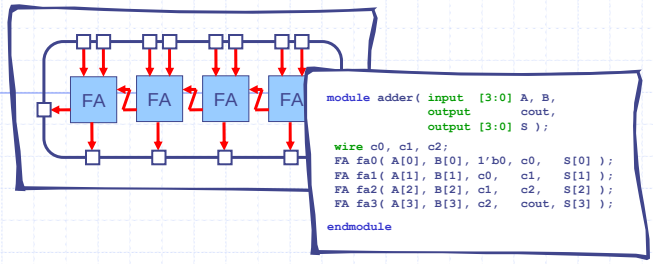
Verilog 1 - Fundamentals



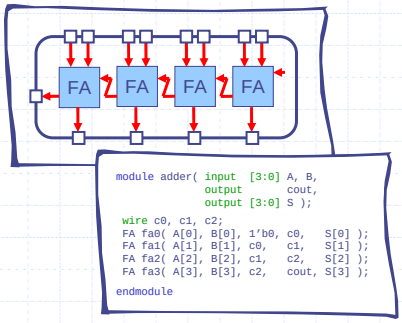
6.375 Complex Digital Systems

Arvind

February 6, 2009

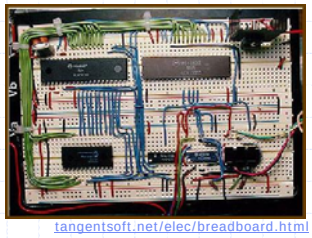
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors



Originally designers used breadboards for prototyping

Solderless Breadboard



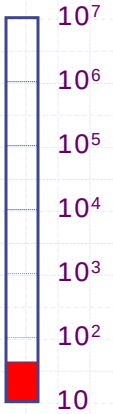
tangentsoft.net/elec/breadboard.html

No symbolic execution or testing

Printed circuit board

home.cogeco.ca

Number of Gates in Design



HDLs enabled logic level simulation and testing

Gate Level Description

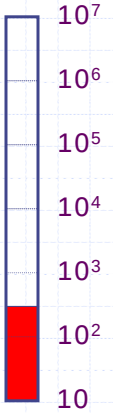
Simulate

Test Results

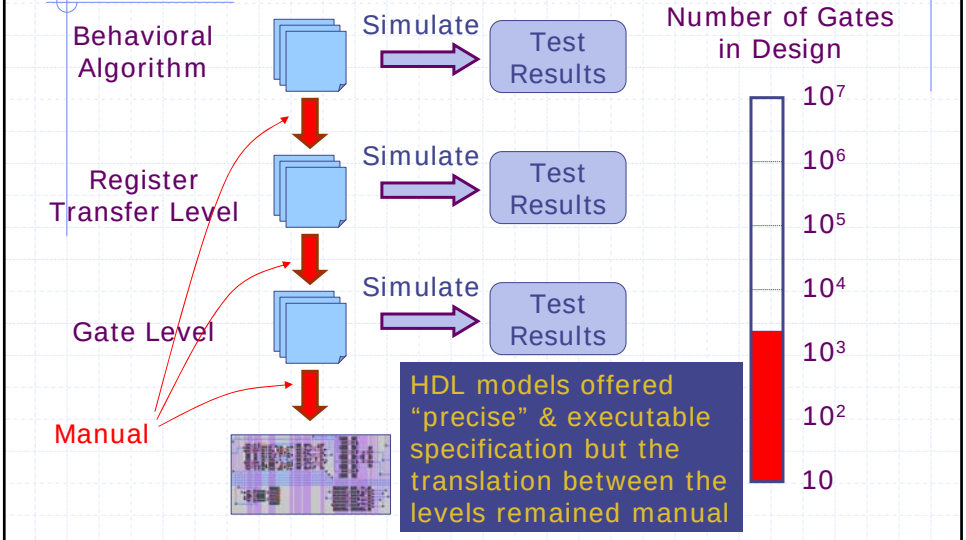
Manual

HDL = Hardware Description Language

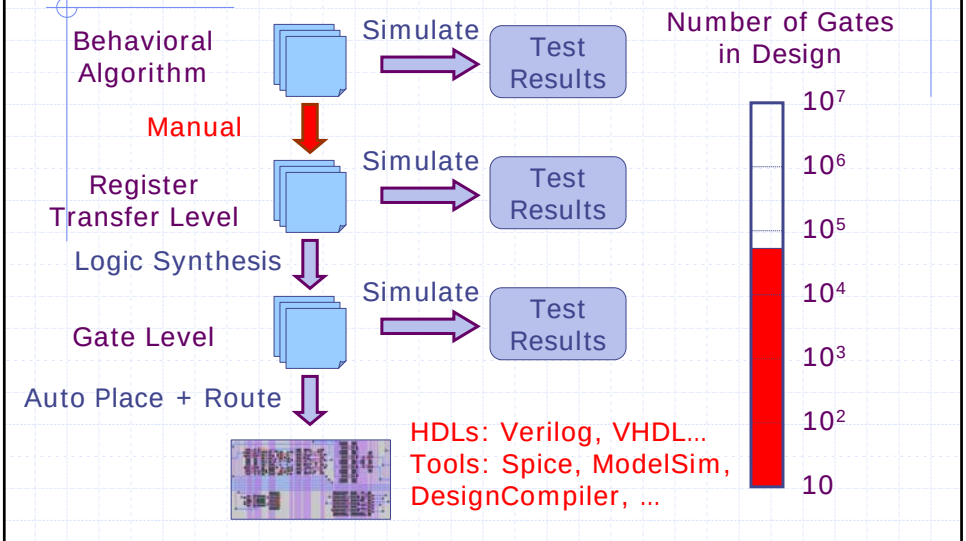
Number of Gates in Design



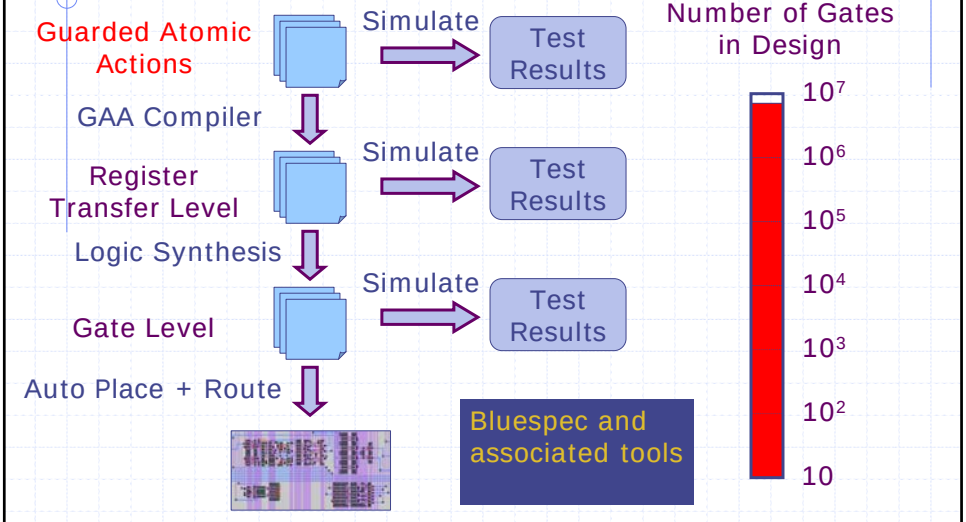
Designers began to use HDLs for higher level design



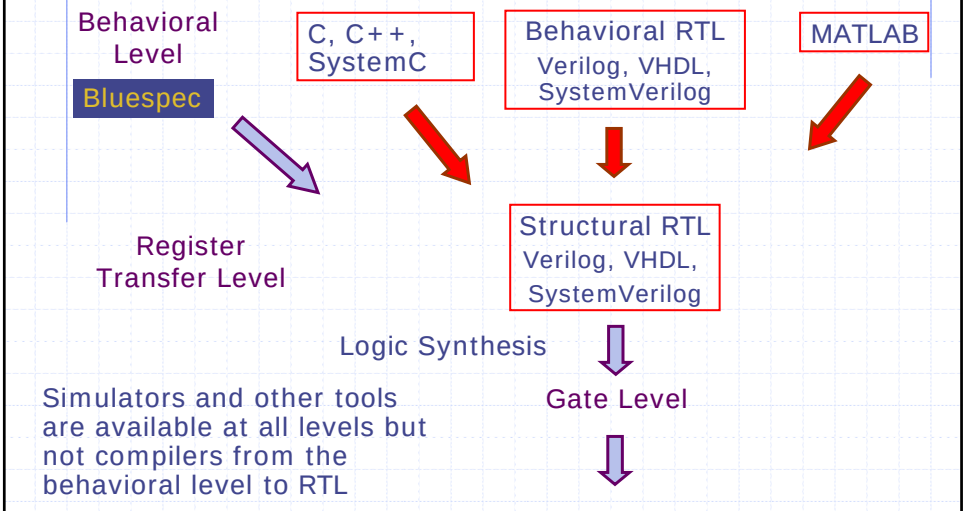
HDLs led to tools for automatic translation



Raising the abstraction further …



The current situation



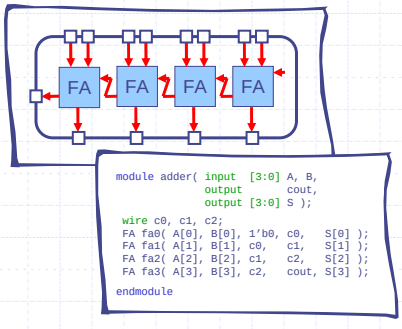
Common misconceptions

- The only behavioural languages are C, C++
- RTL languages are necessarily lower-level than behavioral languages
  - Yes- in the sense that C or SystemC is farther away from hardware
  - No- in the sense that HDLs can incorporate the most advanced language ideas.

Bluespec is a modern high-level language and at the same time can describe hardware to the same level of precision as RTL

Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors




```
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
```

Bit-vector is the only data type in Verilog

A bit can take on one of four values

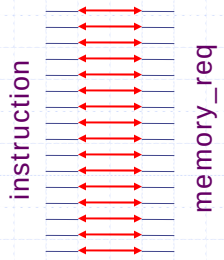
Value	Meaning
0	Logic zero
1	Logic one
X	Unknown logic value
Z	High impedance, floating

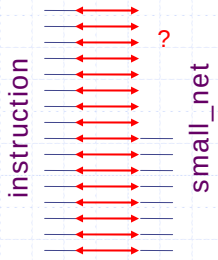
An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don't care what the value is. This can help catch bugs and improve synthesis quality.

"wire" is used to denote a hardware net

```
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
```

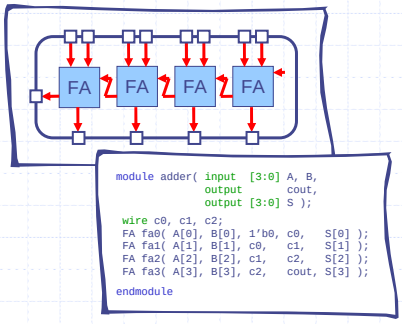
- Underscores are ignored
- Base format (d,b,o,h)
- Decimal number representing size in bits

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`

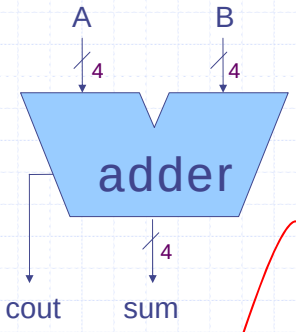
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors




```
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
```

A Verilog module has a name and a port list

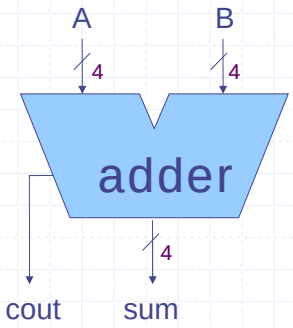



```
module adder( A, B, cout, sum );
  input  [3:0] A;
  input  [3:0] B;
  output       cout;
  output [3:0] sum;

  // 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



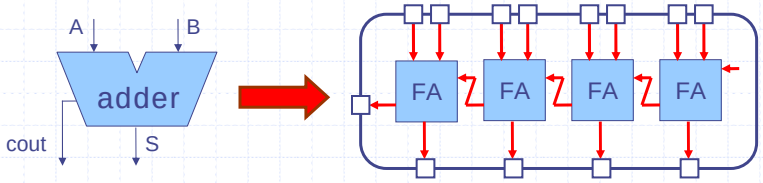
Traditional Verilog-1995 Syntax

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

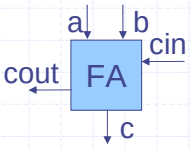
ANSI C Style Verilog-2001 Syntax

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

A module can instantiate other modules

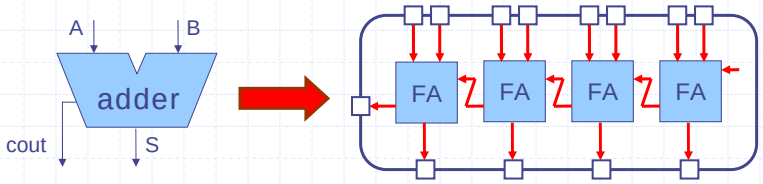



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

  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
```


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]) );
```

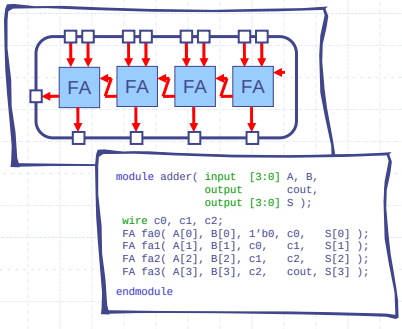
Argument order does not matter when ports are connected by name

```
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.

Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors

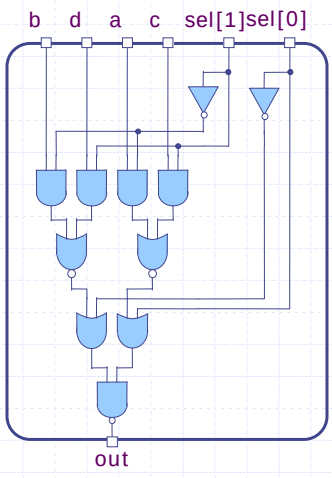


A module's behavior can be described in many different ways but it should not matter from outside

Example: mux4

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

```
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
```

Language defined operators

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

```
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
```

default value

Parameterization is a good practice for reusable modules

Writing a mux*n* is challenging

Instantiation Syntax

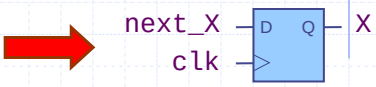
```
mux4#(32) alu_mux
( .a (op1),
  .b (op2),
  .c (op3),
  .d (op4),
  .sel (alu_mux_sel),
  .out (alu_mux_out) );
```


Verilog Registers "reg"

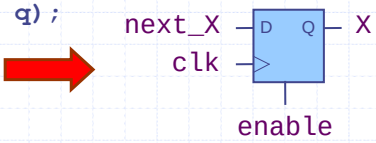
- Wires are line names – they do not represent storage and can be assigned only once
- Regs are imperative variables (as in C):
  - "nonblocking" assignment r <= v
  - can be assigned multiple times and holds values between assignments

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
```



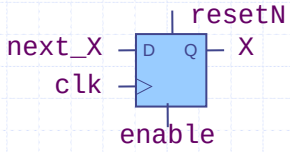
flip-flops with reset

```
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

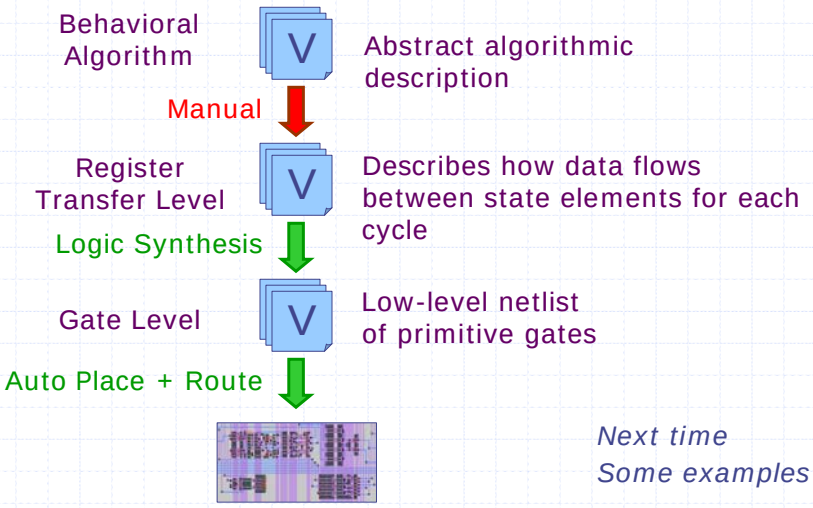
Static Elaboration: Generate

```
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



Next time
Some examples