Verilog Basics and RTL Simulation

Song Chen

Dept. of Electronic Science & Technology, USTC

September 24, 2015

http://staff.ustc.edu.cn/~songch/da-ug.htm

Outline

- Verilog Basics
  - Data Type
  - Structural Verilog
  - Simple Behavior

- Verilog Execution Semantics
  - Driven by simulation
  - Explained using event queues

Bit-vector is the only data type in Verilog

A bit can take on one of four values

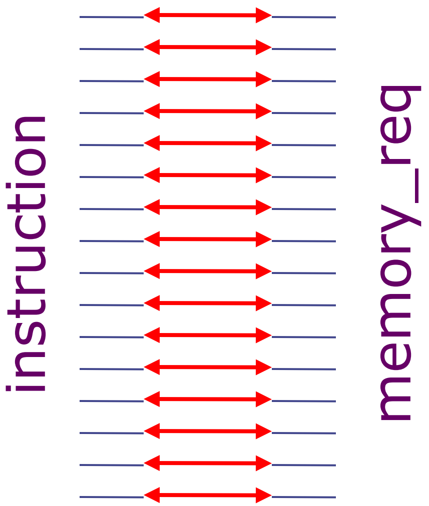
Value	Meaning
0	Logic zero
1	Logic one
X	Unknown logic value
Z	High impedance, floating

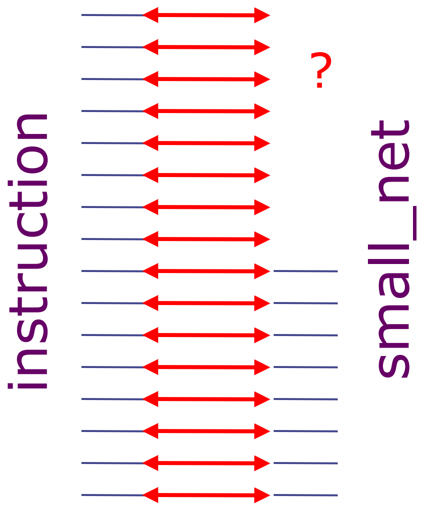
An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’t care what the value is. This can help catch bugs and improve synthesis quality.

“wire” is used to denote a hardware net (connection)

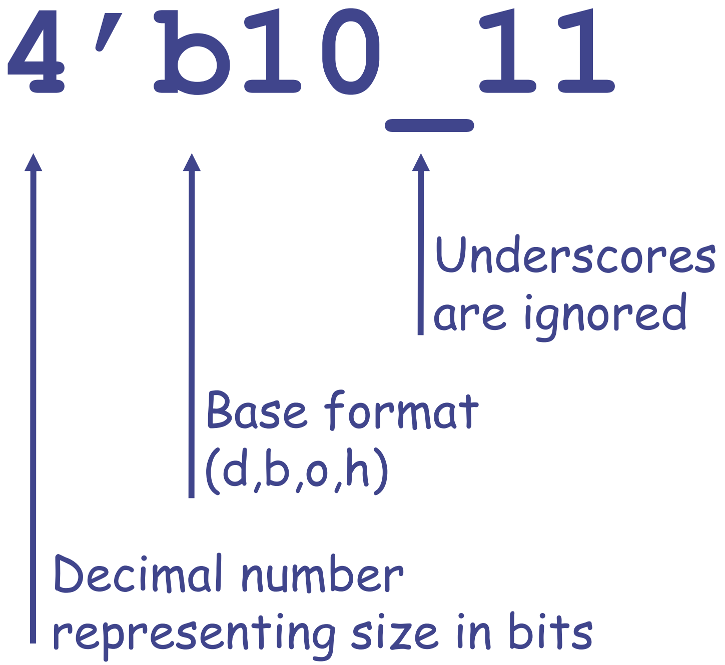
```
wire [15:0] instruction;
wire [15:0] memory_req;
wire [  7:0] small_net;
```

Absolutely no type safety when connecting nets!





Bit literals



`4’b10_11`

Underscores are ignored

Base format (d,b,o,h)

Decimal number representing size in bits

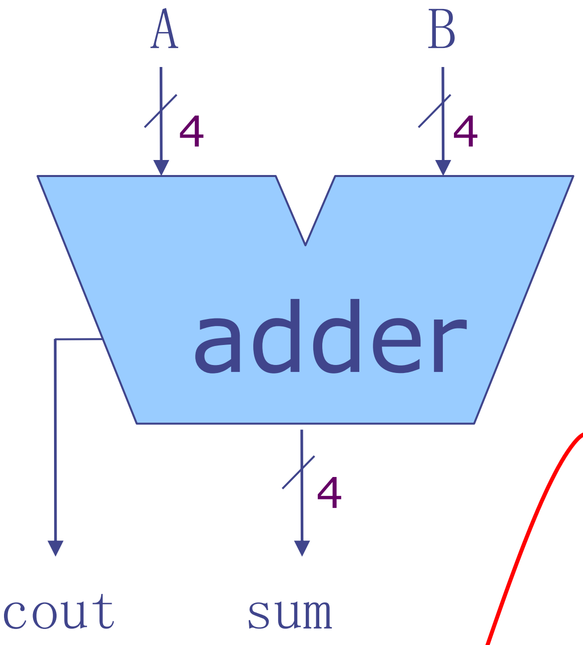
We’ll learn how to actually assign literals to nets a little later

- Binary literals
  - `8’b0000_0000`
  - `8’b0xx0_1xx1`
- Hexadecimal literals
  - `32’h0a34_def1`
  - `16’haxxx`
- Decimal literals
  - `32’d42`

Outline

- Verilog Basics
  - Data Type
  - Structural Verilog
  - Simple Behavior
- Verilog Execution Semantics
  - Driven by simulation
  - Explained using event queues

A Verilog module has a name and a port list

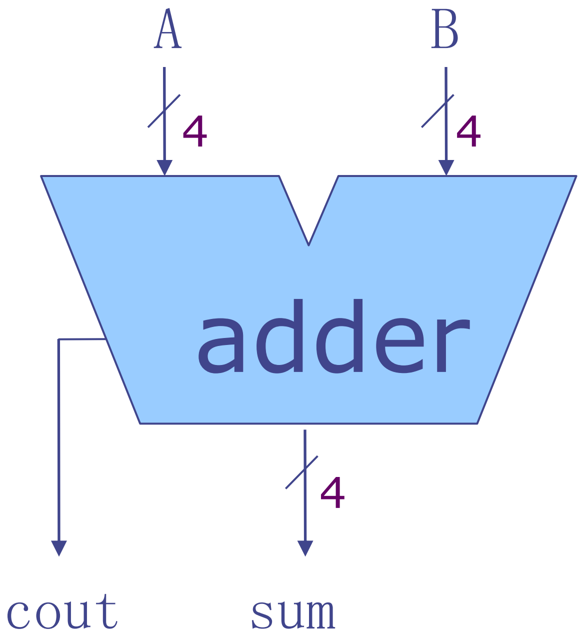


```
module adder( A, B, cout, sum );
  input  [3:0] A;
  input  [3:0] B;
  output       cout;
  output [3:0] sum;

  // HDL modeling of
  // adder functionality

endmodule
```

Ports must have a direction (or be bidirectional) and a bitwidth

Note the semicolon at the end of the port list!

Alternate syntax



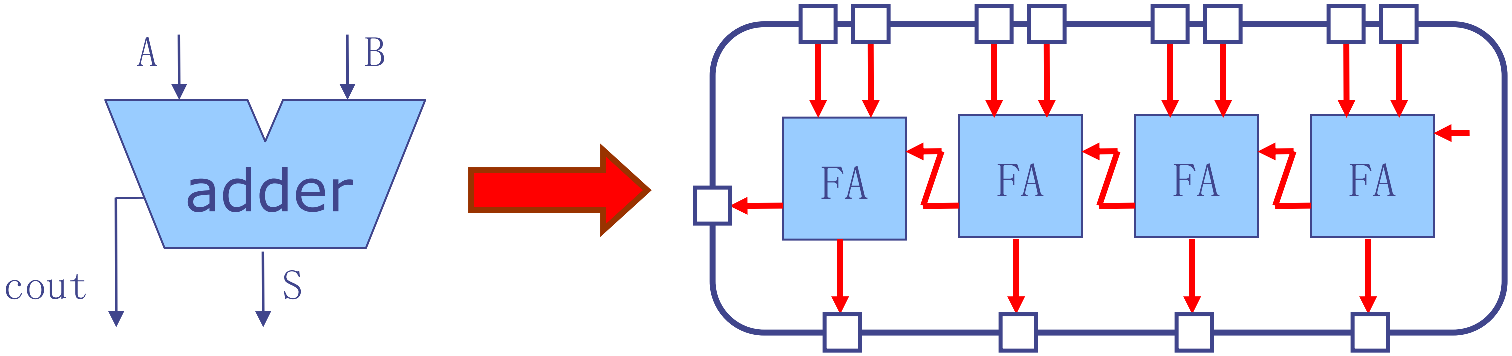
Traditional Verilog-1995 Syntax

```
module adder( A, B, cout, sum );
  input  [3:0] A;
  input  [3:0] B;
  output       cout;
  output [3:0] sum;
```

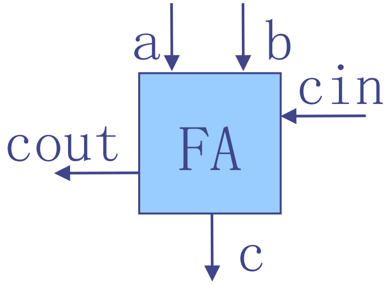
ANSI C Style Verilog-2001 Syntax

```
module adder( input  [3:0] A,
              input  [3:0] B,
              output       cout,
              output [3:0] sum );
```

A module can instantiate other modules



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );
```

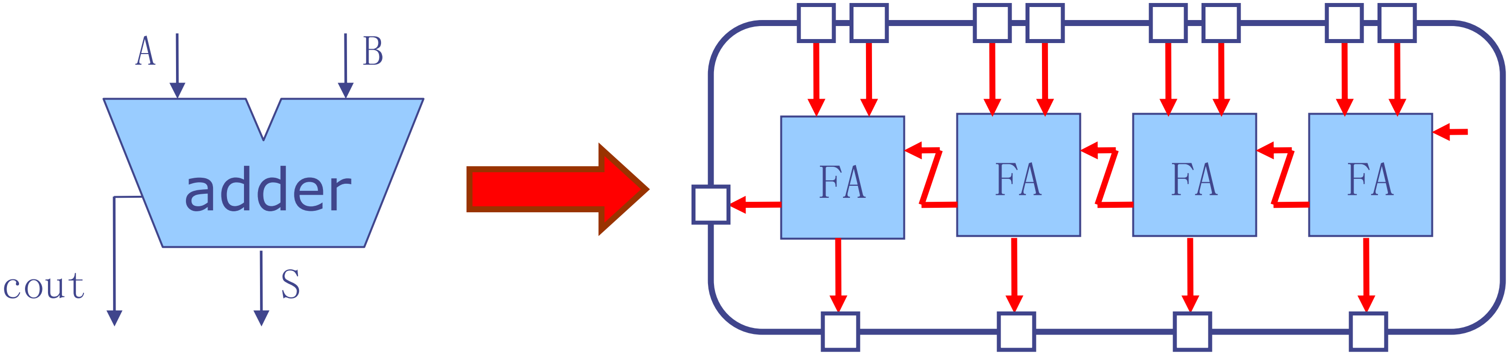


```
 wire c0, c1, c2;
 FA fa0( ... );
 FA fa1( ... );
 FA fa2( ... );
 FA fa3( ... );

endmodule
```

```
module FA( input  a, b, cin
           output cout, sum );
  // HDL modeling of 1 bit
  // full adder functionality
endmodule
```

```
assign {cout, s} = A + B;
```

Connecting modules



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

  wire c0, c1, c2;
  FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
  FA fa1( A[1], B[1], c0,   c1,   S[1] );
  FA fa2( A[2], B[2], c1,   c2,   S[2] );
  FA fa3( A[3], B[3], c2,   cout, S[3] );

endmodule
```

Carry Chain

Alternative syntax

Connecting ports by ordered list

```
FA fa0( A[0], B[0], 1’b0, c0, S[0] );
```

Connecting ports by name (compact)

```
FA fa0( .a(A[0]), .b(B[0]),
        .cin(1’b0), .cout(c0), .sum(S[0]) );
```

Argument order does not matter when ports are connected by name

```
FA fa0
( .a    (A[0]),
  .cin  (1’b0),
  .b    (B[0]),
  .cout (c0),
  .sum  (S[0]) );
```

Connecting ports by name yields clearer and less buggy code.

Outline

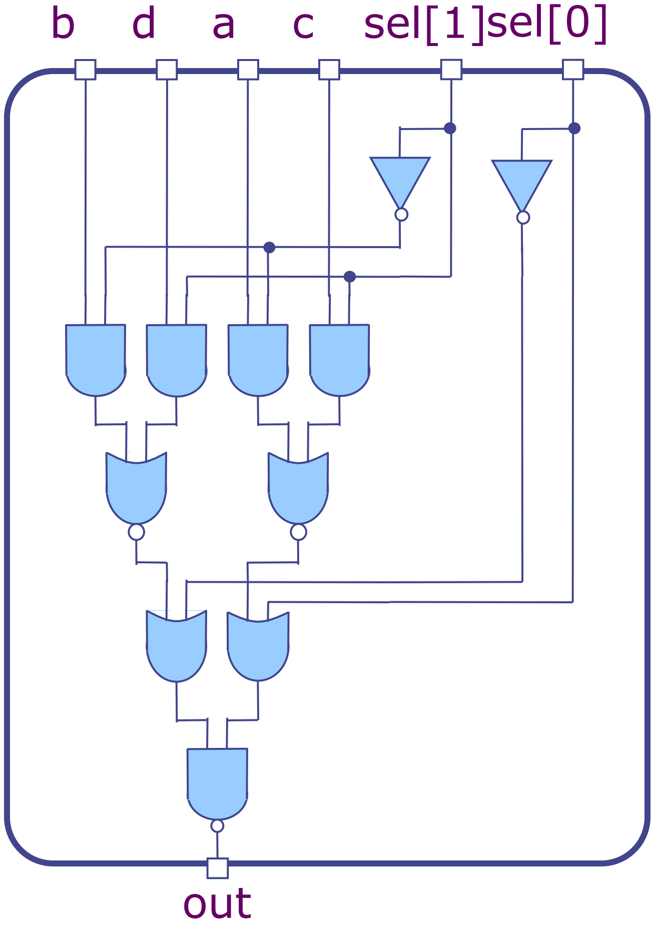
- Verilog Basics
  - Data Type
  - Structural Verilog
  - Simple Behavior
- Verilog Execution Semantics
  - Driven by simulation
  - Explained using event queues

A module’s behavior can be described in many different ways but it should not matter from outside

Example: 4-1 multiplexor

mux4: Gate-level structural Verilog

```
module mux4(input a,b,c,d, input [1:0] sel, output out);
  wire [1:0] sel_b;
  not not0( sel_b[0], sel[0] );
  not not1( sel_b[1], sel[1] );

  wire n0, n1, n2, n3;
  and and0( n0, c, sel[1]   );
  and and1( n1, a, sel_b[1] );
  and and2( n2, d, sel[1]   );
  and and3( n3, b, sel_b[1] );

  wire x0, x1;
  nor nor0( x0, n0, n1 );
  nor nor1( x1, n2, n3 );

  wire y0, y1;
  or or0( y0, x0, sel[0]   );
  or or1( y1, x1, sel_b[0] );
  nand nand0( out, y0, y1 );
endmodule
```



mux4: Using continuous assignments

Language defined operators

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire out, t0, t1;
  assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );

endmodule
```

The order of these continuous assignment statements does not matter. They essentially happen in parallel!

mux4: Behavioral style

```
// Four input multiplexer
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  assign out = ( sel == 0 ) ? a :
               ( sel == 1 ) ? b :
               ( sel == 2 ) ? c :
               ( sel == 3 ) ? d : 1’bx;

endmodule
```

If input is undefined
we want to propagate
that information.

mux4: Using “always block”

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out, t0, t1;

  always @( a or b or c or d or sel )
  begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
  end

endmodule
```

Motivated by simulation

The order of these procedural assignment statements DOES matter. They essentially happen sequentially!

“Always blocks” permit more advanced sequential idioms

```
module mux4( input  a,b,c,d
             input [1:0] sel,
             output out );
  reg out;
  always @( * )
  begin
    if ( sel == 2’d0 )
      out = a;
    else if ( sel == 2’d1 )
      out = b;
    else if ( sel == 2’d2 )
      out = c;
    else if ( sel == 2’d3 )
      out = d;
    else
      out = 1’bx;
  end
endmodule
```

```
module mux4( input  a,b,c,d
             input [1:0] sel,
             output out );
  reg out;
  always @( * )
   begin
    case ( sel )
      2’d0 : out = a;
      2’d1 : out = b;
      2’d2 : out = c;
      2’d3 : out = d;
      default : out = 1’bx;
    endcase
  end
endmodule
```

Typically we will use always blocks only to describe sequential circuits

What happens if the case statement is not complete?

```
module mux3( input  a, b, c
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2’d0 : out = a;
      2’d1 : out = b;
      2’d2 : out = c;
    endcase
  end

endmodule
```

If sel = 3, mux will output the previous value!

What have we created?

What happens if the case statement is not complete?

```
module mux3( input  a, b, c
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      2′d0 : out = a;
      2′d1 : out = b;
      2′d2 : out = c;
      default : out = 1′bx;
    endcase
  end

endmodule
```

We CAN prevent creating state with a default statement

Parameterized mux4

default value

```
module mux4 #( parameter WIDTH = 1 )
            ( input[WIDTH-1:0]  a, b, c, d
              input [1:0] sel,
              output[WIDTH-1:0] out );

  wire [WIDTH-1:0] out, t0, t1;

  assign t0  = (sel[1]? c : a);
  assign t1  = (sel[1]? d : b);
  assign out = (sel[0]? t0: t1);
endmodule
```

Parameterization is a good practice for reusable modules

Writing a mux*n* is challenging

Instantiation Syntax

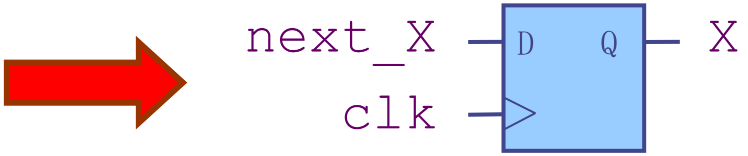
```
mux4#(32) alu_mux
(  .a (op1),
   .b (op2),
   .c (op3),
   .d (op4),
   .sel (alu_mux_sel),
   .out (alu_mux_out) );
```

Verilog Registers “reg”

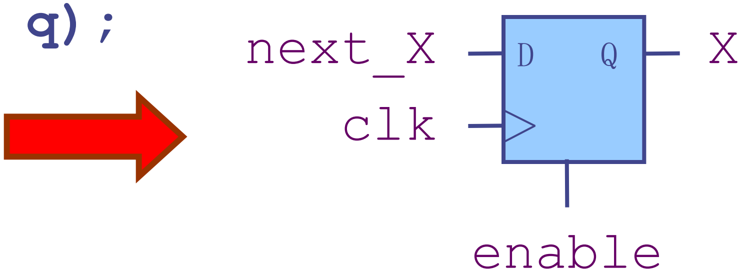
- Wires are line names – they do not represent storage and can be assigned only once
- Regs are imperative variables (as in C):
  - “nonblocking” assignment `r <= v`
  - can be assigned multiple times and holds values between assignments

flip-flops

```
module FF0 (input clk, input  d,
            output reg q);
always @( posedge clk )
  begin
     q <= d;
  end
endmodule
```



```
module FF (input clk, input  d,
           input  en, output reg q);
always @( posedge clk )
  begin
    if ( en )
     q <= d;
  end
endmodule
```



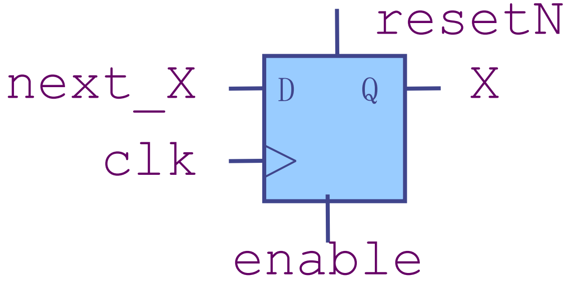
flip-flops with reset

```
always @( posedge clk)
begin
  if (~resetN)
    Q <= 0;
  else if ( enable )
    Q <= D;
end
```

synchronous reset

```
always @( posedge clk or
          negedge resetN)
begin
  if (~resetN)
    Q <= 0;
  else if ( enable )
    Q <= D;
end
```

asynchronous reset



What is the difference?

Latches versus flip-flops

```
module latch
(
  input  clk,
  input  d,
  output reg q
);

  always @( clk or d )
  begin
    if ( clk )
      q <= d;
  end

endmodule
```

```
module flipflop
(
  input  clk,
  input  d,
  output reg q
);

  always @( posedge clk )
  begin
    q <= d;
  end

endmodule
```

Edge-triggered always block

Register

```
module register#(parameter WIDTH = 1)
(
  input  clk,
  input  [WIDTH-1:0] d,
  input  en,
  output [WIDTH-1:0] q
);

  always @( posedge clk )
  begin
    if (en)
      q <= d;
  end

endmodule
```

Register in terms of Flipflops

```
module register2
( input  clk,
  input  [1:0] d,
  input  en,
  output [1:0] q );

  always @(posedge clk)
  begin
    if (en)
     q <= d;
  end

endmodule
```

```
module register2
( input  clk,
  input  [1:0] d,
  input  en,
  output [1:0] q
);
 FF ff0 (.clk(clk), .d(d[0]),
         .en(en),   .q(q[0]));

 FF ff1 (.clk(clk), .d(d[1]),
         .en(en),   .q(q[1]));

endmodule
```

Do they behave the same?

yes

Register file with 2 combinational read ports and 1 write port

```
module smipsProcDpathRegfile
( input          clk,
  input  [ 4:0] raddr0,  // Read 0 address (combinational input)
  output [31:0] rdata0,  // Read 0 data (combinational on raddr)
  input  [ 4:0] raddr1,  // Read 1 address (combinational input)
  output [31:0] rdata1,  // Read 1 data (combinational on raddr)
  input          wen_p,   // Write enable (sample on rising clk edge)
  input  [ 4:0] waddr_p, // Write address(sample on rising clk edge)
  input  [31:0] wdata_p  // Write data (sample on rising clk edge));

  // We use an array of 32 bit register for the regfile itself
  reg [31:0] registers[31:0];

  // Combinational read ports
  assign rdata0 = registers[raddr0];
  assign rdata1 = registers[raddr1];

  // Write port is active only when wen is asserted
  always @( posedge clk )
    if ( wen_p)
      registers[waddr_p] <= wdata_p;
endmodule
```

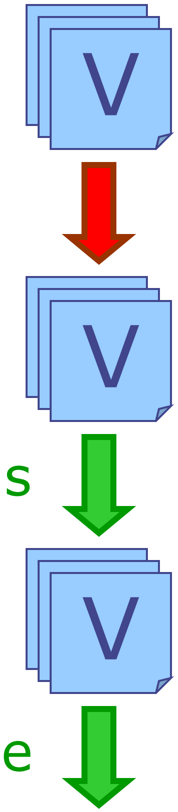
Static Elaboration: Generate

```
module register#(parameter WIDTH = 1)
( input  clk,
  input  [WIDTH-1:0] d,
  input  en,
  output [WIDTH-1:0] q
);

 genvar i;
 generate
 for (i =0; i < WIDTH; i = i + 1)
   begin: regE
    FF ff(.clk(clk), .d(d[i]), .en(en), .q(q[i]));
   end
 endgenerate
endmodule
```

genvars disappear after static elaboration

Generated names will have `regE[i].` prefix

Three abstraction levels for functional descriptions



Behavioral
Algorithm

Abstract algorithmic
description

Manual &
HLS

Register
Transfer Level

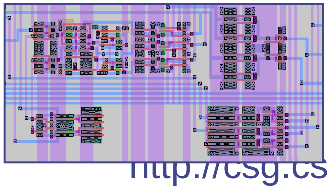
Describes how data flows
between state elements for each
cycle

Logic Synthesis

Gate Level

Low-level netlist
of primitive gates

Auto Place + Route



Next time
Some examples

Guidelines for writing synthesizable (合成可能) Verilog

- Combinational logic:
  - Use continuous assignments (**assign**)

```
assign C_in = B_out + 1;
```

  - Use **always@(*)** blocks with blocking assignments (**=**)

```
  always @(*)
   begin
     out = 2’d0;
     if (in1 == 1)
         out = 2’d1;
     else if (in2 == 1)
         out = 2’d2;
   end
```

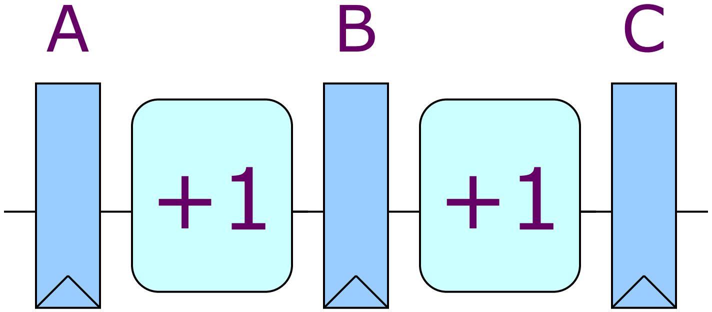
- Sequential logic:
  - Use **always @(posedge clk)** and non-blocking assignments (**<=**)

```
 always @( posedge clk )
C_out <= C_in;
```

  - Use only positive-edge triggered flip-flops for state
  - Do not assign the same variable from more than one always block
- Only leaf modules should have functionality; use higher-level modules only for wiring together sub-modules

An example

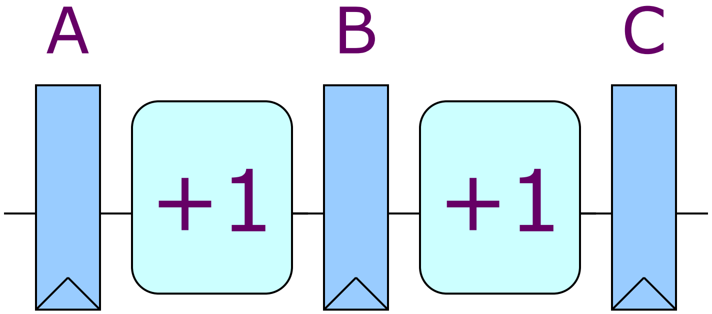
```
module non_block #(parameter WIDTH = 4)
(input clk,
 input [WIDTH-1:0] A_in,
 output reg [WIDTH-1:0] C_out = 0);

reg  [WIDTH-1:0] A_out, B_out;
always @( posedge clk )
begin
  A_out <= A_in;
  B_out <= A_out + 1;
  C_out <= B_out + 1;
end
endmodule
```



The order of non-blocking assignments does not matter!

Another style – multiple `always` blocks

```
reg  [WIDTH-1:0] A_out, B_out;

always @( posedge clk )
  A_out <= A_in;

always @( posedge clk )
  B_out <= A_out + 1;

always @( posedge clk )
  C_out <= B_out + 1;
```



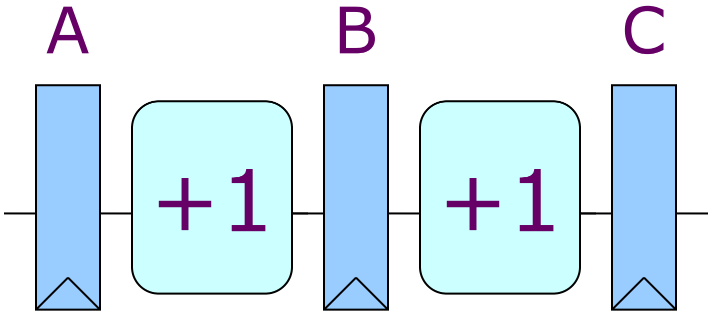
Does it have the same functionality?

Yes. But why?

Need to understand something about Verilog execution semantics

Yet another style – blocking assignments

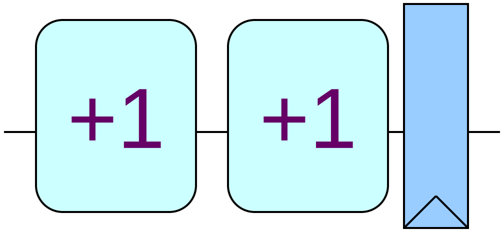
```
reg [WIDTH-1:0] A_out, B_out;
always @( posedge clk )
begin
  A_out = A_in;
  B_out = A_out + 1;
  C_out = B_out + 1;
end
```



Does it have the same functionality?

Not even close!

```
always @( posedge clk )
begin
  C_out = B_out + 1;
  B_out = A_out + 1;
  A_out = A_in;
end
```



Test module structure (test bench)

Once a design module is completed it must be tested

Functionality of the design module can be tested by applying stimulus and checking results.

```
Module <test module name>;
// Data type declaration
// Instantiate module (call the module that is going to be tested)
// Apply the stimulus
// Display results
endmodule
```

Test bench for the adder (Synopsys vcs)

```
module adder_test;
reg  [3:0] A, B;     // reg type for input
wire  [3:0] sum;   // wire type for output
wire  cout;
adder  adder_a(.cout(cout),.A(A),.B(B),.s(sum));
initial
 begin
   $vcdpluson(0);
   $monitor("%t A = %h B = %h, c_out=%b, sum = %h", $time, A, B,
cout, sum);
   A <= 4'b0000; B <= 4'b0000;
   #10  A <= 4'b1100; B <= 4'b0111;  #10  A <= 4'b0000; B <= 4'b0111;
   #10  A <= 4'b1100; B <= 4'b1011; #10  A <= 4'b0110; B <= 4'b0101;
   #10  A <= 4'b1110; B <= 4'b0011; #10
   $finish;
 end
//high-level descriptions
endmodule
```

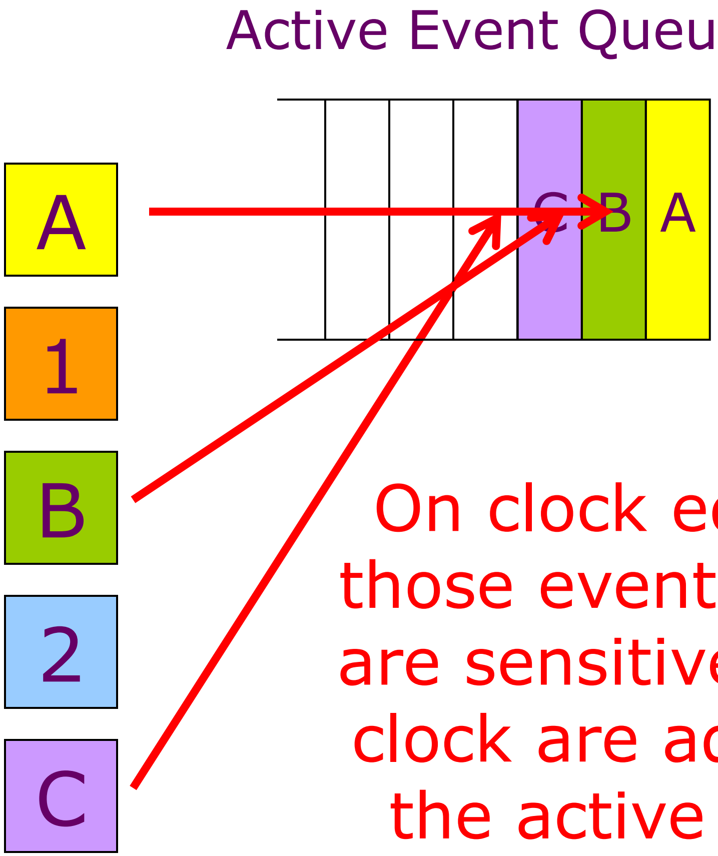
Verilog execution semantics

- Driven by simulation

- Explained using event queues

Execution semantics of Verilog - 1

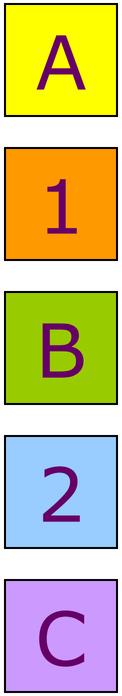
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```



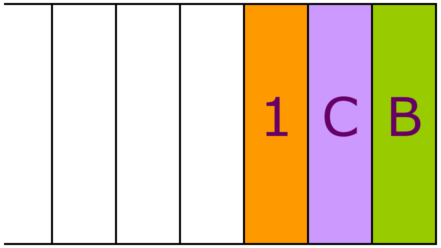
On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!

Execution semantics of Verilog - 2

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```



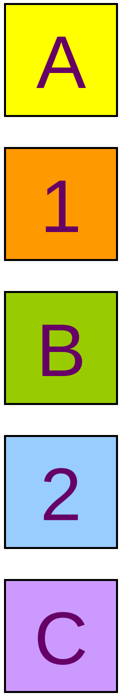
Active Event Queue



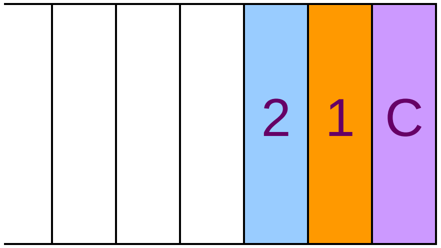
A evaluates and as a consequence 1 is added to the event queue

Execution semantics of Verilog -3

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```



Active Event Queue



Event queue is emptied before we go to next clock cycle

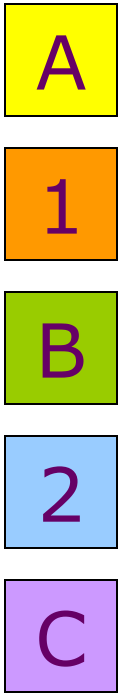
Non-blocking assignment

- Within a “clock cycle” all RHS variables are read first and all the LHS variables are updated together at the end of the clock cycle

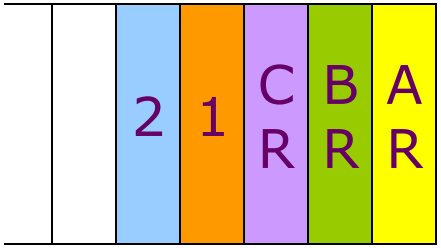
- Consequently, two event queues have to be maintained – one keeps the computations to be performed while the other keeps the variables to be updated

Non-blocking assignments require two event queues

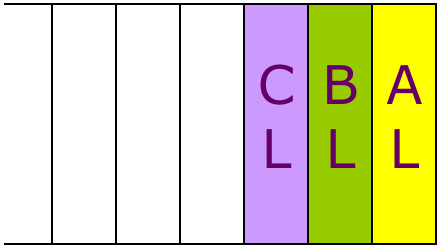
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```



Active Event Queue



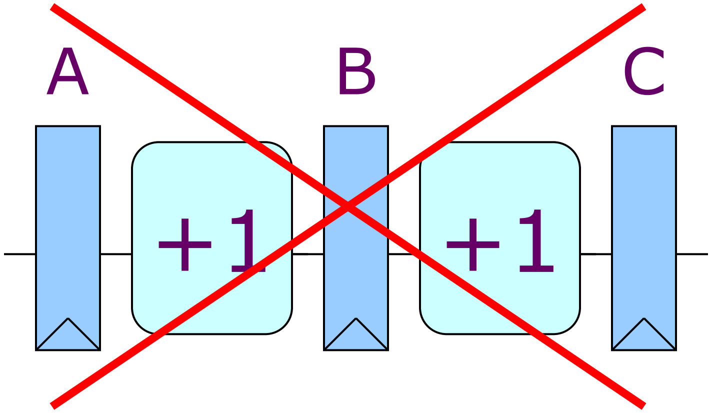
Non-Blocking Queue

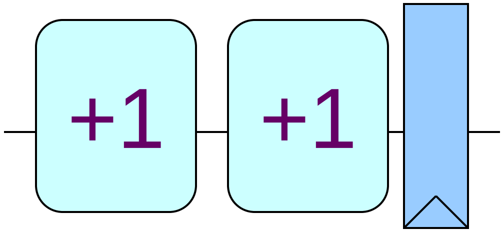


Variables in RHS of always blocks are not updated until all inputs (e.g. LHS + dependencies) are evaluated

Blocking assignments have a sequential language like semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
begin
  A_out = A_in;
  B_out = A_out + 1;
  C_out = B_out + 1;
end
```





Behavioral Verilog is richer

- Characterized by heavy use of sequential blocking statements in large always blocks
- Many constructs are not synthesizable but can be useful for **behavioral modeling** and **test benches**
  - Data dependent **for** and **while** loops
  - Additional behavioral datatypes: **`integer, real`**
  - Magic initialization blocks: **`initial`**
  - Magic delay statements: **`#<delay>`**
  - System calls: **`$display, $monitor, $finish`**

A simple summary

- Define a Module

```
module name( port list);
   // HDL modeling of functionality
   assign …. // data flow assignment
   always @ (posedge clk) … // sequential
   always @ (*) … // combinational
   instantiation of lower level modules …
   task and function
endmodule
```

Take away points

- Follow the simple guidelines to write synthesizable Verilog
- Parameterized models provide the foundation for reusable libraries of components
- Use explicit state to prevent unwanted state inference and to more directly represent the desired hardware
- Begin your RTL design by identifying the external interface and then move on to partition your design into the memories, datapaths, and control logic