שפת תכנון חומרה
ורילוג - Verilog

Verilog for Sequential Circuits

Dr. Avihai Aharon

What will we learn?

- **Incomplete specification**
- **Sequential Logic in Verilog**
- **Using Sequential Constructs for Combinational Design**

Summary: Verilog Number Representation

N'Bxx : 8'b0000_0001

(N) Number of bits : Expresses how many bits will be used to store the value

(B) Base : Can be b (binary), h (hexadecimal), d (decimal), o (octal)

(xx) Number :
- value expressed in base, it can also have X and Z as values.
- Underscore _ can be used to improve readability

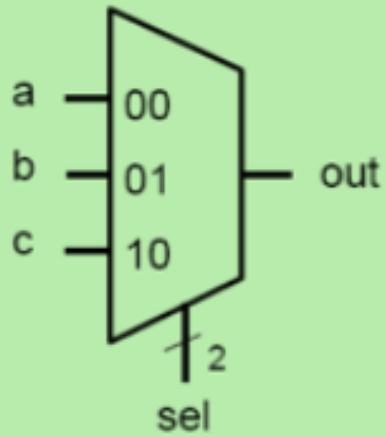
Verilog	Stored Number	Verilog	Stored Number
4'b1001	1001	4'd5	0101
8'b1001	0000 1001	12'hFA3	1111 1001 0011
8'b0000_1001	0000 1001	8'o12	00 001 010
8'bxX0X1zZ1	XX0X 1ZZ1	4'h7	0111
'b01	0000 .. 0001	12'h0	0000 0000 0000

Precedence of Operations in Verilog

Highest	~	NOT
	*, /, %	mult, div, mod
	+, -	add,sub
	<<, >>	shift
	<<<, >>>	arithmetic shift
	<, <=, >, >=	comparison
	==, !=	equal, not equal
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Lowest	?:	ternary operator

Beware of Incomplete Specification

Intention



3-to-1 MUX
('11' input is a don't-care)

What you may write, but wrong!

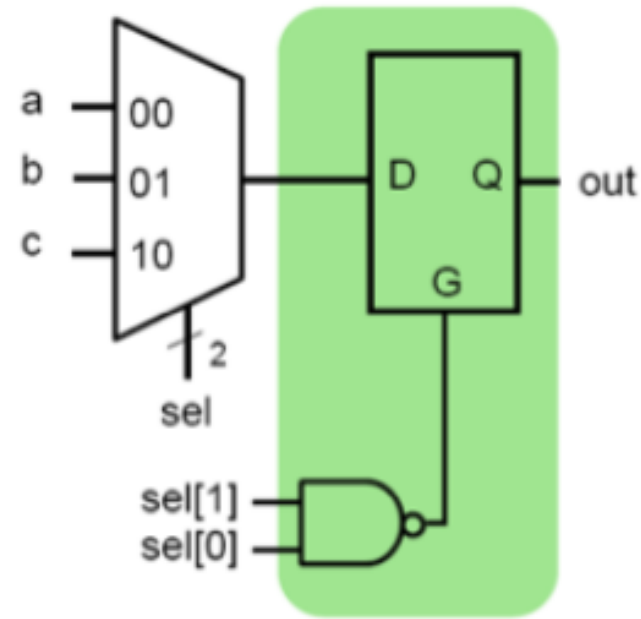
```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

Incomplete Specification:
adds unwanted latch circuit

if `out` is not assigned during any pass through the always block, then the previous value must be retained!

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

Synthesized Circuit



- When sel = 2'b11, G = 0, therefore the latch stores the previous output value as required by Verilog in this situation.

Always avoid Incomplete Specification

- Solution 1: Precede all conditionals with a default assignment for all signals:

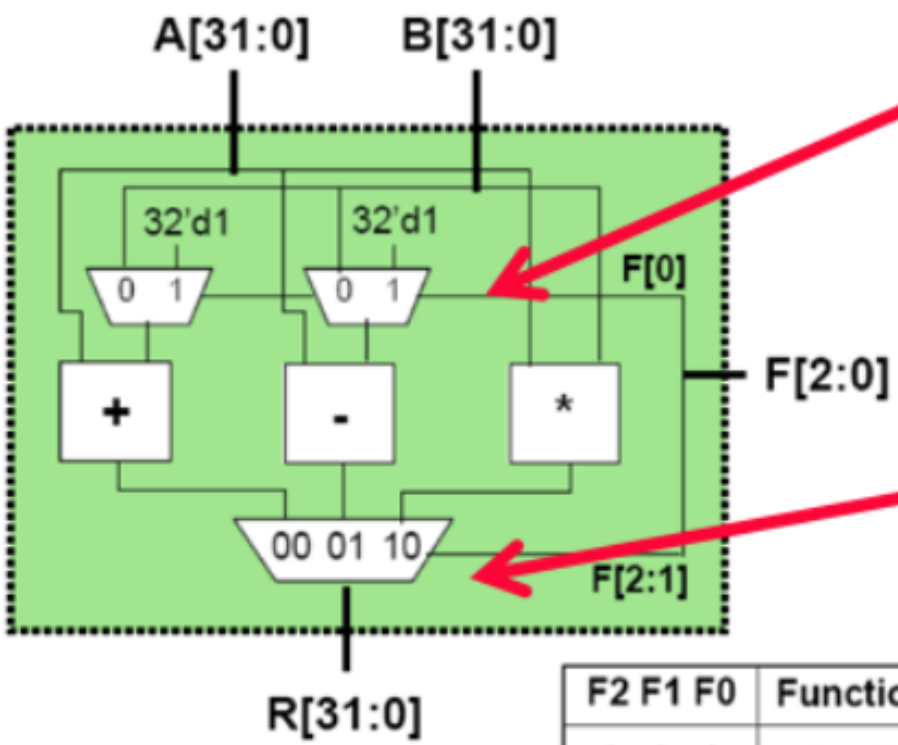
```
always @(a or b or c or sel)
  begin
    out = 1'bx;
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

- Solution 2: Fully specify all branches of if-else construct, or include a default statement in case construct:

```
always @(a or b or c or sel)
  begin
  case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
      default: out = 1'bx;
    endcase
  end
endmodule
```

A larger example: 32-bits ALU

- Here is an 32-bit ALU with 5 simple instructions:



F2 F1 F0	Function
0 0 0	A + B
0 0 1	A + 1
0 1 0	A - B
0 1 1	A - 1
1 0 X	A * B

2-to-1 MUX

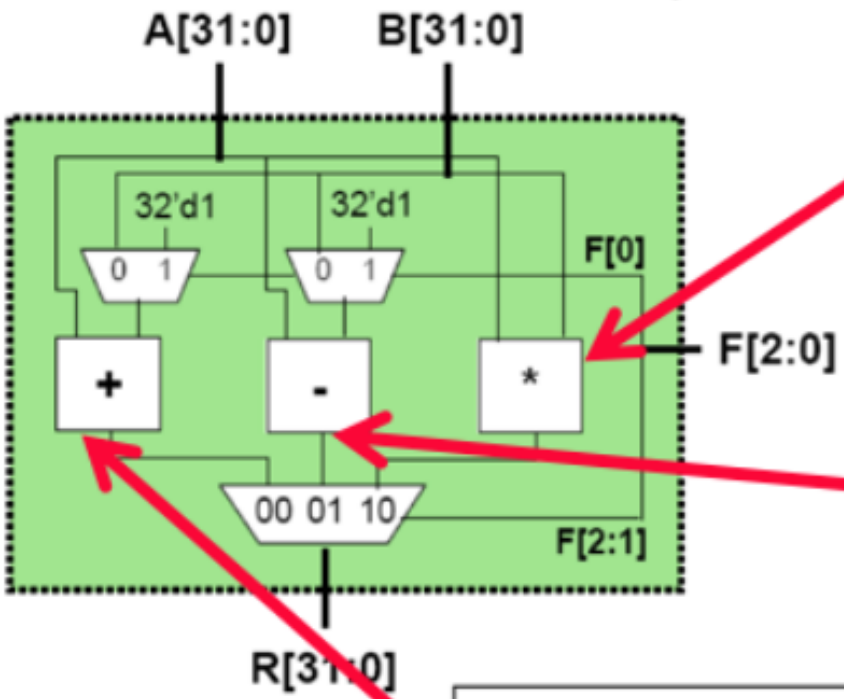
```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX

```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;
always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    default: out = 32'bx;
  endcase
end
endmodule
```

The Arithmetic modules

- Here is an 32-bit ALU with 5 simple instructions:




```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

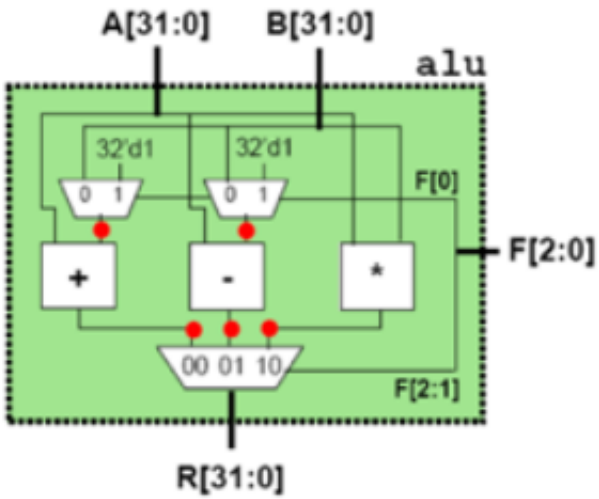
```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```

Top level module - putting them together

- Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```




```
module alu(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;

  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;

  mux32two    adder_mux(b, 32'd1, f[0], addmux_out);
  mux32two    sub_mux(b, 32'd1, f[0], submux_out);
  add32       our_adder(a, addmux_out, add_out);
  sub32       our_subtracter(a, submux_out, sub_out);
  mul16       our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three  output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule
```

intermediate output nodes

module names

(unique) instance names

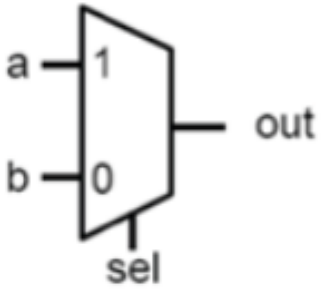
corresponding wires/regs in module alu

How to specify a sequential circuit

- Edge-triggered flipflop is specified with: **always @ (posedge clk):**

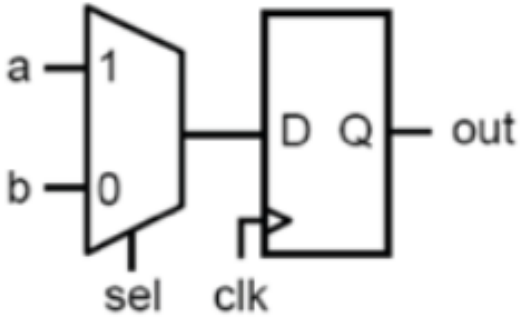
Combinational cct

```
module combinational(a, b, sel,
                                out);
  input a, b;
  input sel;
  output out;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```



Sequential cct

```
module sequential(a, b, sel,
                    clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;

  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```



Sequential Logic in Verilog

- **Define blocks that have memory**
  - Flip-Flops, Latches, Finite State Machines
- **Sequential Logic is triggered by a 'CLOCK' event**
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of clock
- **Combinational constructs are not sufficient**
  - We need new constructs:
    - `always`
    - `initial`

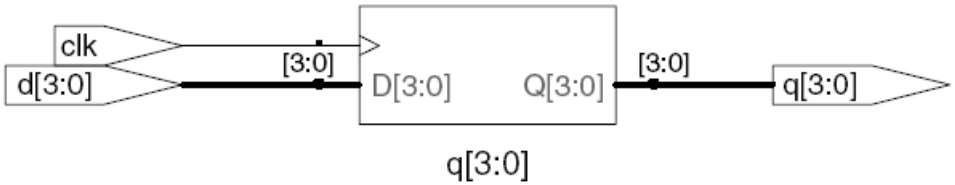
always Statement, Defining Processes

```
always @ (sensitivity list)
      statement;
```

- **Whenever the event in the sensitivity list occurs, the statement is executed**

Example: D Flip-Flop

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```



Example: D Flip-Flop

```
module flop(input              clk,
            input       [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                    // pronounced “q gets d”

endmodule
```

- **The posedge defines a rising edge (transition from 0 to 1).**
- **This process will trigger only if the clk signal rises.**
- **Once the clk signal rises: the value of d will be copied to q**

Example: D Flip-Flop

```
module flop(input              clk,
            input       [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                    // pronounced “q gets d”

endmodule
```

- **‘assign’ statement is not used within always block**
- **The <= describes a ‘non-blocking’ assignment**
  - We will see the difference between ‘blocking assignment’ and ‘non-blocking’ assignment in a while

Example: D Flip-Flop

```
module flop(input              clk,
            input       [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                    // pronounced “q gets d”

endmodule
```

- **Assigned variables need to be declared as reg**
- **The name reg does not necessarily mean that the value is a register. (It could be, it does not have to be).**
- **We will see examples later**

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input            clk,
                input            reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
       if (reset == ‘0’) q <= 0;   // when reset
       else              q <= d;   // when clk
    end
endmodule
```

- **In this example: two events can trigger the process:**
  - A ***rising edge*** on clk
  - A ***falling edge*** on reset

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input            clk,
                input            reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == ‘0’) q <= 0;   // when reset
      else              q <= d;   // when clk
    end
endmodule
```

- **For longer statements a begin end pair can be used**
  - In this example it was not necessary
- **The always block is *highlighted***

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input            clk,
                input            reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == '0') q <= 0;   // when reset
      else              q <= d;   // when clk
    end
endmodule
```

- **First reset is checked, if reset is 0, q is set to 0.**
  - This is an 'asynchronous' reset as the reset does not care what happens with the clock
- **If there is no reset then normal assignment is made**

D Flip-Flop with Synchronous Reset

```
module flop_sr (input              clk,
                input              reset,
                input       [3:0]  d,
                output reg  [3:0]  q);

  always @ (posedge clk)
    begin
       if (reset == ‘0’) q <= 0;   // when reset
       else              q <= d;   // when clk
    end
endmodule
```

- **The process is only sensitive to clock**
  - Reset ***only happens*** when the ***clock rises***. This is a ‘synchronous’ reset

- **A small change, has a large impact on the outcome**

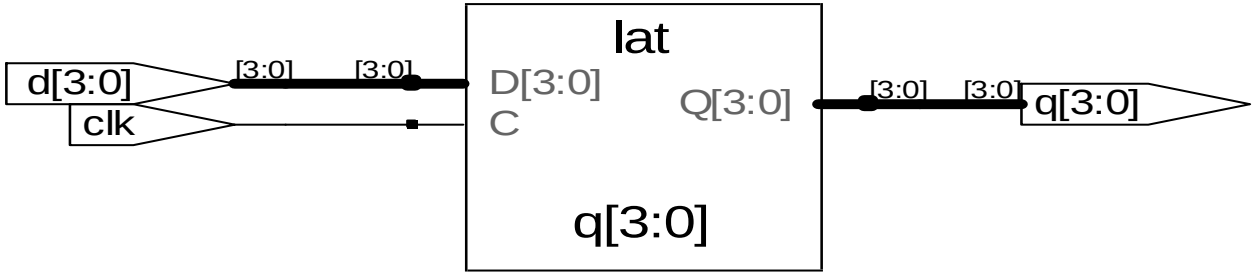
D Flip-Flop with Enable and Reset

```
module flop_ar (input            clk,
                input            reset,
                input            en,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk. negedge reset)
    begin
       if (reset == ‘0’) q <= 0;   // when reset
       else if (en)      q <= d;   // when en AND clk
    end
endmodule
```

- **A flip-flop with enable and reset**
  - Note that the en signal is ***not*** in the sensitivity list
- **Only when “clk is rising” *AND* “en is 1” data is stored**

Example: D Latch

```
module latch (input            clk,
              input      [3:0] d,
              output reg [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;       // latch is transparent when
                           // clock is 1
endmodule
```



Summary: Sequential Statements so far

- Sequential statements are within an '`always`' block
- The sequential block is triggered with a change in the sensitivity list
- Signals assigned within an always must be declared as `reg`
- We use `<=` for (non-blocking) assignments and do not use '`assign`' within the always block.

Summary: Basics of always Statements

```
module example (input            clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;          // standard wire
  reg  [3:0] special;         // assigned in always

  always @ (posedge clk)
    special <= d;             // first FF array

  assign normal = ~ special; // simple assignment

  always @ (posedge clk)
    q <= normal;              // second FF array
endmodule
```

- **You can have many always blocks**

Summary: Basics of always Statements

```
module example (input            clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;          // standard wire
  reg  [3:0] special;         // assigned in always

  always @ (posedge clk)
    special <= d;             // first FF array

  assign normal = ~ special;  // simple assignment

  always @ (posedge clk)
    q <= normal;              // second FF array
endmodule
```

- **Assignments are different within always blocks**

Why does an always Statement Memorize?

```
module flop (input            clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

- **This statement describes what happens to signal q**
- **… but what happens when clock is not rising?**

Why does an always Statement Memorize?

```
module flop (input            clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

- **This statement describes what happens to signal q**
- **… but what happens when clock is not rising?**
- **The value of q is preserved (memorized)**

Why does an always Statement Memorize?

```
module comb (input              inv,
             input       [3:0] data,
             output reg [3:0] result);

  always @ (inv, data)        // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

- **This statement describes what happens to signal result**
  - When inv is 1, result is ~data
  - What happens when inv is ***not 1*** ?

Why does an always Statement Memorize?

```
module comb (input              inv,
             input       [3:0] data,
             output reg [3:0] result);

  always @ (inv, data)        // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

- **This statement describes what happens to signal result**
  - When `inv` is 1, result is `~data`
  - When `inv` is not 1, result is `data`
- **Circuit is combinational (no memory)**
  - The output (`result`) is defined for all possible inputs (`inv data`)

always Blocks for Combinational Circuits

- **If the statements define the signals completely, nothing is memorized, block becomes combinational.**
  - Care must be taken, it is easy to make mistakes and unintentionally describe memorizing elements (latches).
- **Always blocks allow powerful statements**
  - `if .. then .. else`
  - `case`
- **Use always blocks only if it makes your job easier**

Always Statement is not Always Practical…

```
  reg  [31:0] result;
  wire [31:0] a, b, comb;
  wire        sel,

  always @ (a, b, sel)    // trigger with a, b, sel
    if (sel) result <= a; // result is a
    else     result <= b; // result is b

  assign comb = sel ? a : b;

endmodule
```

- **Both statements describe the same multiplexer**
- **In this case, the always block is more work**

Sometimes Always Statements are Great

```
module sevensegment (input      [3:0] data,
                     output reg [6:0] segments);

  always @ ( * )                    // * is short for all signals
    case (data)                     // case statement
      0: segments = 7'b111_1110;    // when data is 0
      1: segments = 7'b011_0000;    // when data is 1
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      // etc etc
      default: segments = 7'b000_0000; // required
    endcase

endmodule
```

The case Statement

- **Like `if .. then .. else` can only be used in always blocks**
- **The result is combinational only if the output is defined for all cases**
  - Did we mention this before ?
- **Always use a `default` case to make sure you did not forget a case (which would infer a latch)**
- **Use `casez` statement to be able to check for don't cares**

Non-blocking and Blocking Statements

Non-blocking

```
always @ (a)
begin
   a <= 2'b01;
   b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- **Values are assigned at the end of the block.**
- **All assignments are made in parallel, process flow is not-blocked.**

Blocking

```
always @ (a)
begin
   a = 2'b01;
// a is 2'b01
   b = a;
// b is now 2'b01 as well
end
```

- **Value is assigned immediately.**
- **Process waits until the first assignment is complete, it blocks progress.**

Example: Blocking Statements

- **Assume all inputs are initially '0'**

```
always @ ( * )
  begin
    p    = a ^ b ;          // p    = 0
    g    = a & b ;          // g    = 0
    s    = p ^ cin ;        // s    = 0
    cout = g | (p & cin) ; // cout = 0
  end
```

Example: Blocking Statements

- Now a changes to '1'

```
always @ ( * )
  begin
    p    = a ^ b ;          // p    = 1
    g    = a & b ;          // g    = 0
    s    = p ^ cin ;        // s    = 1
    cout = g | (p & cin) ; // cout = 0
  end
```

- The process triggers
- All values are updated in order
- At the end, s = 1

Same Example: Non-Blocking Statements

- **Assume all inputs are initially '0'**

```
always @ ( * )
  begin
    p    <= a ^ b ;          // p    = 0
    g    <= a & b ;          // g    = 0
    s    <= p ^ cin ;        // s    = 0
    cout <= g | (p & cin) ; // cout = 0
  end
```

Same Example: Non-Blocking Statements

- Now a changes to '1'

```
always @ ( * )
  begin
    p    <= a ^ b ;          // p    = 1
    g    <= a & b ;          // g    = 0
    s    <= p ^ cin ;        // s    = 0
    cout <= g | (p & cin) ; // cout = 0
  end
```

- The process triggers
- All assignments are concurrent
- When s is being assigned, p is still 0, result is still 0

Same Example: Non-Blocking Statements

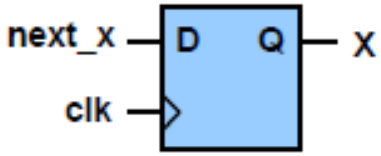
- After the first iteration p has changed to '1' as well

```
always @ ( * )
  begin
    p    <= a ^ b ;          // p    = 1
    g    <= a & b ;          // g    = 0
    s    <= p ^ cin ;        // s    = 1
    cout <= g | (p & cin) ; // cout = 0
  end
```

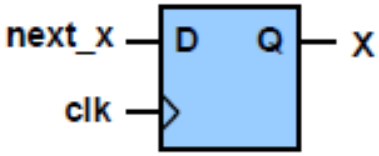
- Since there is a change in p, process triggers again
- This time s is calculated with p=1
- The result is correct after the second iteration

Another Example: Non-Blocking Statements

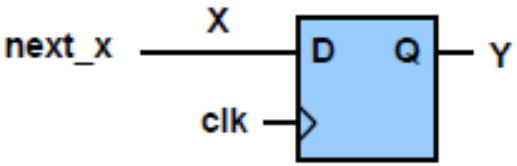
- ```
  always @( posedge clk )
  begin
    x = next_x;
  end
  ```

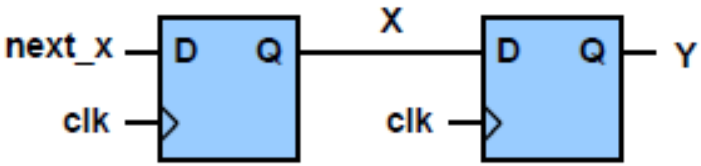

  ```
  always @( posedge clk )
  begin
    x <= next_x;
  end
  ```


- ```
  always @( posedge clk )
  begin
    x = next_x;
    y = x;
  end
  ```


  ```
  always @( posedge clk )
  begin
    x <= next_x;
    y <= x;
  end
  ```

Rules for Signal Assignment

- **Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model synchronous sequential logic**

```
always @ (posedge clk)
    q <= d; // nonblocking
```

- **Use continuous assignments (`assign …`)to model simple combinational logic.**

```
assign y = a & b;
```

Rules for Signal Assignment (cont)

- Use `always @ (*)` and blocking assignments (`=`) to model more complicated combinational logic where the always statement is helpful.

- Do not make assignments to the same signal in more than one always statement or continuous assignment statement