Dataflow Modeling

- Dataflow modeling provides means of describing a circuit with logic functions rather than gate structure
- Uses different operators to produce the desired results
- Dataflow modeling provides a powerful way to implement a design.
- Higher level of abstraction than gate level modeling
- Gate level modeling is preferred in simpler circuits where as Dataflow is preferred in complex circuit
- 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.
- Automated tools are used to create a gate-level circuit from a dataflow design description.
- This process is called logic synthesis
- RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling

Dataflow Modeling – continuous assignment

- A continuous assignment is the most basic statement in dataflow modeling, 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 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;
```

- The left hand side (LHS) 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
- Right hand side (RHS) is a Boolean expression with operators and operands
- Whenever any operand changes its value the RHS is computed and the computed value is assigned to LHS

Dataflow Modeling – continuous assignment

- A continuous assignment is the most basic statement in dataflow modeling, 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.

  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
- drive strength is optional
- 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
- 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

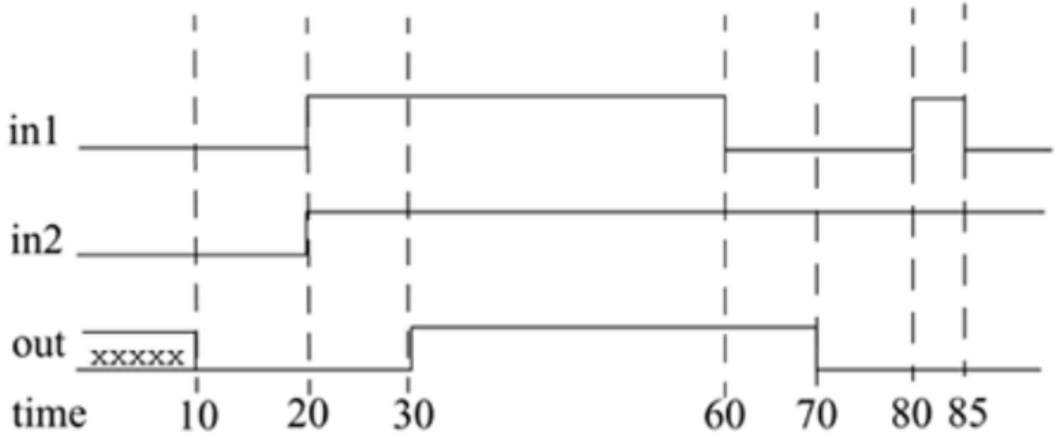
Dataflow Modeling – Delay

- 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
  - regular assignment delay
  - implicit continuous assignment delay
  - net declaration delay

Regular Assignment Delay

- Any change in values of in1 or in2 will result in a delay of re-computation of the expression in1 & in2, and the result will be assigned to out

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



Dataflow Modeling – Delay

Implicit Continuous Assignment Delay

Implicit continuous assignment 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;
```

Net Declaration Delay

A delay can be specified on a net when it is declared without putting a continuous assignment on the net.

```
//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;
```

Dataflow Modeling – Examples

4-to-1 Multiplexer, 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
```

4-to-1 Multiplexer, Using Conditional Operators

```
// 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
```

Dataflow Modeling – Examples

4-bit Full Adder with dataflow operators

```
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
```

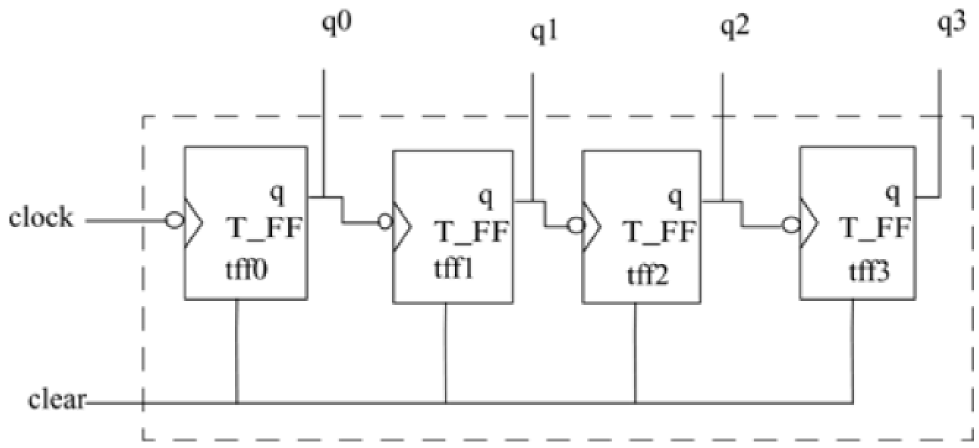
4-bit Full Adder with Carry Lookahead

```
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
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
```

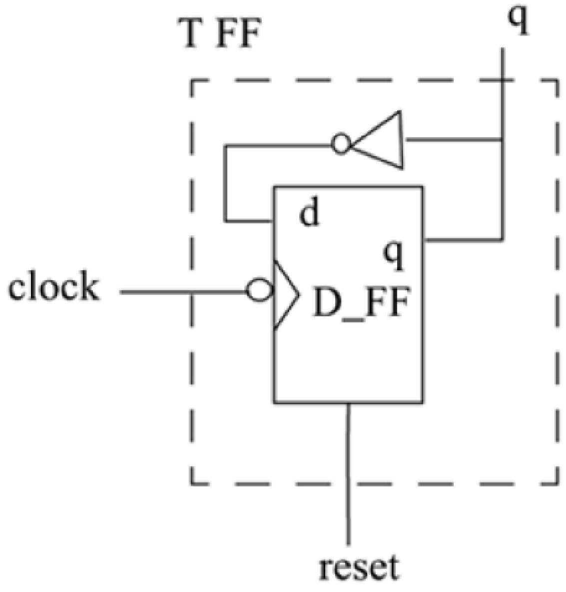
Dataflow Modeling – Examples

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
```



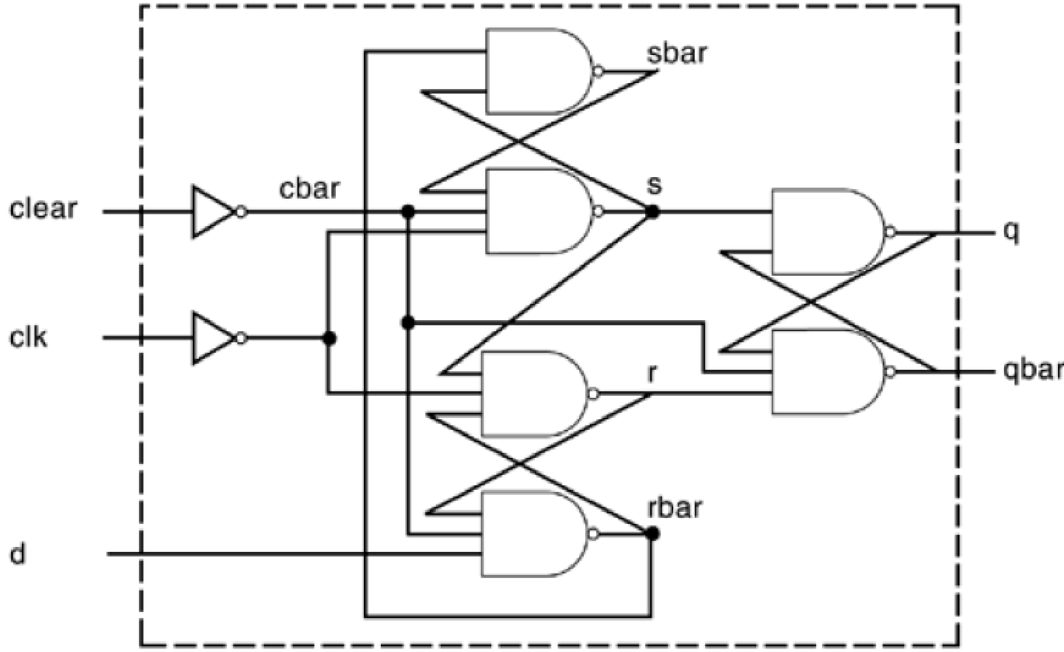

```
// Edge-triggered T-flipflop. Toggles every clock cycle.
module T_FF(q, clk, clear);
// I/O ports
output q;
input clk, clear;
// Instantiate the edge-triggered DFF
edge_dff ff1(q, ,~q, clk, clear);
endmodule
```



Dataflow Modeling – Examples

Verilog Code for Ripple Counter

```
// 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.
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
```



Dataflow Modeling – Examples

Stimulus Module for Ripple Counter

```
module stimulus;
reg CLOCK, CLEAR;
wire [3:0] Q;
initial
$monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);
counter c1(Q, CLOCK, CLEAR);
initial
begin
CLEAR = 1'b1;
#34 CLEAR = 1'b0;
#200 CLEAR = 1'b1;
#50 CLEAR = 1'b0;
end
initial
begin
CLOCK = 1'b0;
forever #10 CLOCK = ~CLOCK;
end
initial
begin
#400 $finish;
end
endmodule
```

Output

```
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
```