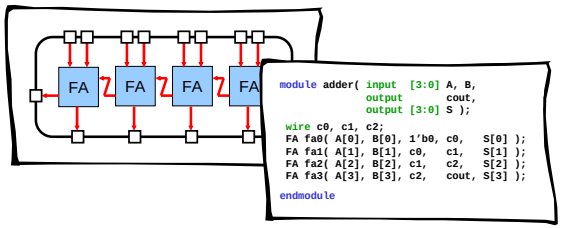
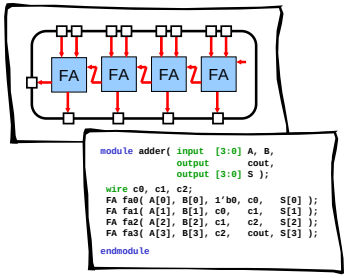
Verilog 1 - Fundamentals



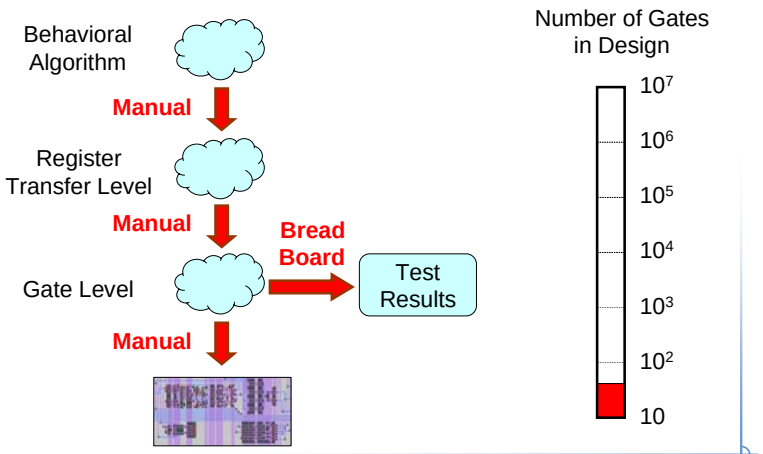
6.375 Complex Digital Systems
Krste Asanovic
February 9, 2007

Verilog Fundamentals

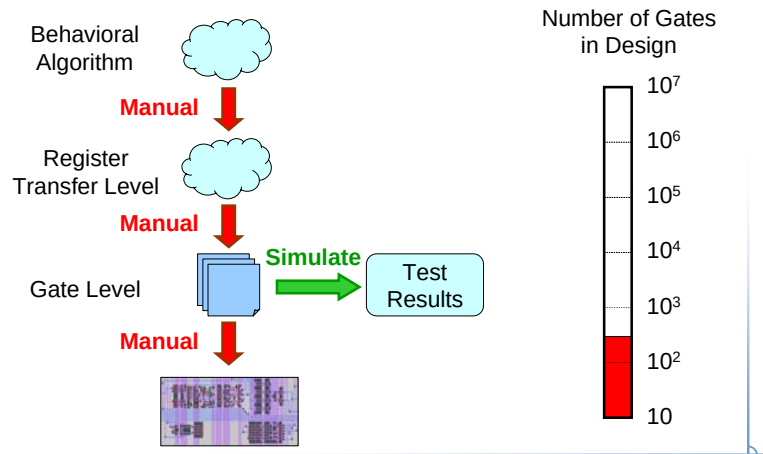
- History of hardware design languages
- Data types
- Structural Verilog
- Functional Verilog
  - Gate level
  - Register transfer level
  - High-level behavioral



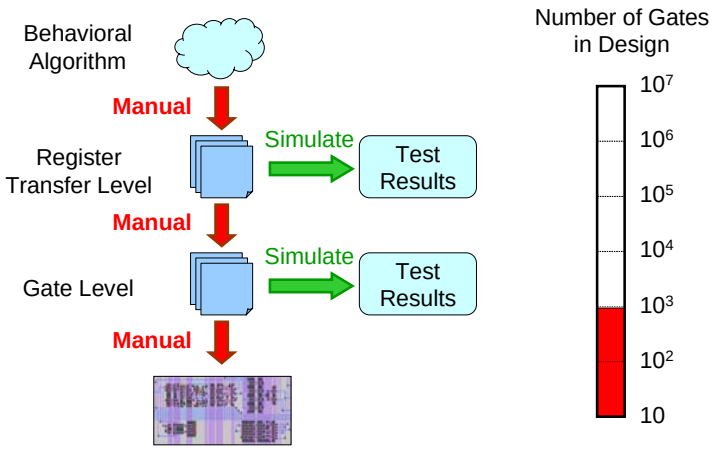
Originally designers used manual translation + bread boards for verification



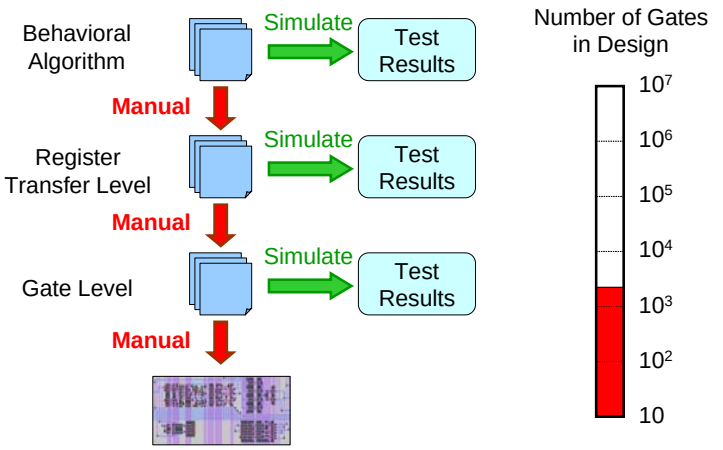
Hardware design languages enabled logic level simulation and verification



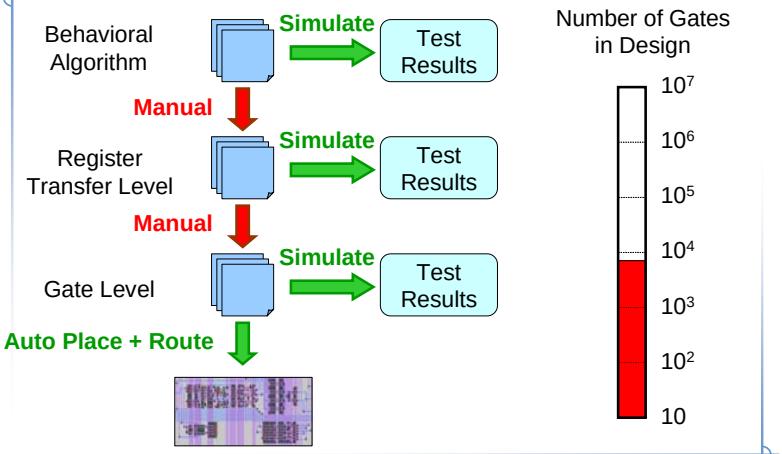
Designers began to use HDLs for higher level verification and design exploration



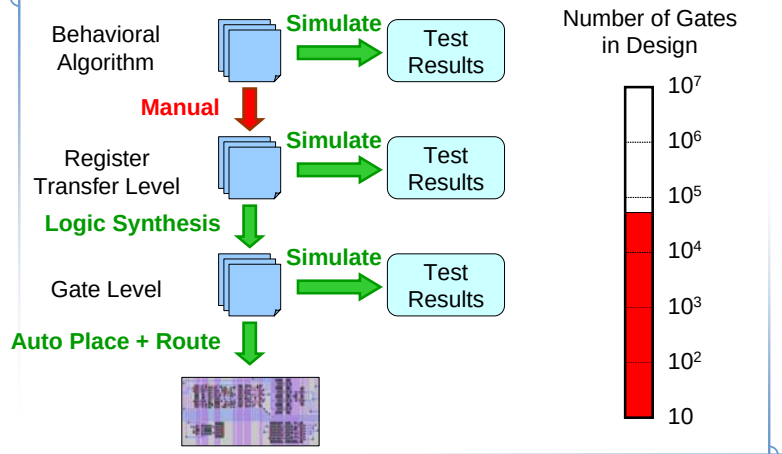
HDL behavioral models act as a precise and executable specification



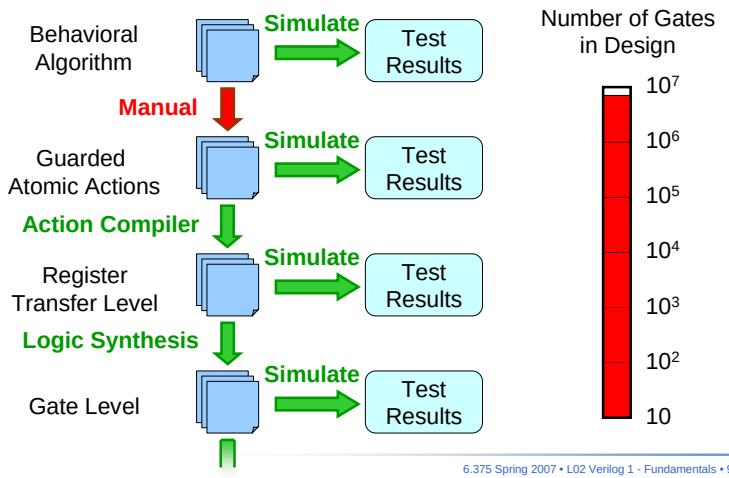
Once designs were written in HDLs tools could be used for automatic translation



Once designs were written in HDLs tools could be used for automatic translation



Guarded atomic actions can help us to efficiently raise the abstraction level

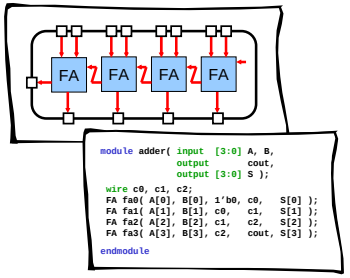


Various hardware design languages are available

Verilog-1995	C-like concise syntax; only bit vector data types; very low level
Verilog-2001	Several small changes to avoid common mistakes; static elaboration
VHDL	ADA-like verbose syntax; extensible types; DoD mandate; separates interface from implementation; operator overloading
SystemVerilog	Adds strong type checking; separates interface from implementation
Bluespec	Uses guarded atomic actions; advanced static elaboration and type system; separates interface from implementation

Verilog Fundamentals

- History of hardware design languages
- **Data types**
- Structural Verilog
- Functional Verilog
  - Gate level
  - Register transfer level
  - High-level behavioral



Primary Verilog data type is a bit-vector where bits can take on one of four values

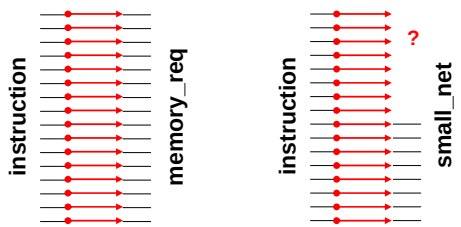
Value	Meaning
0	Logic zero
1	Logic one
X	Unknown logic value
Z	High impedance, floating

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.

The Verilog keyword wire is used to denote a standard hardware net

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

Absolutely no type safety when connecting nets!



Verilog includes ways to specify bit literals in various bases

```
4'b10_11
```

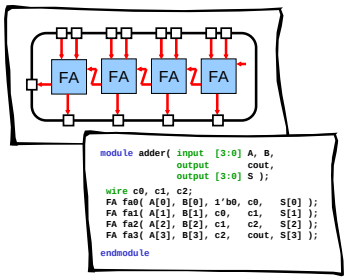
- Decimal number representing size in bits
- Base format (d,b,o,h)
- Underscores are ignored

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

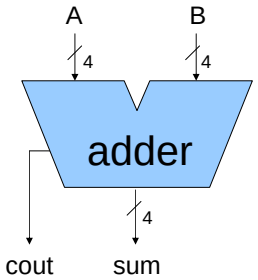
We'll learn how to actually assign literals to nets a little later

Verilog Basics

- History of hardware design languages
- Data types
- **Structural Verilog**
- Functional Verilog
  - Gate level
  - Register transfer level
  - High-level behavioral



A Verilog module includes a module name and a port list

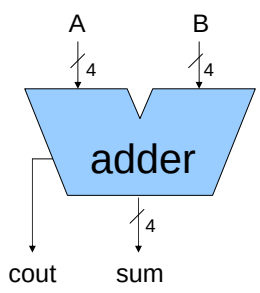



```
module adder( A, B, cout, sum );
  input  [3:0] A;
  input  [3:0] B;
  output       cout;
  output [3:0] sum;

  // 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!

A Verilog module includes a module name and a port list



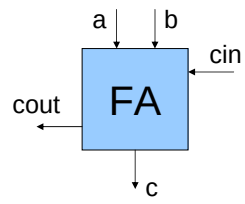
Traditional Verilog-1995 Syntax

```
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

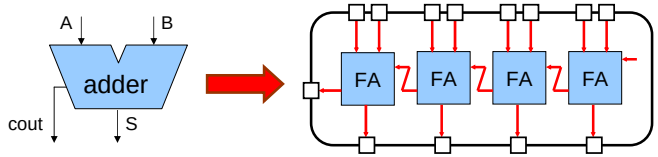
```
module adder( input  [3:0] A,
              input  [3:0] B,
              output       cout,
              output [3:0] sum );
```


A module can instantiate other modules creating a module hierarchy



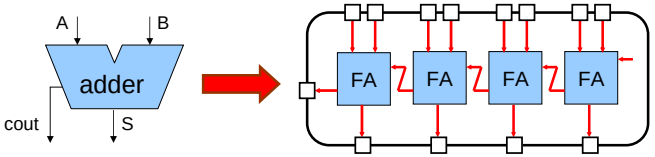

```
module FA( input  a, b, cin
           output cout, sum );

  // HDL modeling of 1 bit
  // full adder functionality

endmodule
```

A module can instantiate other modules creating a module hierarchy




```
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
```


A module can instantiate other modules creating a module hierarchy




```
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

Verilog supports connecting ports by position and by name

Connecting ports by ordered list

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

Connecting ports by name (compact)

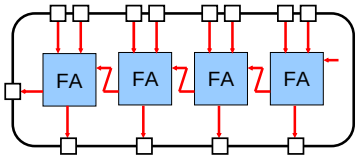
```
FA fa0( .a(A[0]), .b(B[0]),
        .cin(1'b0), .cout(c0), .sum(S[0]) );
```

Connecting ports by name

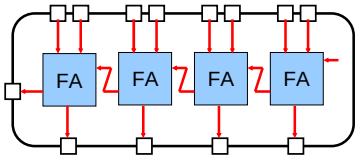
```
FA fa0
(
  .a    (A[0]),
  .b    (B[0]),
  .cin  (1'b0),
  .cout (c0),
  .sum  (S[0])
);
```

For all but the smallest modules, connecting ports by name yields clearer and less buggy code.

Let's review how to turn our schematic diagram into structural Verilog



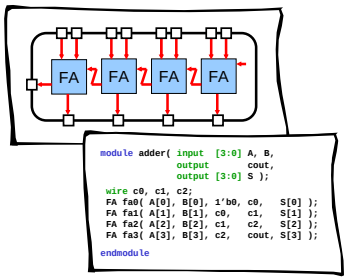
Let's review how to turn our schematic diagram into structural Verilog




```
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
```


Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- **Functional Verilog**
  - **Gate level**
  - **Register transfer level**
  - **High-level behavioral**



Functional Verilog can roughly be divided into three abstraction levels

Behavioral Algorithm — Abstract algorithmic description

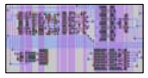
Manual

Register Transfer Level — Describes how data flows between state elements for each cycle

Logic Synthesis

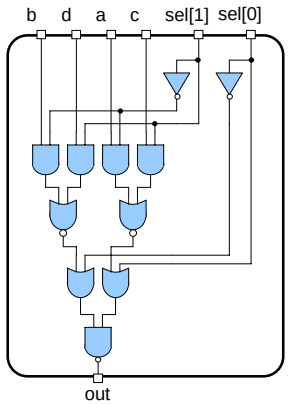
Gate Level — Low-level netlist of primitive gates

Auto Place + Route



Gate-level Verilog uses structural Verilog to connect primitive gates

```
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
```



Continuous assignment statements assign one net to another or to a literal

Explicit continuous assignment

```
wire [15:0] netA;
wire [15:0] netB;

assign netA = 16'h3333;
assign netB = netA;
```

Implicit continuous assignment

```
wire [15:0] netA = 16'h3333;
wire [15:0] netB = netA;
```


Using continuous assignments to implement an RTL four input multiplexer

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

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

endmodule
```

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

Verilog RTL includes many operators in addition to basic boolean logic

```
// 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

// Simple four bit adder
module adder( input  [3:0] op1, op2,
              output [3:0] sum );

  assign sum = op1 + op2;

endmodule
```

If input is undefined we want to propagate that information.

Verilog RTL operators

Arithmetic	`+ - * / % **`	Reduction	`& ~& | ~| ^ ^~`
Logical	`! && ||`	Shift	`>> << >>> <<<`
Relational	`> < >= <=`	Concatenation	`{ }`
Equality	`== != === !===`	Conditional	`?:`
Bitwise	`~ & | ^ ^~`		

```
wire [ 3:0] net1 = 4'b00xx;
wire [ 3:0] net2 = 4'b1110;
wire [11:0] net3 = { 4'b0, net1, net2 };

wire equal = ( net3 === 12'b0000_1110_00xx );
```

Avoid (`/ % **`) since they usually synthesize poorly

Always blocks have parallel inter-block and sequential intra-block semantics

```
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
```

The always block is reevaluated whenever a signal in its sensitivity list changes

Always blocks have parallel inter-block and sequential intra-block semantics

```
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
```

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

Always blocks have parallel inter-block and sequential intra-block semantics

```
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
```

LHS of procedural assignments must be declared as a reg type. Verilog reg is not necessarily a hardware register!

Always blocks have parallel inter-block and sequential intra-block semantics

```
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
```

What happens if we accidentally forget a signal on the sensitivity list?

Always blocks have parallel inter-block and sequential intra-block semantics

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

  reg out, t0, t1;

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

endmodule
```

Verilog-2001 provides special syntax to automatically create a sensitivity list for all signals read in the always block

Continuous and procedural assignment statements are very different

Continuous assignments are for naming and thus we cannot have multiple assignments for the same wire

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

Procedural assignments hold a value semantically, but it is important to distinguish this from hardware state

```
reg out, t0, t1, temp;
always @( * )
begin
  temp = ~( (sel[1] & c) | (~sel[1] & a) );
  t0   = temp;
  temp = ~( (sel[1] & d) | (~sel[1] & b) );
  t1   = temp;
  out  = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
```

Always blocks can contain more advanced control constructs

```
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
```


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

So is this how we make latches and flip-flops?

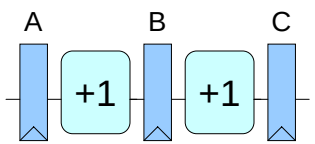
```
module latch
(
  input  clk,
  input  d,
  output reg q
);

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

endmodule
```

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

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

endmodule
```

Edge-triggered always block

To understand why we need to know more about Verilog execution semantics

```
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;
```



To understand why we need to know more about Verilog execution semantics

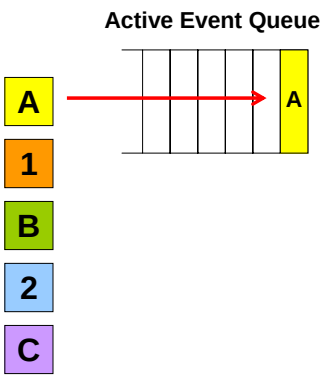
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

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

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

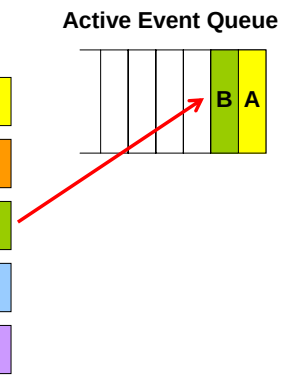
To understand why we need to know more about Verilog execution semantics

```
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;
```



To understand why we need to know more about Verilog execution semantics

```
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;
```

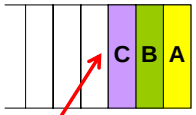


To understand why we need to know more about Verilog execution semantics

```
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;
```

A 1 B 2 C

Active Event Queue

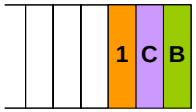


To understand why we need to know more about Verilog execution semantics

```
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;
```

A 1 B 2 C

Active Event Queue



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

To understand why we need to know more about Verilog execution semantics

```
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;
```

A

1

B

2

C

Active Event Queue

B evaluates, using *new* value of A_out, and as a consequence 2 is added to the event queue

Race Condition!

To understand why we need to know more about Verilog execution semantics

```
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;
```

A

1

B

2

C

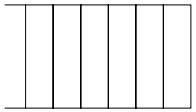
Active Event Queue

Event queue is emptied before we go to next clock cycle

To understand why we need to know more about Verilog execution semantics

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

always @( posedge clk )
  A_out = A_in;                 A

assign B_in = A_out + 1;        1

always @( posedge clk )
  B_out = B_in;                 B

assign C_in = B_out + 1;        2

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

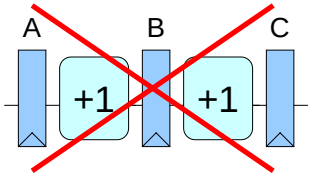
Active Event Queue

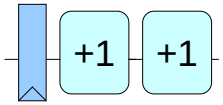


Event queue is emptied before we go to next clock cycle

We didn't model what we expected due to Verilog execution semantics

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

always @( posedge clk )
  A_out = A_in;                 A

assign B_in = A_out + 1;        1

always @( posedge clk )
  B_out = B_in;                 B

assign C_in = B_out + 1;        2

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





Non-blocking procedural assignments add an extra event queue

```
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;
```

A

1

B

2

C

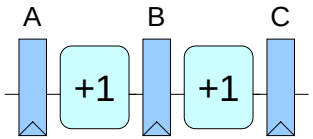
Active Event Queue

Non-Blocking Queue

Change to left-hand side deferred in non-blocking queue, takes effect after all active events scheduled

Non-blocking procedural assignments add an extra event queue

```
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;
```



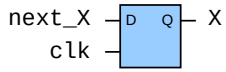
The order of non-blocking assignments does not matter!

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

always @( posedge clk )
begin
  A_out <= A_in;
  B_out <= B_in;
  C_out <= C_in;
end

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

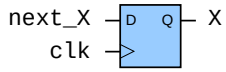
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
begin
  C_out <= C_in;
  B_out <= B_in;
  A_out <= A_in;
end

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


Common patterns for latch and flip-flop inference

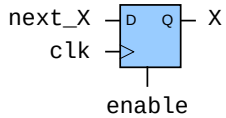
```
always @( clk or D)
begin
  if ( clk )
    Q <= D;
end
```




```
always @( posedge clk )
begin
  Q <= D;
end
```




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



Writing Good Synthesizable Verilog

- Only leaf modules should have functionality
  - All other modules are strictly structural, i.e., they only wire together sub-modules
- Use only positive-edge triggered flip-flops for state
- Do not assign to the same variable from more than one always block
- Separate combinational logic from sequential logic
  - Combinational logic described using `always @(*)` and blocking `=` assignments and `assign` statements
  - Sequential logic described with `always @(posedge clk)` and non-blocking `<=` assignments

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

Combinational logic

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

Update of state only (don't be tempted to add combinational logic into clocked always block)

Behavioral Verilog is used to model the abstract function of a hardware module

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

Verilog can be used to model the high-level behavior of a hardware block

```
module factorial( input [ 7:0] in, output reg [15:0] out );

  integer num_calls;
  initial num_calls = 0;

  integer multiplier;
  integer result;
  always @(*)
  begin

    multiplier = in;
    result = 1;
    while ( multiplier > 0 )
    begin
      result = result * multiplier;
      multiplier = multiplier - 1;
    end

    out = result;
    num_calls = num_calls + 1;
  end

endmodule
```

Initial statement

Variables of type integer

Data dependent while loop

Delay statements should only be used in test harnesses

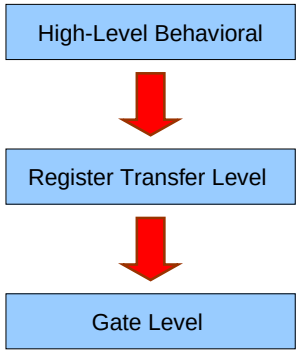
```
module mux4
(
  input       a,
  input       b,
  input       c,
  input       d,
  input [1:0] sel,
  output      out
);

  wire #10 t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  wire #10 t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  wire #10 out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

Although this will add a delay for simulation, these are ignored in synthesis

System tasks are used for test harnesses and simulation management

```
reg [ 1023:0 ] exe_filename;

initial
begin

  // This turns on VCD (plus) output
  $vcdpluson(0);

  // This gets the program to load into memory from the command line
  if ( $value$plusargs( "exe=%s", exe_filename ) )
    $readmemh( exe_filename, mem.m );
  else
  begin
    $display( "ERROR: No executable specified! (use +exe=<filename>)" );
    $finish;
  end

  // Stobe reset
  #0  reset = 1;
  #38 reset = 0;

end
```


Which abstraction is the right one?



Designers usually use a **mix of all three**! Early on in the design process they might use mostly behavioral models. As the design is refined, the behavioral models begin to be replaced by dataflow models. Finally, the designers use automatic tools to synthesize a low-level gate-level model.

A common approach is to use C/C++ for initial behavioral modeling, and for building test rigs.

Take away points

- Structural Verilog enables us to describe a hardware schematic textually
- Verilog can model hardware at three levels of abstraction: **gate level**, **register transfer level**, and **behavioral**
- Understanding the Verilog execution semantics is critical for understanding blocking + non-blocking assignments
- Designers must have the hardware they are trying to create in mind when they write their Verilog

Next Lecture: We will use a simple SMIPS processor to illustrate many of concepts introduced today.