Section 6

Dataflow Modeling

Continuous Assignments

- The continuous assignment statement is the most basic statement in dataflow modeling.
- A continuous assignment is declared with the ***assign*** statement.
- The ***assign*** statement is used to drive a value on to a net.
- Continuous assignments have the following rules:
  - The LHS has to be either a scalar or vector net (wire).
  - Continuous assignments are always active.
  - The RHS can be a register, net, function call, or expression.
  - Delay values can be specified.
- Syntax:
  - ***assign*** [drive strength] [delay] <net assignment> = <net, register, function, expression>;

Continuous Assignments

- Examples:

```
module assignexamp (OUT1, OUT2, OUT3,
                    A, B, C, D);

input A, B, C, D;
output OUT1, OUT2, OUT3;

assign OUT1 = A;
assign OUT2 = B & C;
assign OUT3 = ((B | C) & (A ^ D));

endmodule
```

```
module assignexamp2 (OUT, A, B, C, D);

input A, B, C, D;
output [3:0] OUT;

wire [3:0] OUT;

assign OUT = {A,B,C,D};

endmodule
```

```
module addbit (CO,SUM,A,B,CI);

  input A, B, CI;
  output CO, SUM;

  assign {CO,SUM} = A + B + CI;

endmodule
```

Implicit Continuous Assignment

- You can combine the net declaration with the ***assign*** statement in Verilog.
- The implicit continuous assignment follows the same rules as the continuous assignment.

```
module assignexamp2 (OUT, A,
                     B, C, D);

input A, B, C, D;
output [3:0] OUT;

wire [3:0] OUT = {A,B,C,D};

endmodule
```

Expressions, Operators, and Operands

- Expressions are constructs that combine operators and operands to produce a result.
- Operators act on the operands to produce a desired result.
- Operands can be any legal data type.
  - Operands can be constants, integers, real numbers, registers, nets, etc.
  - Some constructs will only take certain operands.

Operators

- Verilog has several operators that perform various functions.

<table>
<tr><th>Operator Type</th><th>Symbol</th></tr>
<tr><td>Arithmetic</td><td>+, -, *, /, %</td></tr>
<tr><td>bit-wise</td><td>~, &, |, ^, ~^</td></tr>
<tr><td>logical</td><td>!, &&, ||</td></tr>
<tr><td>reduction</td><td>&, |, ^, ~&, ~|, ~^</td></tr>
<tr><td>shift</td><td><<, >></td></tr>
<tr><td>relational</td><td><, >, <=, >=</td></tr>
<tr><td>equality</td><td>= =, !=, = = =, != =</td></tr>
<tr><td>conditional</td><td>?:</td></tr>
<tr><td>concatenation</td><td>{}</td></tr>
<tr><td>replication</td><td>{{}}</td></tr>
</table>

Arithmetic Operators

- Arithmetic on unsigned data types such as reg gives an unsigned result.
- Negative results for unsigned data types are converted to two's complement.
- Integer arithmetic is signed

Arithmetic	
+	add
-	subtract
*	multiply
/	divide
%	modulus

Bitwise Operators

- Bitwise operators perform a bit-by-bit operation on the two operands.

Bitwise	
~	not
&	and
|	or
^	xor
~^	xnor
^~	xnor

```
b =  4'b1010;

a =   4'b0101;

c = a & b;   // c = 0000;
c = a | b;   // c = 1111;
```

Logical Operators

- Logical operators evaluate to 0 (false), 1 (true), or x (ambiguous).
  - If an operand has all zeros, it is false (logic 0);
  - If an operand contains any ones, it is true (logic 1);
  - If an operand contains only zeros and/or unknown bits it is ambiguous (logic x)

Logical

!	not
&&	and
||	or

```
a = 4'b0011;   evaluates to: 1
b = 4'b10xz;   evaluates to: 1
c = 4'b0000;   evaluates to: 0
d = 4'b0z0x;   evaluates to: x

r = a && b;       // r = 1
r = c || d;       // r = x
r = !(a && b);    // r = 0
```

Reduction Operators

- Reduction operators perform a bitwise operation on all the bits of the operand.
- Result is always a single bit: 0 (false), 1 (true), or x (ambiguous).

Reduction

&	and
		or
^	xor
~^	xnor
^~	xnor

```
a = 4'b0011;
b = 4'b11x1;

r = &a;  // r = (0 & 0 & 1 & 1) = 0
r = |a;  // r = (0 | 0 | 1 | 1) = 1
r = ^a;  // r = (0 ^ 0 ^ 1 ^ 1) = 0
r = &b;  // r = (1 & 1 & x & 1) = x
r = |b;  // r = (1 | 1 | x | 1) = 1
```

Shift Operators

- Shift operators perform left or right bit shifts to the operand.
- When the bits are shifted, the vacant bit positions are filled with zeros.
- Shift operations do not wrap around.

Shift	
<<	shift left
>>	shift right

```
a = 8'b0000_1100;

num = a << 4;  // num = 1100_0000
num = a >> 3;  // num = 0000_0001
```

Relational Operators

- **Relational** always give a single bit result: 0 (false), 1 (true), or x (ambiguous).

Relational

>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

```
a = 4’b0011;
b = 4’b1010;
c = 4’b0x10;

val = c > a;  // val = x
val = b < a;  // val = 0
val = a < b;  // val = 1
val = b > c;  // val = 1
```


The “less than or equal” (<=) operator looks the same as the “non-blocking” assignment. Verilog knows the difference by the context in which they’re used.

Equality Operators

- The equality operator "= =" always gives a single bit result: 0 (false), 1 (true), or x (ambiguous).
- The equality operator always evaluates the ambiguous case as 'x'.
- The case equality operator "= = =" always gives a single bit result: 0 (false), or 1 (true).
- The case equality operator always evaluates the ambiguous case as a valid value.

Equality

= =	equality
! =	inequality

Case Equality

= = =	case equality
! = =	case inequality

Equality Operators (cont.)

Equality

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

Case Equality

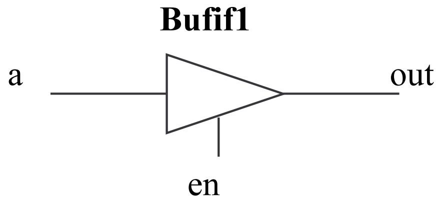
===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
a = 4'b0011;
b = 4'b1010;
c = 4'bxx10;
d = 4'bxx10;

val = (a = = b);        // val = 0
val = (a != b);         // val = 1
val = (c = = = d);      // val = 1
val = (c = = d);        // val = x
```

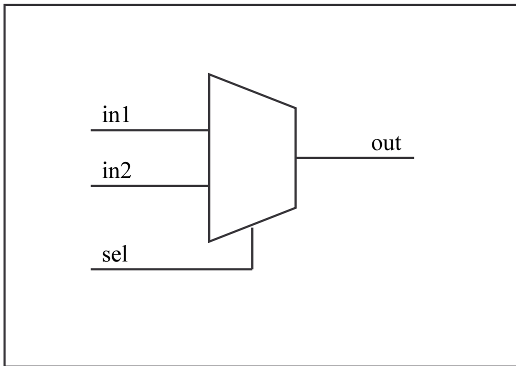
Note: Case Equality is not a synthesizable construct

Conditional Operators

- Conditional operator: ***?:***
- Syntax: ***assign*** <net> = <condition> ***?*** <true value> ***:*** <false value>



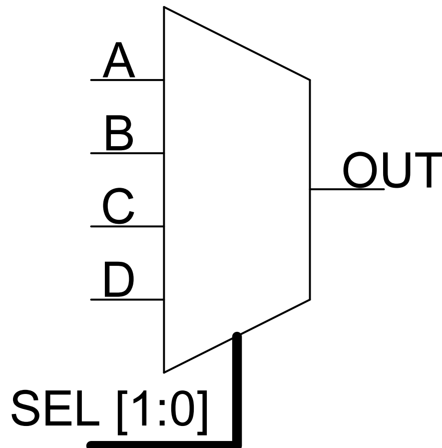

```
assign out = en ? a : 1'bz;
```




```
assign out = sel ? in2 : in1;
```

Conditional Operators

- Conditional Operators can be nested.




```
module four_one_mux (OUT, A, B, C, D,
                                        SEL);

input A, B, C, D;
input [1:0] SEL;
output OUT;

wire [1:0] SEL;

assign OUT = (SEL = = 2'b00) ? A :
                         (SEL = = 2'b01) ? B :
                         (SEL = = 2'b10) ? C : D;

endmodule
```

Concatenation

- Concatenation is used to combine signals.
- Concatenation is declared by the '{ }' symbols.
  - Your variable must match the size of the concatenation, otherwise, the MSBs will be truncated.
  - If concatenation is smaller than the declared variable, the MSBs are filled with zeros.

```
reg [3:0] a, b, c, d;
reg [7:0] val;

a = 4'b0011;
b = 4'b1010;
c = 4'b1x10;
d = 4'b0110;

val = {a, b}                              // val = 00111010
val = {a[2], b[3:1], c[0], d[2:0]} ;      // val = 01010110
val = {a, b, c};                          // val = 10101x10
val = {c[2], d[0]};                       // val = 000000x0
```

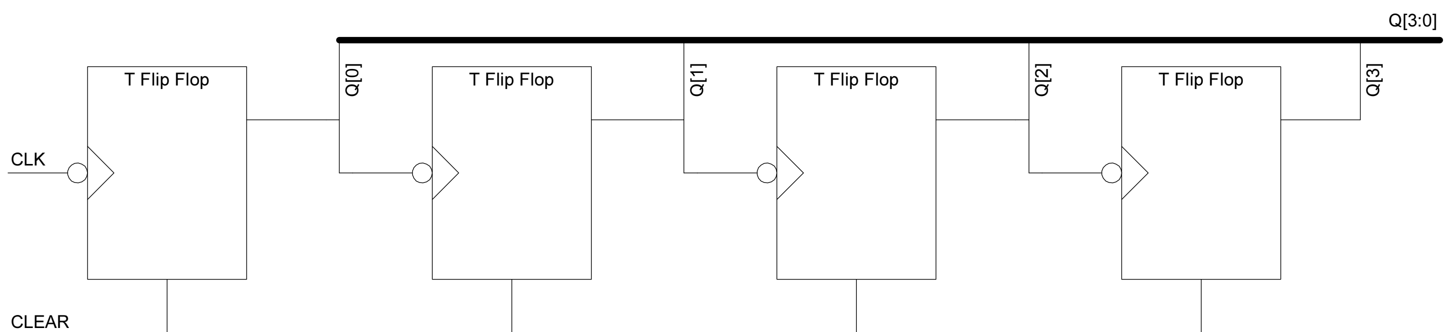
Replication

- You can replicate signals by using the replication operator: **{{ }}**
  - Replication allows you to reproduce the variable inside the { }.
  - The number of replications is always an integer, and is specified between the two leading {.
  - Concatenation rules apply for truncation/zero fill when the variable size doesn't match the replication size.

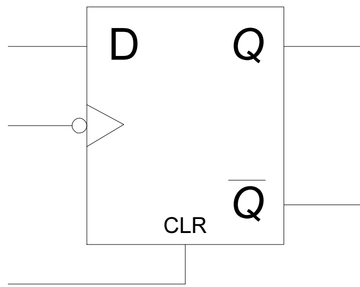
```
reg a;
reg [1:0] b;
reg [7:0] val;

a = 1'b1;
b = 2'b10;

val = {8{a}};                  // val = 11111111
val = {4{b}} ;                 // val = 10101010
val = { {4{a}},  {2{b}} };  // val = 11111010
```

Dataflow Example: 4-bit Ripple Counter

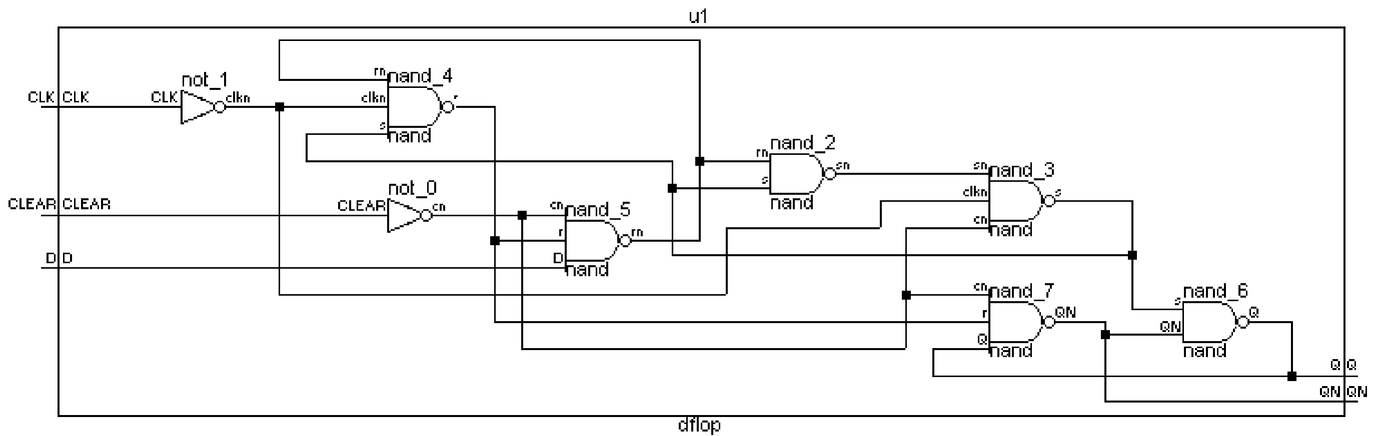
- 4-bit ripple counter consists of:
  - negative-edge triggered D Flip-Flop.
  - T Flip-Flop (based on instantiations of D Flip-Flop).
  - Counter (based on instantiations of T Flip-Flops).



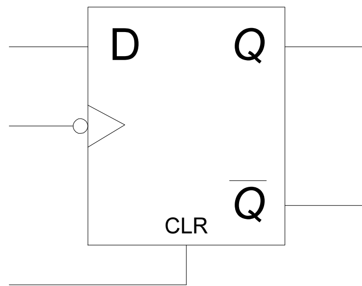
Negative Edge Triggered D Flip-Flop

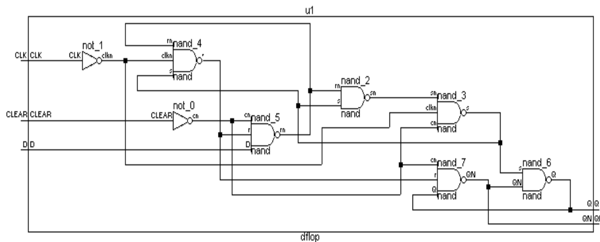


- A negative edge triggered D Flip Flop can be created with inverters and NAND gates.
  - The symbol shown at the left represents the gate-level implementation shown below.



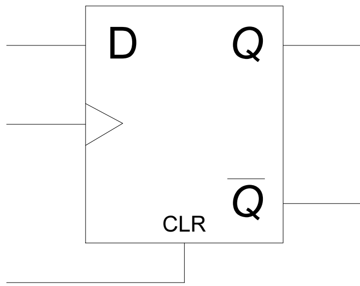
Negative Edge Trigger D Flip-Flop

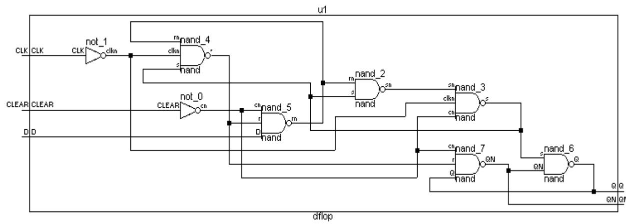




```
module dflop (Q, QN, D, CLK, CLEAR);

   input D, CLK, CLEAR;
   output Q, QN;

   wire   s, sn, r, rn, cn;

   assign cn = ~CLEAR;
   assign sn = ~(rn & s);
   assign s = ~(sn & cn & ~CLK);
   assign r = ~(rn & ~CLK & s);
   assign rn = ~(r & cn & D);
   assign Q = ~(s & QN);
   assign QN = ~(Q & r & cn);

endmodule // dflop
```

D Flip-Flop Test

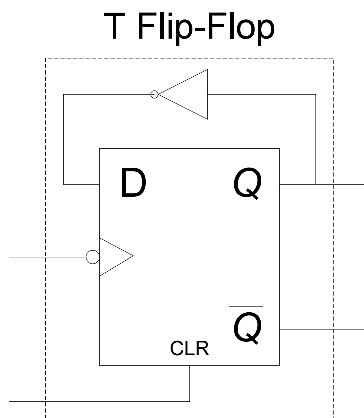





```
module tb;

reg D, CLK, CLEAR;

dflop u1 (Q, QN, D, CLK, CLEAR);

  always #10 CLK = ~CLK;

  initial begin
    #5 {D,CLK,CLEAR} = 3'b001;
    #10 CLEAR = 0;
    #20 D = 1;
    #40 D = 0;
    #40 CLEAR = 1;
    #10 CLEAR = 0;
    #20 D = 1;
    #40 D = 0;
    #40 $finish;
  end // initial begin

endmodule // tb
```

T Flip-Flop

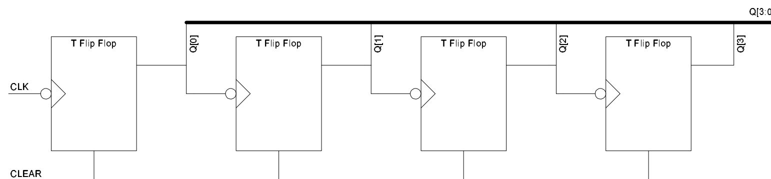
- A 'T Flip-Flop' is created by feeding the inversion of output Q to the data input of a D Flip-Flop.




```
module tflop (Q, CLK, CLEAR);

  input CLK, CLEAR;
  output Q;

  dflop u1 (.Q(Q), .QN(), .D(~Q), .CLK(CLK),
            .CLEAR(CLEAR));

endmodule // tflop
```

Ripple Counter

- The Ripple Counter is created by instantiating 4 - ‘T Flip-Flops’.



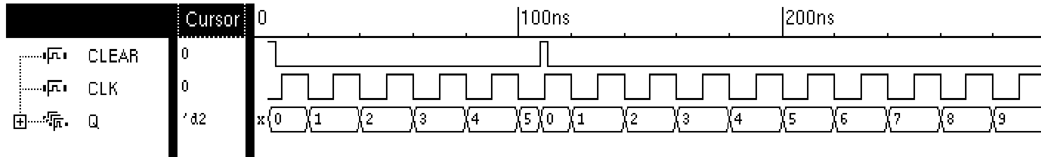

```
module counter (Q, CLOCK, CLEAR);

  input CLOCK, CLEAR;
  output [3:0] Q;

  wire [3:0] Q;

  tflop u0 (Q[0],CLOCK,CLEAR);
  tflop u1 (Q[1],Q[0],CLEAR);
  tflop u2 (Q[2],Q[1],CLEAR);
  tflop u3 (Q[3],Q[2],CLEAR);

endmodule // counter
```

Ripple Counter Test

- The counter is tested by initially clearing the 4 -‘T Flip-Flops’, and then by allowing the clock to run for a set amount of time.

```
module counter_tb;

  reg CLK, CLEAR;
  wire [3:0] Q;

  counter u1 (.Q(Q), .CLOCK(CLK), .CLEAR(CLEAR));

  always #10 CLK = ~CLK;

  initial begin
    $monitor("CLEAR = %b  Q = %d",CLEAR,Q);
    #5 {CLK,CLEAR} = 2'b01;
    #3 CLEAR = 0;
    #100 CLEAR = 1;
    #3 CLEAR = 0;
    #350 $stop;

  end // initial begin

endmodule // tb
```

Ripple Counter Test Results




```
# CLEAR = x  Q =  x
# CLEAR = 1  Q =  0
# CLEAR = 0  Q =  0
# CLEAR = 0  Q =  1
# CLEAR = 0  Q =  2
# CLEAR = 0  Q =  3
# CLEAR = 0  Q =  4
# CLEAR = 0  Q =  5
# CLEAR = 1  Q =  0
# CLEAR = 0  Q =  0
# CLEAR = 0  Q =  1
# CLEAR = 0  Q =  2
# CLEAR = 0  Q =  3
# CLEAR = 0  Q =  4
# CLEAR = 0  Q =  5
# CLEAR = 0  Q =  6
# CLEAR = 0  Q =  7
# CLEAR = 0  Q =  8
# CLEAR = 0  Q =  9
# CLEAR = 0  Q = 10
# CLEAR = 0  Q = 11
# CLEAR = 0  Q = 12
# CLEAR = 0  Q = 13
# CLEAR = 0  Q = 14
# CLEAR = 0  Q = 15
# CLEAR = 0  Q =  0
# CLEAR = 0  Q =  1
# CLEAR = 0  Q =  2
```

Review

- What is the difference between '= =' and '= = ='? What is the name of each operator?
- What happens if the concatenation size exceeds the declared register size?
- What is the difference between a bitwise operator and a logical operator?
- What Verilog keyword is fundamental to dataflow design?