Data flow level modelling

Data flow modelling is a higher level of abstraction compared to gate level modelling.

To design a circuit in a data flow level, the designer should be aware of data flow of the design .

The gate level design description makes use of the gate primitives available in Verilog. It becomes very complex for a VLSI circuit because number of gate is very large, hence data flow modelling become a very important way of implementing the design.

Data flow modelling makes the circuit description more compact as compared to the design through gate primitives.

Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than the instantiation of individual gates.

In data flow modelling most of the design is implemented using continuous assignment, which are used to drive a value onto a net.

Continuous assignment structure

A continuous assignment is the most basic statement in dataflow modelling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.
The assignment statement starts with the **keyword 'assign".**

Continuous assignment can be used in two ways , as the **net declaration statement** and as the **continuous assignment statement**.

The **syntax** of an assign statement is as follows.

```
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 have the following characteristics:

- 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 as soon as one of 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.
  Registers or nets can be scalars or vectors.
- Continuous assignment cannot be used within initial or always blocks.
- 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.

Examples of Continuous Assignment

```
// 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];

// continusous assignment with 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;

// delay specification in continuous assignment statement
assign #25 out = i1 & i2;
```

Implicit continuous assignment

Regular **continuous assignment** means, the declaration of a net and its continuous assignment are done in two different statements
but in **implicit continuous assignment**, assignment can be done on a net when it is declared itself.

Example

```
//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

In Verilog during implicit assignment , if L.H.S. is declared then it will assign the R.H.S. to the declared net.
But if the L.H.S. is not defined it will automatically create a net for the signal name.

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.

Combining assignment and net declaration

The assignment statement can be combined with the net declaration itself making the assignment implicit in the net declaration itself.
Thus the two statements
Wire c;
Assign c = a&b;
Can be combined as
Wire c = a&b;

Continuous assignment and strength

A net to which the continuous assignment is being made can be assigned strength for its logic levels.

```
module aoi4 (g, a, b, c, d);
output g;
input a, b, c, d;
wire g;
wire e  = a &&b;
wire f  = c & &d;
wire g1 = e || f;
assign (pull1, strong0) g = ~g1;
endmodule
```

Delay and continuous assignment

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**
- **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.

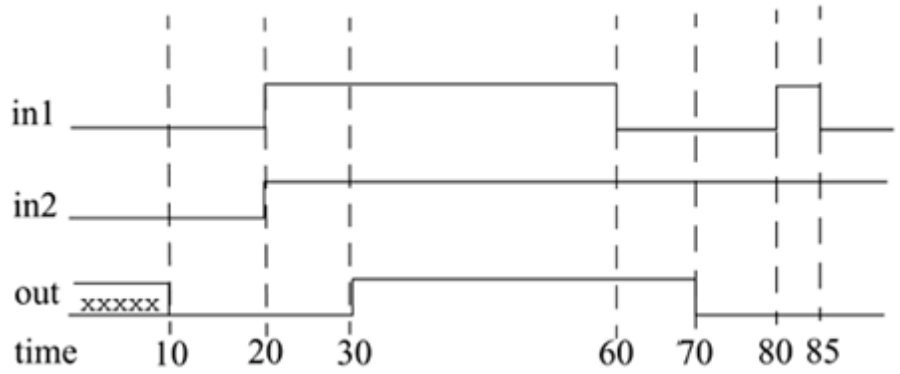
assign #10 out = in1 & in2; // Delay in a continuous assign

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.

Figure:



1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later

(time = 30).

2. When in1 goes low at 60, out changes to low at 70.

3. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.

Hence, at the time of recomputation, 10 units after time 80, in1 is 0.

Thus, out
gets the value 0. A pulse of width less than the specified assignment delay is not
propagated to the output.

Implicit Continuous Assignment Delay

An implicit continuous assignment is used to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay
wire #10 out = in1 & in2;
//same as
wire out;
assign #10 out = in1 & in2;
```

The declaration above has the same effect as defining a wire out and declaring a
continuous assignment on out.

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 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;
```

Assignment to vectors

The continuous assignment are equally applicable to vectors
Example:- **8 bit adder**

```
module add_8(a,b,c);
Input[7:0] a,b;
Output [7:0] c;
assign c=a+b;
endmodule
```

concatenation of vectors

one can concatenate vectors , scalars, and part vectors to form other vectors. The concatenated vector is enclosed with in braces. Commas separate the components - vectors , scalars, and part vectors.

Example

{a,b,c}

concatenated vector of 13 bit width if a is 8 bit vector, b is 4 bit vector and c is scalar.

The vector component formed in order , first is most significant bit i.e. a[7], last is least significant bit i.e. c and other bit are in between.

The concatenation can also be with selected segment of vectors

Example

{a(7:4), b(2:0)}

7 bit vector formed by combining the 4 most significant bit of vector a with 3 least significant bit of vector b.

The size of each operand with in the braces has to be specified fully to form the concatenated vector

A complete 8 bit adder module at data flow level

```
module add__8 bit (c,co,a,b,ci);
input[7:0] a,b;
input ci;
output[7:0] c;
output co;
assign {co,c} = (a+b+ci);
endmodule
```

when it is necessary to replicate vector, scalars etc. to form other vectors, then use repetition multiplier through concatenation.

For example

{2{p}}

It represents the concatenated vector

{p,p}

Concatenation operation can be nested to form a bigger vector when components combination are repeated.

For example

{a,3{2{b,c},d}}

It is equivalent to

{a,b,c,b,c,d,b,c,b,c,d,b,c,b,c,d}

The two statement
assign GND = supply0;
p = {8{GND}};
it ground the 8 bit of vector p.

operators

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Arithmetic Operators

There are two types of arithmetic operators: binary and unary.

Binary operators

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

example

```
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.

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

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand.

```
-4 // Negative 4
+5 // Positive 5
```

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 isfalse.
If there are any unknown or z bits in the operands, the expression takes a value x.

```
// 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
```

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 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 is normally treated by simulators as a false condition.
3. Logical operators take variables or expressions as operands.

```
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)
```

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

```
// 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.
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 for Bitwise Operators

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

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

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

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

bitwise negation	result
0	1
1	0
x	x

Examples of bitwise operators are shown below.
// 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

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 a 1-bit result .
Bitwise operations are on bits from two different operands, whereas reduction
operations are on the bits of the same operand. Reduction operators work bit by bit fromright to left.
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 (>>>), and arithmetic left shift (<<<).

```
X = 4'b1100
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB
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 ({ }).

example

```
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
```

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.

Operator precedence

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

EXAMPLES

4-to-1 Multiplexer

```
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
```

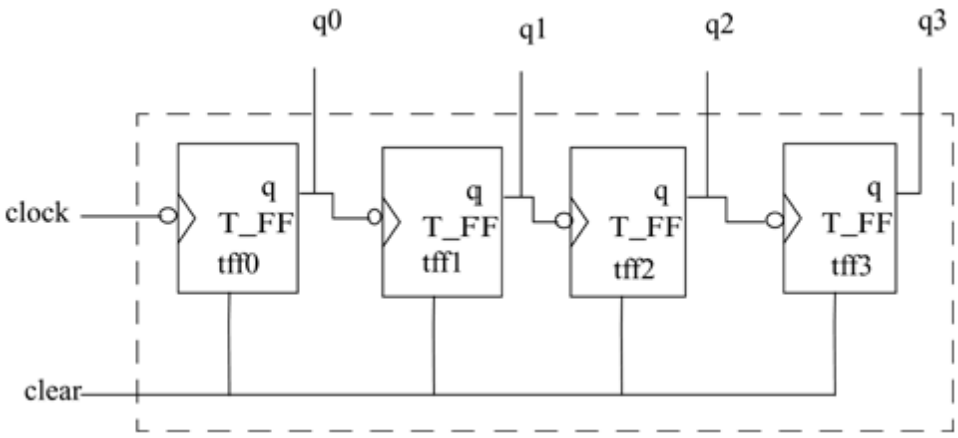
4-bit Full Adder

```
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
```

4-bit Full Adder with Carry Lookahead

```
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
// Note that c_in is equivalent c0 in the arithmetic equation for
// carry lookahead computation
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
```

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
```

Ring counter

```
module rng_ctr(cen,clk,sd,rd,q,qb);
input clk,cen;
input[3:0]sd,rd;
output [3:0]q,qb;
wire [3:0]d;
unishrg uu(clk,d,sd,rd,q,qb);
assign d[1]=(cen)? q[0]:1'b0;
assign d[2]=(cen)? q[1]:1'b0;
assign d[3]=(cen)? q[2]:1'b0;
assign d[0]=(cen)? q[3]:1'b0;
endmodule

module tst_rng_ctr;//test-bench
reg clk,cen;
reg[3:0]sd,rd;
wire [3:0]q,qb;
rng_ctr rsh(cen,clk,sd,rd,q,qb);
initial
begin
       clk=0;sd=4'b1000;rd=4'b0111;
     #3sd=4'b0000;rd=4'b0000;
     #2cen=1'b1;
end
always
begin
#2clk =~clk;
end
initial #50 $stop;
endmodule
```

BCD ADDER

```
module bcd(co,sumd,a,b);
input [3:0]a,b;
output [3:0]sumd;
output co;
wire [3:0]sumb;
assign sumb = a + b;
assign{co,sumd}=(sumb<=4'b1001)?{1'b0,sumb}:(sumb+4'b01
10);
endmodule

module tst_bcd;//Test bench
reg [3:0]a,b;
wire co;
wire [3:0]sumd;
bcd bcc(co,sumd,a,b);
initial
     begin
          a = 4'h0 ; b = 4'h0;
     #2   a = 4'h1 ; b = 4'h0;
     #2   a = 4'h2 ; b = 4'h1;
     #2   a = 4'h4 ; b = 4'h5;
     #2   a = 4'h6 ; b = 4'h6;
     #2   a = 4'hd ; b = 4'h1;
     #2   a = 4'hf ; b = 4'h0;
     end
initial $monitor($time,"a = %b, b = %b, co = %b, sumd =
%b",a,b,co,sumd);
initial #16 $stop;
endmodule
```