Digital Design with Verilog

Verilog

Lecture 8: Dataflow Modeling

Introduction

- Gate-level modelling
  - Suits when smaller number of gates
  - Each gate is instantiated, and connection are made
  - More intuitive: one-to-one mapping of gates
- Gate densities on chip increasing rapidly, dataflow modeling has become very important.

Introduction

- Verilog allows a circuit to be designed

in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

Learning Objectives

- Continuous assignment (**assign**) statement
  - Restrictions on the **assign** statement, and
  - Implicit continuous assignment statement.
- Define expressions, operators, and operands.

Learning Objectives

- List operator types for all possible operations:
  - arithmetic,
  - logical,
  - relational,
  - equality,
  - bitwise,
  - reduction,
  - shift,
  - concatenation, and
  - conditional.

- Use dataflow constructs to model practical digital circuits in Verilog.

Continuous Assignment

- Most basic statement in the dataflow modeling which drives a value onto a net

- Replaces gates in the description of the circuit and describes the circuit at higher level of abstraction.

and (z, a, b)

Z = a & b

Continuous Assignment

- 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 operands on the right-hand side can be registers or nets or function calls.
- Delay is used to decide when to assign the evaluated value to LHS.

Example: 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;
```

Implicit Continuous Assignment

```
//Regular continuous assignment
wire out;
assign out = in1 & in2;
//Same effect is achieved by an
//implicit continuous assignment
wire out = in1 & in2;
```

Implicit Net Declaration

```
// 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
```

Expressions, Operators, and Operands

Expressions

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

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

```
// Examples of expressions.
//Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

Operands

- Operands can be
  - constants, integers, real numbers,
  - nets, registers, times,
  - bit-select
    - one bit of vector net or a vector register,
  - part-select
    - selected bits of the vector net or register vector
  - memories or
  - function calls

Example: 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.

- Operator Types
  - Arithmetic, Logical, Relational, Equality
  - Bitwise,
  - Reduction // Not available in software languages
  - Shift, Concatenation, Replication
  - Conditional

- Syntax is very similar to C

Arithmetic Operators

- Binary Operators

```
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
```

- 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.

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

Arithmetic Operators (Contd.)

- Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

```
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: +, -**
  - Ex: +5 and –4
  - the negatives are stored as two's complement –default 32-bit, else the specified number of bits.
  - For eg. a = -4'b0011 is stored as 1101

Logical Operators

- Logical operators are
  - logical-and (&&)
  - logical-or (||)

  Binary operators

  - logical-not (!) → Unary Operator

Logical Operators

- Logical operators follow these conditions:
  - Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x(ambiguous).
  - If an operand is not equal to zero, it is a logical 1 and if it is equal to zero, it is a logical 0.
  - Logical operators take variables or expressions as operands.

Example: Logical Operators

```
// 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

- Greater-than (>)
- Less-than (<)
- Greater-than-or-equal-to (>=)
- Less-than-or-equal-to (<=)
- Evaluates to 1 or 0, depending on the values of the operands
  - If one of the bits is an ‘x’ or ‘z’, it evaluates to ‘x’

Example: Relational Operators

```
// 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
```

Equality Operators

- Equality operators are
  - logical equality (==)
  - logical inequality (!=)
  - case equality (===)
  - case inequality (!==)

Equality Operators

- Logical equality (==), logical inequality (!=)
  - if one of the bits is ‘x’ or ‘z’, they output ‘x’ else returns ‘0’ or ‘1’
- Case equality (===), case inequality (!==)
  - compares both operands bit by bit and compare all bits including ‘x’ and ‘z’.
  - Returns only ‘0’ or ‘1’

Example: Equality Operators

```
// 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

- Bitwise operators perform a bit-by-bit operation on two operands.
- They take each bit in one operand and perform the operation with the corresponding bit in the other operand.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand.

Truth Tables: Bitwise Operators

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

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

buf	in	out	not	in	out
	0	0		0	1
	1	1		1	0
	x	x		x	x
	z	x		z	x

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

Example: 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
```

Bitwise Operators (Contd.)

- It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||.

- Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value.

- Logical operators perform a logical operation, not a bit-by-bit operation.

```
// 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.
```

Reduction Operators

- Reduction operators are:

  and (&)

  nand (~&)

  or (|)

  nor (~|)

  xor (^)

  xnor (~^, ^~)

Reduction Operators

- Reduction operators take only one operand.
- Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
- Reduction operators work bit by bit from right to left.

Example: 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

- Shift operators are:

  right shift (>>)

  left shift (<<)

  arithmetic right shift (>>>)

  arithmetic left shift (<<<).

Shift Operators

- Regular shift 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.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

Example: 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.
```

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.

```
// 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
```

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;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} } // Result Y is 4'bllll
Y = { 4{A} , 2{B} } // Result Y is 8'bllll0000
Y = { 4{A} , 2{B} , C } // Result Y is 8'bllll000010
```

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.
  - false (logical 0), then the *false_expr* is evaluated.

Conditional Operator (Contd.)

- 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.

- The action of a conditional operator is similar to a multiplexer.

- It can be also compared to the if-else expression.

- Conditional operators are frequently used in dataflow modeling to model conditional assignments.

Conditional Operator (Contd.)

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```

Operator Precedence

Operators	Operator Symbols	Precedence
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

Example 1: 4x1 Mux using logic equations

```
// 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
```

Example 2:4x1 Mux using conditional Operator

```
// 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
```

Example 3: 4-bit Full Adder

```
// 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
```

Summary

- Continuous assignment is one of the main constructs used in dataflow modeling.
- Delay values control the time between the change in a right-hand-side variable and when the new value is assigned to the left-hand side.
- Assignment statements contain expressions, operators, and operands.
- Dataflow description of a circuit is more concise than a gate-level description.

References

- Chapter 6, Verilog HDL by Samir Palnitkar, Second Edition.

Thank you