction or</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>^&amp;</td>
<td>reduction nor</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>^~</td>
<td>reduction xor</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>~^</td>
<td>reduction xnor</td>
<td>one</td>
</tr>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td>Two</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>Left shift</td>
<td>Two</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;&gt;</td>
<td>Arithmetic right shift</td>
<td>Two</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;&lt;</td>
<td>Arithmetic left shift</td>
<td>Two</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{}</td>
<td>Concatenation</td>
<td>Any number</td>
</tr>
<tr>
<td>Replication</td>
<td>{ { } }</td>
<td>Replication</td>
<td>Any number</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
<td>Conditional</td>
<td>Three</td>
</tr>
</tbody>
</table>
6.4 Operator Types

- **Arithmetic Operators**

  - There are two types of arithmetic operators: **binary** and **unary**.
6.4 Operator Types

- **Binary Operators**

  - Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). 
  - Binary operators take two operands. 
  - Examples:

    ```
    A = 4'b0011; B = 4'b0100;   // A and B are register vectors
    D = 6; E = 4; F=2          // D and E are integers
    A * B // Multiply A and B. Evaluates to 4'b1100
    D / E // Divide D by E. Evaluates to 1.
           //Truncates any fractional part.
    A + B // Add A and B. Evaluates to 4'b0111
    B - A // Subtract A from B. Evaluates to 4'b0001
    F = E ** F;    //E to the power F, yields 16
    ```
6.4 Operator Types

- **Binary Operators** (Cont.)

- 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.)

- Examples:

  ```
  in1 = 4'b101x;
  in2 = 4'b1010;
  sum = in1 + in2;    // sum will be evaluated to the value 4'bx
  ```
6.4 Operator Types

- **Binary Operators** (Cont.)

- Modulus operators produce the remainder from the division of two numbers.
- They operate similarly to the modulus operator in the C programming language.
- Examples:
  
  13 \% 3  // Evaluates to 1
  16 \% 4  // Evaluates to 0
  -7 \% 2  // Evaluates to -1, takes sign of the first operand
  7 \% -2  // Evaluates to +1, takes sign of the first operand
 Unary Operators

- Operators + and - can work as unary operators.
- Operators + and - are used to specify the positive or negative sign of the operand.
- Unary + or - operators have higher precedence than the binary + or - operators.

Examples:
-4 // Negative 4
+5 // Positive 5
Unary Operators (Cont.)

- Negative numbers are represented as 2's complement internally in Verilog.
- To use negative numbers only of the type integer or real in expressions.
- Designers should avoid negative numbers of the type `<sss>` '<`<base>` `<nnn>` in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.
Examples:

// Advisable to use integer or real numbers
-10 / 5  // Evaluates to -2

//Do not use numbers of type <sss> '<base> <nnn>
-'d10 / 5  // Is equivalent (2's complement of 10)/5
// = (2^32 - 10)/5 where 32 is the default machine word width.
// This evaluates to an incorrect and unexpected result
6.4 Operator Types

Logical Operators

- Logical operators are logical-and (&&), logical-or (||), and logical-not (!).
- Operators && and || are binary operators.
- Operator ! is a unary operator.
- Logical operators follow these conditions:
  1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
  2. If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
  3. Logical operators take variables or expressions as operands.
Use of parentheses to group logical operations is highly recommended to improve readability.

Users do not have to remember the precedence of operators.
Logical Operators (Cont.)

- **Examples:**

  // Logical operations
  A = 3; B = 0;
  A && B   // Evaluates to 0. Equivalent to (logical-1 && logical-0)
  A | | B  // Evaluates to 1. Equivalent to (logical-1 || logical-0)
  !A      // Evaluates to 0. Equivalent to not(logical-1)
  !B      // Evaluates to 1. Equivalent to not(logical-0)

  // Unknowns
  A = 2'b0x; B = 2'b10;
  A && B   // Evaluates to x. Equivalent to (x && logical 1)

  // Expressions
  (a == 2) && (b == 3)  // Evaluates to 1 if both a == 2
  // and b == 3 are true. Evaluates
  // to 0 if either is false.
Relational Operators

- Relational 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 if the expression is true and 0 if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value x.
- Operators function exactly as the corresponding operators in the C programming language.
### 6.4 Operator Types

#### Relational Operators (Cont.)

**Examples:**

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B  // Evaluates to a logical 0
A > B   // Evaluates to a logical 1
Y >= X  // Evaluates to a logical 1
Y < Z   // Evaluates to an x
```
6.4 Operator Types

Equality Operators

- Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==).
- In an expression, equality operators return logical value 1 if true, 0 if false.
- Operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.
### Equality Operators (Cont.)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Possible Logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a == b</code></td>
<td><code>a</code> equal to <code>b</code>, result unknown if <code>x</code> or <code>z</code> in <code>a</code> or <code>b</code></td>
<td><code>0, 1, x</code></td>
</tr>
<tr>
<td><code>a != b</code></td>
<td><code>a</code> not equal to <code>b</code>, result unknown if <code>x</code> or <code>z</code> in <code>a</code> or <code>b</code></td>
<td><code>0, 1, x</code></td>
</tr>
<tr>
<td><code>a === b</code></td>
<td><code>a</code> equal to <code>b</code>, including <code>x</code> and <code>z</code></td>
<td><code>0, 1</code></td>
</tr>
<tr>
<td><code>a !== b</code></td>
<td><code>a</code> not equal to <code>b</code>, including <code>x</code> and <code>z</code></td>
<td><code>0, 1</code></td>
</tr>
</tbody>
</table>
The logical equality operators (==, !!=) will yield an x if either operand has x or z in its bits.

The case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z.

The result of the case equality is 1 if the operands match exactly, including x and z bits.

The result of the case equality is 0 if the operands do not match exactly.

Case equality operators never result in an x.
6.4 Operator Types

- **Equality Operators (Cont.)**

**Examples:**

// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
Bitwise Operators

- Bitwise operators are negation (~), and (&), or ( | ), xor (^), xnor ( ^~, ~^ ).
- Bitwise operators perform a bit-by-bit operation on two operands.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand.
- A z is treated as an x in a bitwise operation.
- The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.
## 6.4 Operator Types

- **Bitwise Operators** (Cont.)

- **Truth Tables for Bitwise Operators**

<table>
<thead>
<tr>
<th>bitwise and</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bitwise or</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bitwise xor</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bitwise xnor</th>
<th>0</th>
<th>1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bitwise negation</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
6.4 Operator Types

- **Bitwise Operators** (cont.)

- Example of Bitwise Operators

  // X = 4'b1010, Y = 4'b1101
  // Z = 4'b10x1

  ~X // Negation. Result is 4'b0101
  X & Y // Bitwise and. Result is 4'b1000
  X | Y // Bitwise or. Result is 4'b1111
  X ^ Y // Bitwise xor. Result is 4'b0111
  X ^~ Y // Bitwise xnor. Result is 4'b1000
  X & Z // Result is 4'b10x0
6.4 Operator Types

- **Bitwise Operators** (Cont.)

- Distinguishing bitwise operators `~`, `&`, and `|` from logical operators `!`, `&&`, `||`.
- Logical operators yield a logical value 0, 1, x.
- Bitwise operators yield a bit-by-bit value.
- Logical operators perform a logical operation, not a bit-by-bit operation.
- Examples:
  
  ```
  // X = 4'b1010, Y = 4'b0000
  X | Y // bitwise operation. Result is 4'b1010
  X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
  ```
6.4 Operator Types

- **Reduction Operators**
  - Reduction operators are and (\&), nand (\sim\&), or (\mid), nor (\sim\mid), xor (\^), and xnor (\sim\^, \^\sim).
  - Reduction operators take only one operand.
  - Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
  - The logic tables for the operators are the same as shown in P-44, Bitwise Operators.
  - 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.
6.4 Operator Types

- **Reduction Operators** (Cont.)

- **Examples of Reduction Operators:**

  // X = 4'b1010

  &X  //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
  |X  //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
  ^X  //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0

  //A reduction xor or xnor can be used for even or odd parity
  //generation of a vector.
Shift operators are right shift (\(\gg\)), left shift (\(\ll\)), arithmetic right shift (\(\ggg\)), and arithmetic left shift (\(\lll\)).

Regular shift operators shift a vector operand to the right or the left by a specified number of bits.

When the bits are shifted, the vacant bit positions are filled with zeros.

Shift operations do not wrap around.

Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.
6.4 Operator Types

- **Shift Operators** (Cont.)

  - **Examples of Shift Operators:**

    ```
    // 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.
    ```

    ```
    integer a, b, c; // Signed data types
    a = 0;
    b = -10; // 00111...10110 binary
    c = a + (b >>> 3); // Results in -2 decimal, due to arithmetic shift
    ```
6.4 Operator Types

- **Concatenation Operator**
  - The concatenation operator (\(\{, \}\)) provides a mechanism to append multiple operands.
  - The operands must be sized. (Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.)
  - 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 constants.
6.4 Operator Types

- **Concatenation Operator** (Cont.)

- **Examples of Concatenation Operators:**
  
  \[
  \text{A} = 1'b1, \text{B} = 2'b00, \text{C} = 2'b10, \text{D} = 3'b110
  \]

  \[
  \text{Y} = \{\text{B} , \text{C}\} \quad \text{\(\text{Result Y is 4'b0010}\)}
  \]

  \[
  \text{Y} = \{\text{A} , \text{B} , \text{C} , \text{D} , 3'b001\} \quad \text{\(\text{Result Y is 11'b10010110001}\)}
  \]

  \[
  \text{Y} = \{\text{A} , \text{B}[0], \text{C}[1]\} \quad \text{\(\text{Result Y is 3'b101}\)}
  \]
6.4 Operator Types

- **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 (\{\}).
  
  - **Examples:**

    ```
    reg A;
    reg [1:0] B, C;
    reg [2:0] D;
    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 8'b1111000010
    ```
6.4 Operator Types

- **Conditional Operator**

- The conditional operator (?) takes three operands. **Usage:** `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` if the bits are different and the value of the bits if they are the same.
6.4 Operator Types

- **Conditional Operator** (Cont.)

  - The action of a conditional operator is **similar to a multiplexer**.
  - Conditional operator can be compared to the if-else expression.
Conditional operators are frequently used in dataflow modeling to model conditional assignments.

The conditional expression acts as a switching control.

Examples:

```vhdl
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'b0;
```

```vhdl
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```
6.4 Operator Types

- **Conditional Operator** (Cont.)

- Conditional operations can be nested.

- Each `true_expr` or `false_expr` can itself be a conditional operation.

- In the example that follows, convince yourself that `(A==3)` and control are the two select signals of 4-to-1 multiplexerer with `n, m, y, x` as the inputs and `out` as the output signal.

- Example:

```plaintext
assign out = (A == 3) ? ( control ? x : y ) : ( control ? m : n ) ;
```
6.4 Operator Types

- **Operator Precedence**
  - If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence.
  - Operators listed in P-59 are in order from highest precedence to lowest precedence.
  - Parentheses be used to separate expressions except in case of unary operators or when there is no ambiguity.
## 6.4 Operator Types

### Operator Precedence (Cont.)

<table>
<thead>
<tr>
<th>Operators</th>
<th>Operator Symbols</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>+, -, !, ~</td>
<td>Highest precedence</td>
</tr>
<tr>
<td>Multiply, Divide, Modulus</td>
<td>*, /, %</td>
<td></td>
</tr>
<tr>
<td>Add, Subtract</td>
<td>+, -</td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;, &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td>==, !=, ===, !==</td>
<td></td>
</tr>
<tr>
<td>Reduction</td>
<td>&amp;, <del>&amp;, ^, ^</del></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>&amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
<td>Lowest precedence</td>
</tr>
</tbody>
</table>
6.1 Continuous Assignments
6.2 Delays
6.3 Expressions, Operations, and Operands
6.4 Operator Types
6.5 Examples
6.5 Examples

- **Examples of Dataflow Modeling**

- A design can be represented in terms of gates, data flow, or a behavioral description.

- In this section, we consider the 4-to-1 multiplexer and 4-bit full adder.

- Previously, these designs were directly translated from the logic diagram into a gate-level Verilog description. Here, we describe the same designs in terms of data flow.

- We also discuss two additional examples: a 4-bit full adder using carry lookahead and a 4-bit counter using negative edge-triggered D-flipflops.
6.5 Examples

- **4-to-1 Multiplexer**
  - Gate-level modeling of a 4-to-1 multiplexer is discussed in Section 5.1.4, Examples (Chapter 5, P-18).
  - The logic diagram for the multiplexer is given in (Chapter 5, P-19).
  - We describe the multiplexer, using dataflow statements.
  - We show two methods to model the multiplexer by using dataflow statements.
Method 1 of 4-to-1 Multiplexer: Logic Equation

- Notice that everything is same as the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations (I/O ports remain the same).
- The interface with the environment does not change, only the internals of the module change.
- Notice how concise the description is compared to the gate-level description.
Method 1 of 4-to-1 Multiplexer: Logic Equation

Verilog Descriptions:

// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out = (~s1 & ~s0 & i0) |
            (~s1 & s0 & i1) |
            (s1 & ~s0 & i2) |
            (s1 & s0 & i3) ;

endmodule
Method 2 of 4-to-1 Multiplexer: Conditional Operator

Verilog Descriptions:

// Module 4-to-1 multiplexer using data flow.
// Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
The 4-bit full adder in Section 5.1.4, Examples (Chapter 5, P-24), was designed by using gates; the logic diagram is shown in P-25 and P-27 (Chapter 5).

In this section, we write the dataflow description for the 4-bit adder.

We again illustrate two methods to describe a 4-bit full adder by means of dataflow statements.
Method 1 of 4-bit Full Adder: Dataflow Operators

Verilog Descriptions:

// Define a 4-bit full adder by using dataflow statements.
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;
endmodule
Method 2 of 4-bit Full Adder with Carry Lookahead

- An $n$-bit ripple carry adder will have $2n$ gate levels.
- The propagation time can be a limiting factor on the speed of the circuit.
- The propagation delay of carry lookahead is reduced to four gate levels, irrespective of the number of bits in the adder.
- The module can be substituted in place of the full adder modules described before without changing any other component of the simulation.
6.5 Examples

- **Method 2 of 4-bit Full Adder with Carry Lookahead**

- **Design Concept:**
6.5 Examples

Method 2 of 4-bit Full Adder with Carry Lookahead

Verilog Descriptions:

```verilog
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;

// Internal wires
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c4, c3, c2, c1;

// compute the p for each stage
assign p0 = a[0] ^ b[0],
p1 = a[1] ^ b[1],
p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];

// compute the g for each stage
assign g0 = a[0] & b[0],
g1 = a[1] & b[1],
g2 = a[2] & b[2],
g3 = a[3] & b[3];

// compute the carry for each stage
assign c1 = g0 | (p0 & c_in),
c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & c_in);

// Compute Sum
assign sum[0] = p0 ^ c_in,
    sum[1] = p1 ^ c1,
    sum[2] = p2 ^ c2,
    sum[3] = p3 ^ c3;

// Assign carry output
assign c_out = c4;
endmodule
```
6.5 Examples

- **Ripple Counter**
  - Designing a 4-bit ripple counter by using negative edge-triggered flipflops.
  - The example was discussed at a very abstract level with Hierarchical Modeling Concepts.
  - Example: 4-bit Ripple Carry Counter
6.5 Examples

- **Ripple Counter** (Cont.)

  - Building the T-Flipflop Design with D-Flipflop

![Diagram of T-Flipflop and D-Flipflop](image-url)
6.5 Examples

- Ripple Counter (Cont.)

- Designing D-Flipflop with Basic Logic Cells
Verilog Code for Ripple Counter

```
// Ripple counter
module counter(Q, clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```
6.5 Examples

- **Ripple Counter** (Cont.)

- Verilog Code for T-Flipflop

  // Edge-triggered T-flipflop. Toggles every clock cycle.

  module T_FF(q, clk, clear);
  // I/O ports
  output q;
  input clk, clear;

  // Instantiate the edge-triggered DFF
  // Complement of output q is fed back.
  // Notice qbar not needed. Unconnected port.
  edge_dff ff1(q, ~q, clk, clear);

  endmodule
6.5 Examples

**Ripple Counter** (Cont.)

Verilog Code for Edge-Triggered D-Flipflop

```verilog
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);

// Inputs and outputs
output q, qbar;
input d, clk, clear;

// Internal variables
wire s, sbar, r, rbar, cbar;

// dataflow statements
// Create a complement of signal clear
assign cbar = ~clear;

// Input latches; A latch is level sensitive. An edge-sensitive flip-flop is implemented by using 3 SR latches.
assign sbar = ~(rbar & s),
    s = ~(sbar & cbar & ~clk),
    r = ~(rbar & ~clk & s),
    rbar = ~(r & cbar & d);

// Output latch
assign q = ~(s & qbar),
    qbar = ~(q & r & cbar);

endmodule
```
Ripple Counter

Verilog Code for Stimulus Module

```
// Top level stimulus module
module stimulus;

// Declare variables for stimulating input
reg CLOCK, CLEAR;
wire [3:0] Q;

initial
$monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);

// Instantiate the design block counter
counter c1(Q, CLOCK, CLEAR);

// Stimulate the Clear Signal
initial begin
  CLEAR = 1'b1;
  #34 CLEAR = 1'b0;
  #200 CLEAR = 1'b1;
  #50 CLEAR = 1'b0;
end

// Set up the clock to toggle every 10 time units
initial begin
  CLOCK = 1'b0;
  forever #10 CLOCK = ~CLOCK;
end

// Finish the simulation at time 400
initial begin
  #400 $finish;
end
endmodule
```
6.5 Examples

■ Ripple Counter (Cont.)

Simulation Results

0 Count Q = 0000 Clear= 1
34 Count Q = 0000 Clear= 0
40 Count Q = 0001 Clear= 0
60 Count Q = 0010 Clear= 0
80 Count Q = 0011 Clear= 0
100 Count Q = 0100 Clear= 0
120 Count Q = 0101 Clear= 0
140 Count Q = 0110 Clear= 0
160 Count Q = 0111 Clear= 0
180 Count Q = 1000 Clear= 0
200 Count Q = 1001 Clear= 0
220 Count Q = 1010 Clear= 0
234 Count Q = 0000 Clear= 1
284 Count Q = 0000 Clear= 0
300 Count Q = 0001 Clear= 0
320 Count Q = 0010 Clear= 0
340 Count Q = 0011 Clear= 0
360 Count Q = 0100 Clear= 0
380 Count Q = 0101 Clear= 0