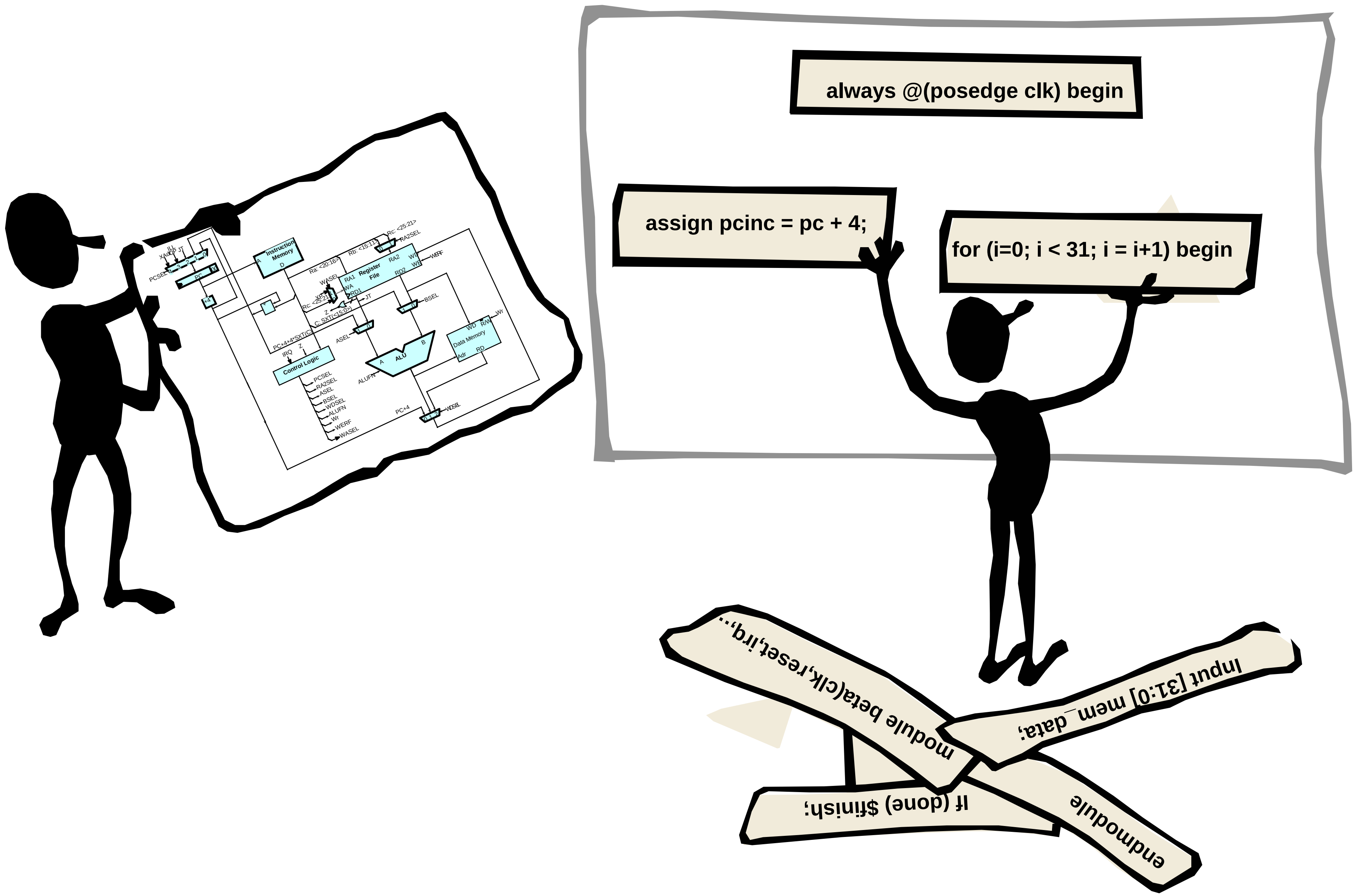
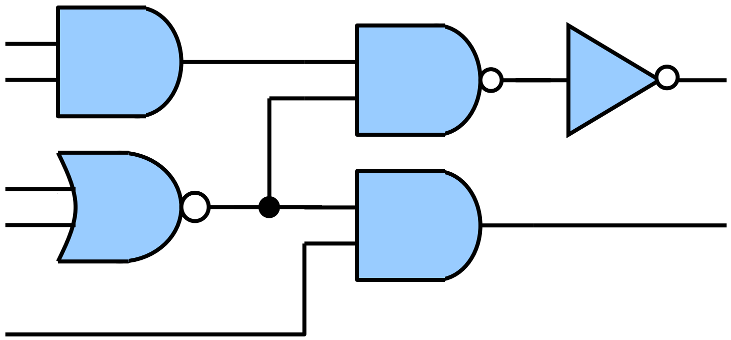
Digital Design Using Verilog

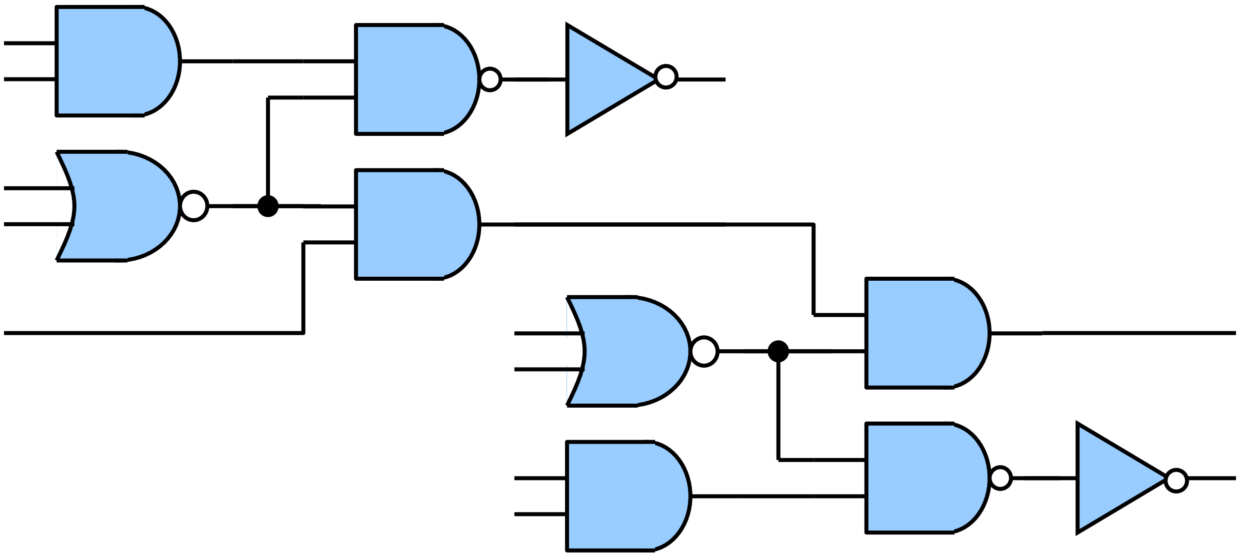


Hardware Description Languages



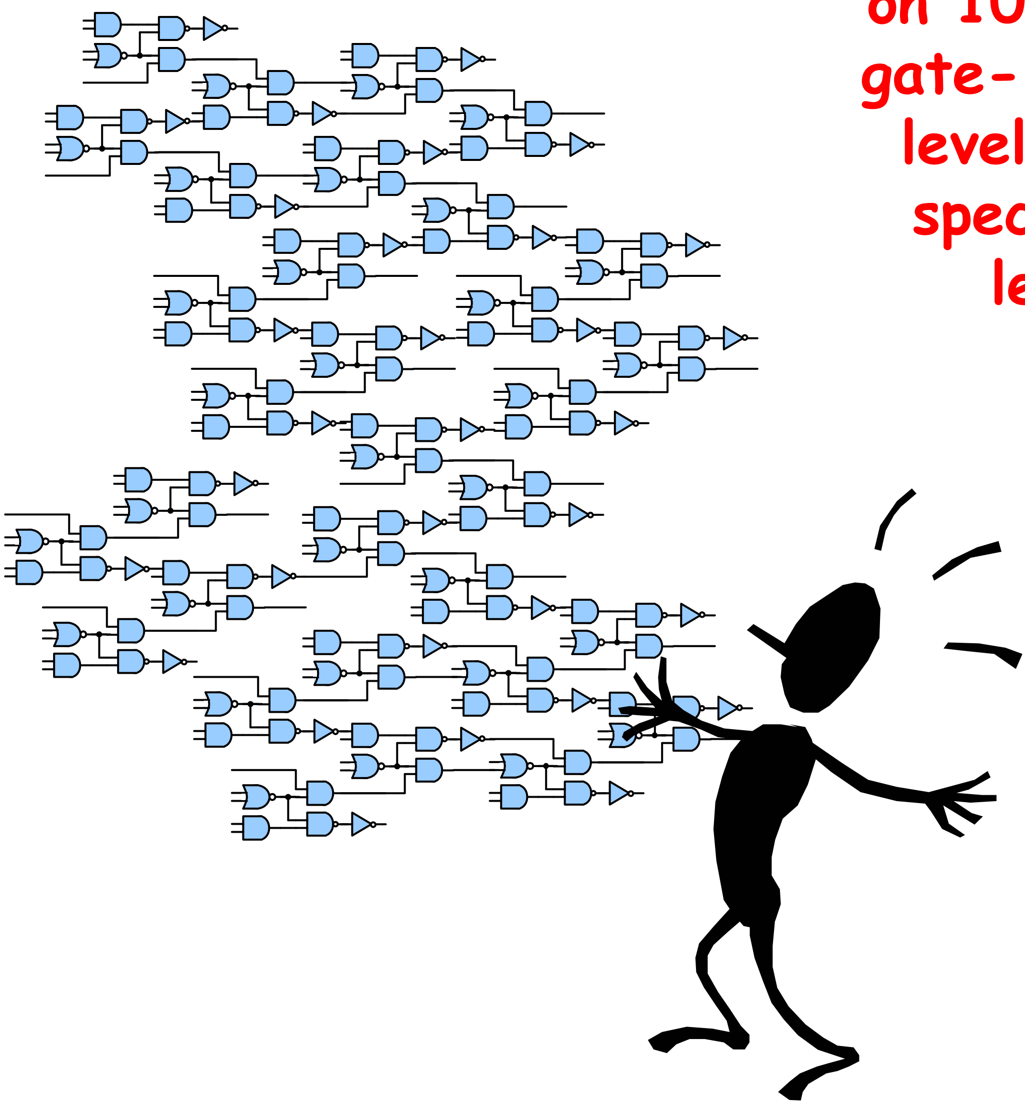
In the beginning designs involved just a few gates, and thus it was possible to verify these circuits on paper or with breadboards

Hardware Description Languages



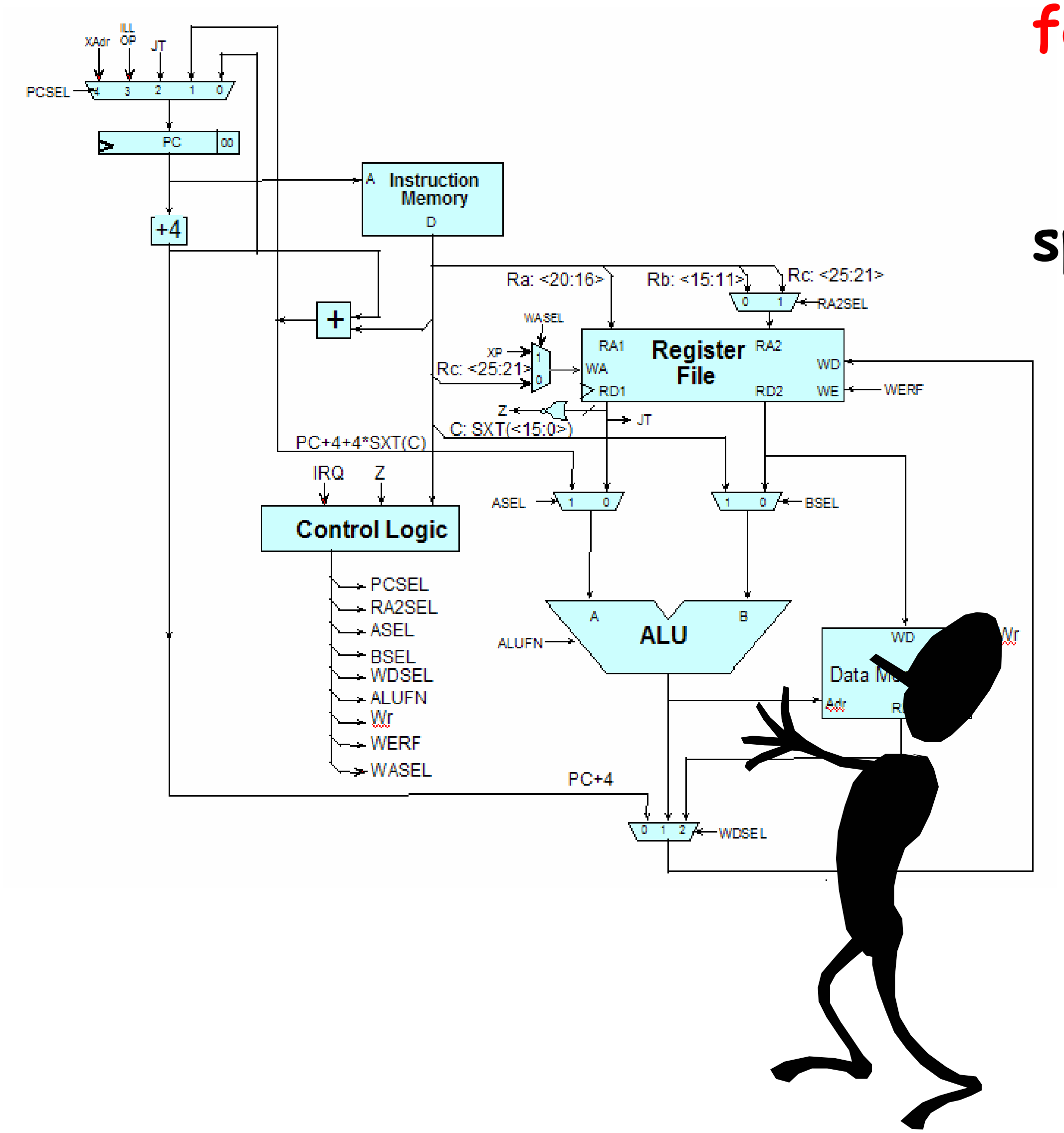
As designs grew larger and more complex, designers began using gate-level models described in a Hardware Description Language to help with verification before fabrication

Hardware Description Languages



When designers began working on 100,000 gate designs, these gate-level models were too low-level for the initial functional specification and early high-level design exploration

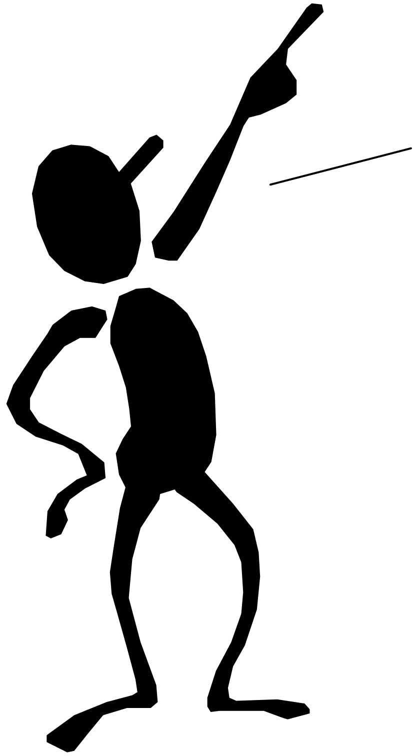
Hardware Description Languages



Designers again turned to HDLs for help – abstract behavioral models written in an HDL provided both a precise specification and a framework for design exploration

Advantages of HDLs

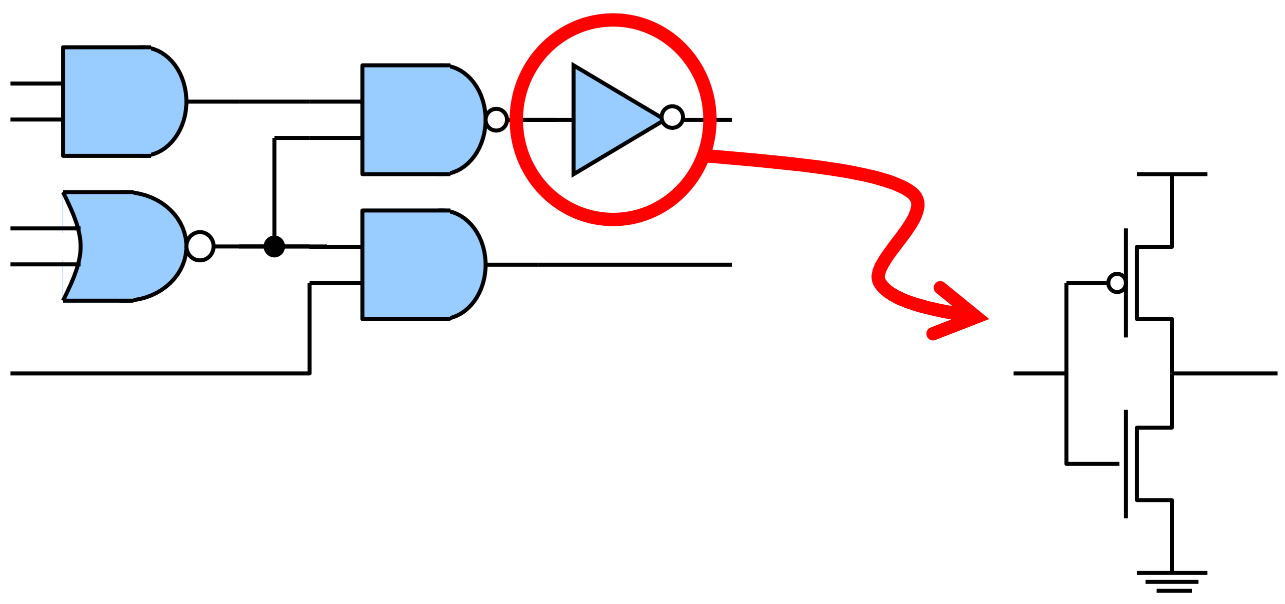
Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate behavior from implementation at various levels of abstraction



HDLs do this with modules and interfaces

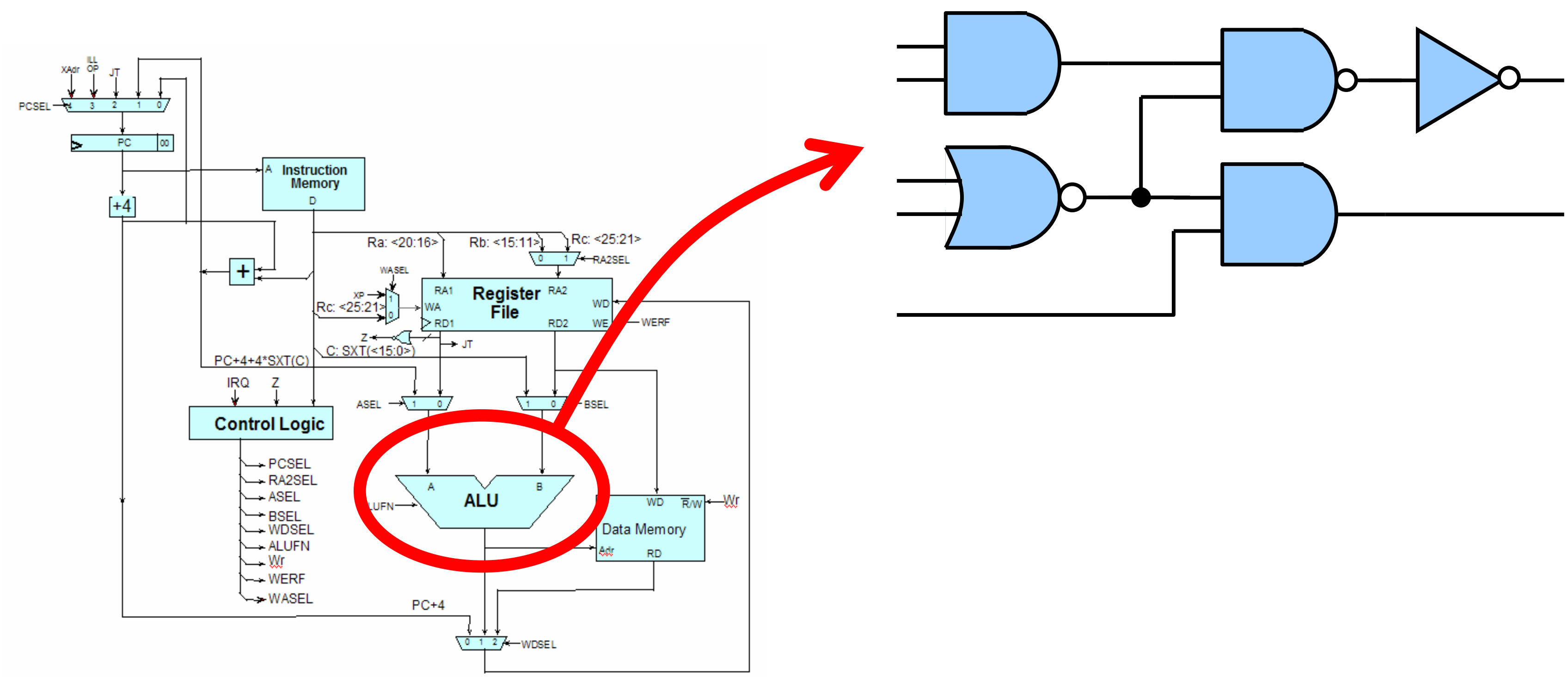
Advantages of HDLs

Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate **behavior from implementation** at various levels of abstraction



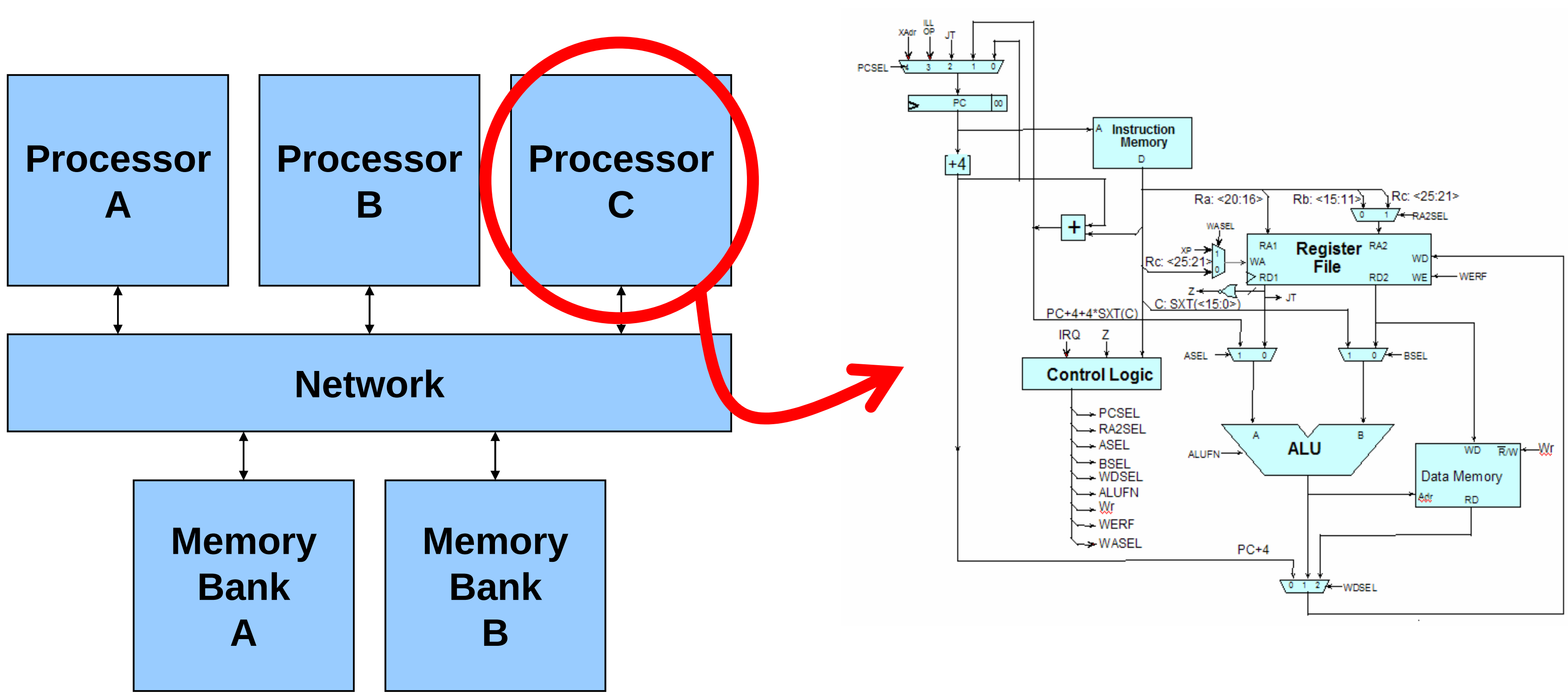
Advantages of HDLs

Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate **behavior from implementation** at various levels of abstraction



Advantages of HDLs

Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate **behavior from implementation** at various levels of abstraction



Advantages of HDLs

Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate behavior from implementation at various levels of abstraction

- Designers can develop an executable functional specification that documents the exact behavior of all the components and their interfaces
- Designers can make decisions about cost, performance, power, and area earlier in the design process
- Designers can create tools which automatically manipulate the design for verification, synthesis, optimization, etc.

A Tale of Two HDLs

VHDL	Verilog
ADA-like verbose syntax, lots of redundancy	C-like concise syntax
Extensible types and simulation engine	Built-in types and logic representations
Design is composed of entities each of which can have multiple architectures	Design is composed of modules which have just one implementation
Gate-level, dataflow, and behavioral modeling. Synthesizable subset.	Gate-level, dataflow, and behavioral modeling. Synthesizable subset.
Harder to learn and use, DoD mandate	Easy to learn and use, fast simulation

We will use Verilog …

Advantages

- Choice of many US design teams
- Most of us are familiar with C-like syntax
- Simple module/port syntax is familiar way to organize hierarchical building blocks and manage complexity
- With care it is well-suited for both verification and synthesis

Disadvantages

- Some comma gotchas which catch beginners everytime
- C syntax can cause beginners to assume C semantics
- Easy to create very ugly code, good and consistent coding style is essential

An HDL is NOT a Software Programming Language

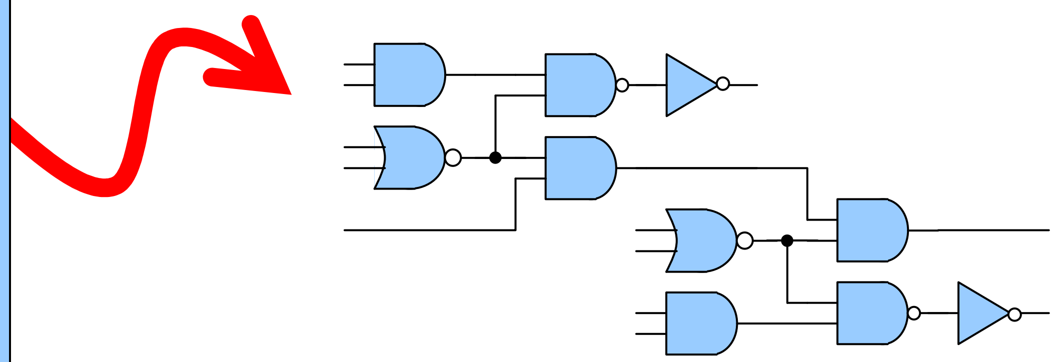
Software Programming Language

- Language which can be translated into machine instructions and then executed on a computer

Hardware Description Language

- Language with syntactic and semantic support for modeling the temporal behavior and spatial structure of hardware

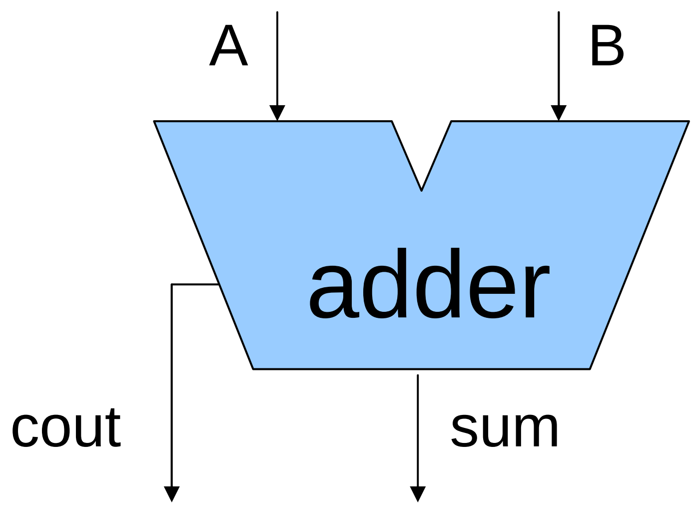
```
module foo(clk,xi,yi,done);
  input [15:0] xi,yi;
  output done;

  always @(posedge clk)
    begin:
      if (!done) begin
        if (x == y) cd <= x;
        else (x > y) x <= x - y;
      end
    end
endmodule
```



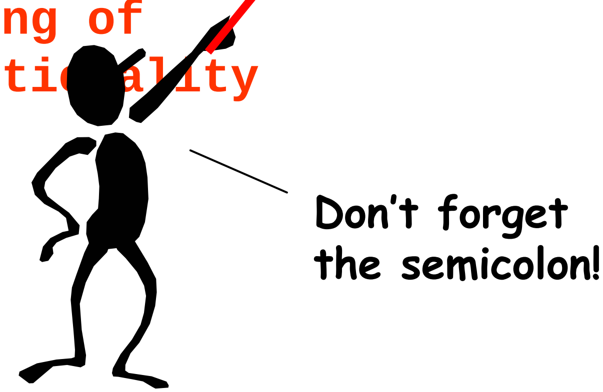
Hierarchical Modeling with Verilog

A Verilog module includes a module name and an interface in the form of a port list

- Must specify direction and bitwidth for each port



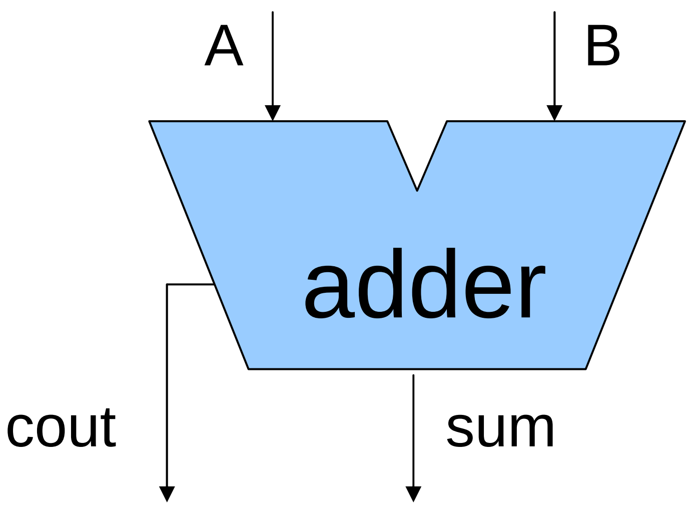

```
module adder( A, B, cout, sum );
  input  [3:0] A, B;
  output       cout;
  output [3:0] sum;

  // HDL modeling of
  // adder functionality

endmodule
```



Hierarchical Modeling with Verilog

A Verilog module includes a module name and an interface in the form of a port list

- Must specify direction and bitwidth for each port
- Verilog-2001 introduced a succinct ANSI C style portlist

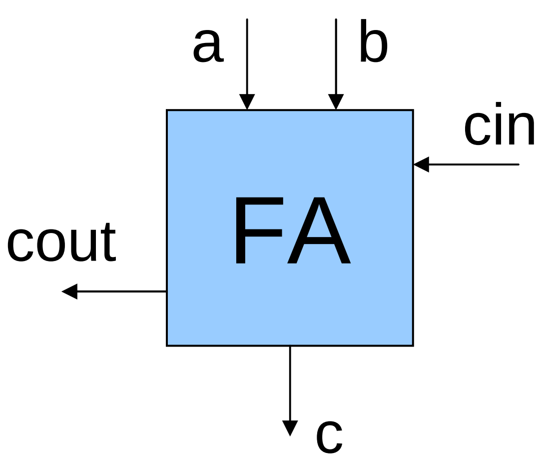



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] sum );

  // HDL modeling of 4 bit
  // adder functionality

endmodule
```

Hierarchical Modeling with Verilog

A module can contain other modules through module instantiation creating a module hierarchy

- Modules are connected together with nets
- Ports are attached to nets either by position or by name

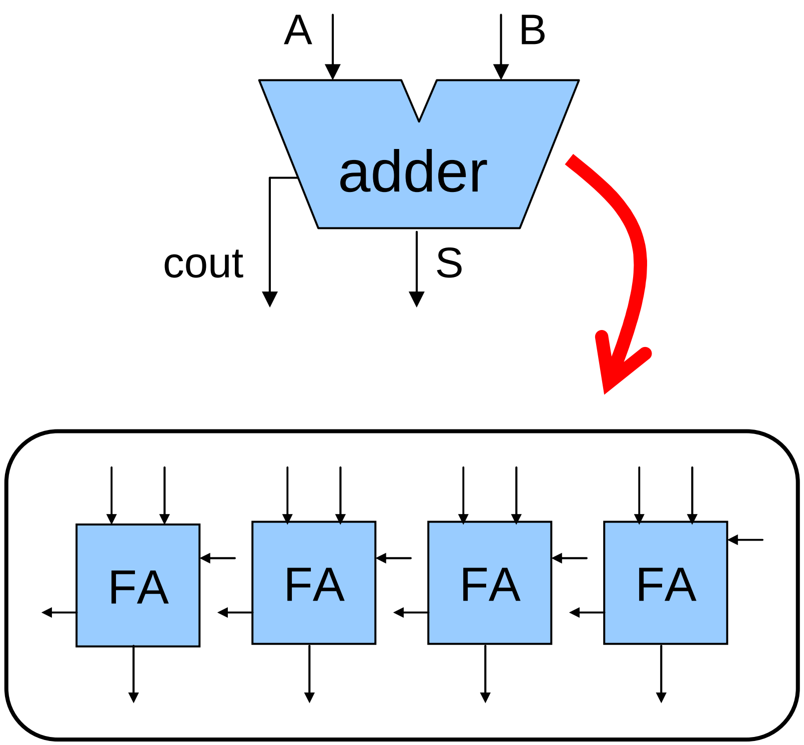



```
module FA( input  a, b, cin
           output cout, sum );

  // HDL modeling of 1 bit
  // adder functionality

endmodule
```

Hierarchical Modeling with Verilog

A module can contain other modules through module instantiation creating a module hierarchy

- Modules are connected together with nets
- Ports are attached to nets either by position or by name

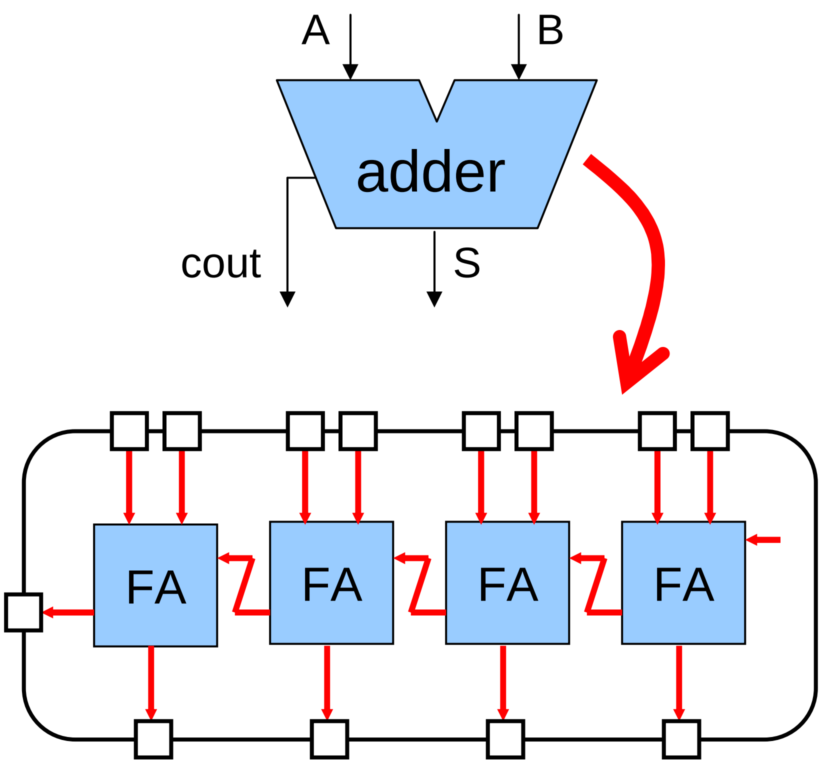



```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

  FA fa0( ... );
  FA fa1( ... );
  FA fa2( ... );
  FA fa3( ... );

endmodule
```

Hierarchical Modeling with Verilog

A module can contain other modules through module instantiation creating a module hierarchy

- Modules are connected together with nets
- Ports are attached to nets either by position



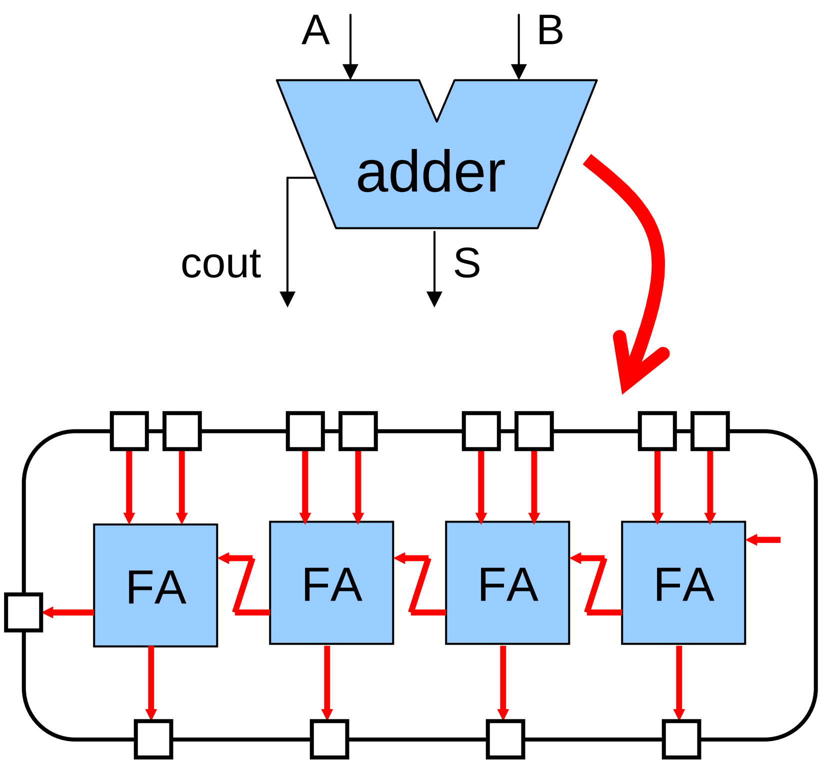

```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] S );

  wire c0, c1, c2;
  FA fa0( A[0], B[0], 0,  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

Hierarchical Modeling with Verilog

A module can contain other modules through module instantiation creating a module hierarchy

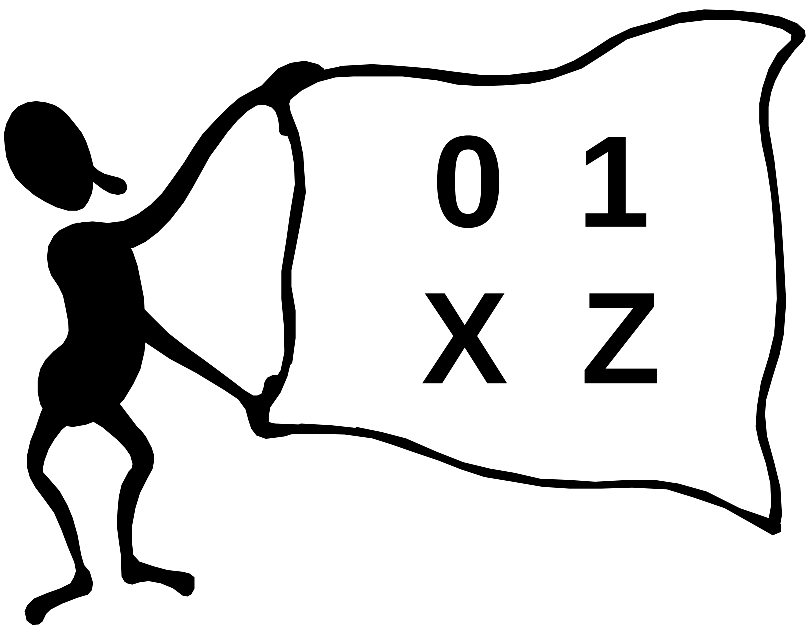
- Modules are connected together with nets
- Ports are attached to nets either by position or by name




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

  wire c0, c1, c2;
  FA fa0( .a(A[0]), .b(B[0]),
          .cin(0), .cout(c0),
          .sum(S[0] );

  FA fa1( .a(A[1]), .b(B[1]),
          ...

endmodule
```

Verilog Basics

Data Values



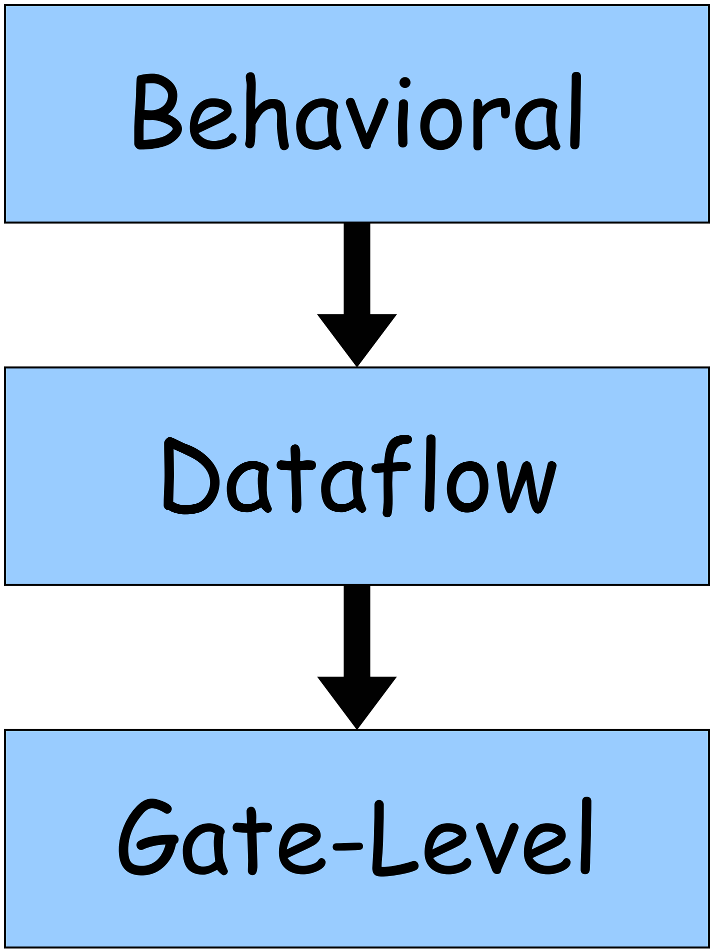
Numeric Literals

4'b10_11

- Decimal number representing size in bits (4)
- Base format (d,b,o,h)
- Underscores are ignored

32'h8XXX_XXA3

3 Common Abstraction Levels

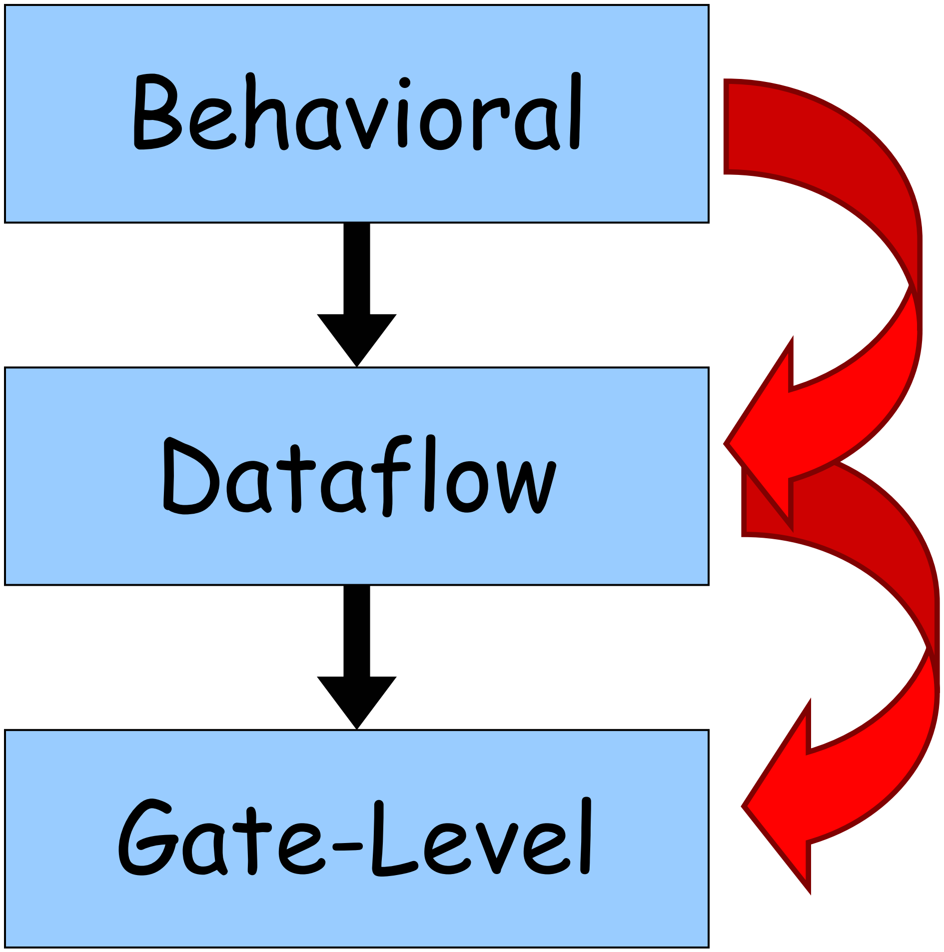


Module's high-level algorithm is implemented with little concern for the actual hardware

Module is implemented by specifying how data flows between registers

Module is implemented in terms of concrete logic gates (AND, OR, NOT) and their interconnections

3 Common Abstraction Levels



Designers can create lower-level models from the higher-level models either manually or automatically

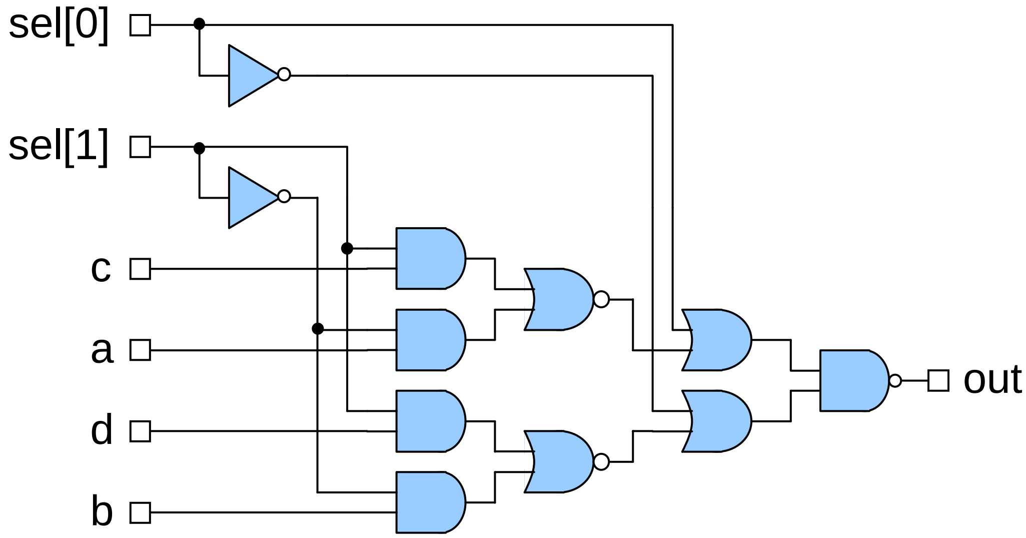
The process of automatically generating a gate-level model from either a dataflow or a behavioral model is called

Logic Synthesis

Gate-Level : 4-input Multiplexer

```
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
```

Basic logic gates are built-in primitives meaning there is no need to define a module for these gates



Dataflow : 4-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
```

This is called a **continuous assignment** since the RHS is always being evaluated and the result is continuously being driven onto the net on the LHS

Dataflow : 4-input Multiplexer

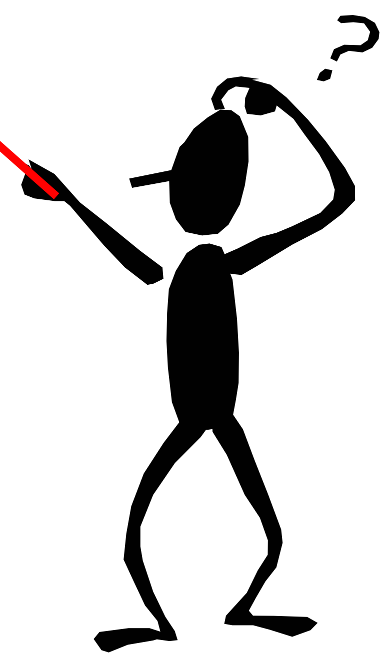
```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

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

endmodule
```

An implicit continuous assignment combines the net declaration with an assign statement and thus is more succinct

Dataflow : 4-input Mux and Adder

```
// Four input muxltiplexor
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
```

Dataflow style Verilog enables descriptions which are more abstract than gate-level Verilog



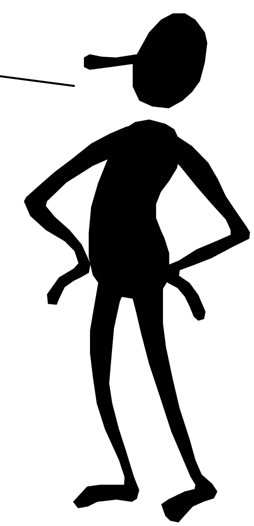
Dataflow : Key Points

Dataflow modeling enables the designer to focus on where the state is in the design and how the data flows between these state elements without becoming bogged down in gate-level details

- Continuous assignments are used to connect combinational logic to nets and ports
- A wide variety of operators are available including:

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

Avoid these operators since they usually synthesize poorly



Dataflow : Key Points

Dataflow modeling enables the designer to focus on where the state is in the design and how the data flows between these state elements without becoming bogged down in gate-level details

- Continuous assignments are used to connect combinational logic to nets and ports
- A wide variety of operators are available including:

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

```
assign signal[3:0]
    = { a, b, 2'b00 }
```

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( a or b or c or d or sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

An always block is a behavioral block which contains a list of expressions which are (usually) evaluated sequentially

The code in an always block can be very abstract (similar to C code) – here we implement a mux with an if/else statement

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( a or b or c or d or sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

An always block can include a sensitivity list – if any of these signals change then the always block is executed

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( a, b, c, d, sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

In Verilog-2001 we can use a comma instead of the or

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( a, b, c, d, sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

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

The always block will not execute if just d changes – so if `sel == 3` and d changes then out will not be updated

This will cause discrepancies between simulated and synthesized hardware – there are no sensitivity lists in real hardware so it would work fine!

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( * )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

In Verilog-2001 we can use the @(*) construct which creates a sensitivity list for all signals read in the always block

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

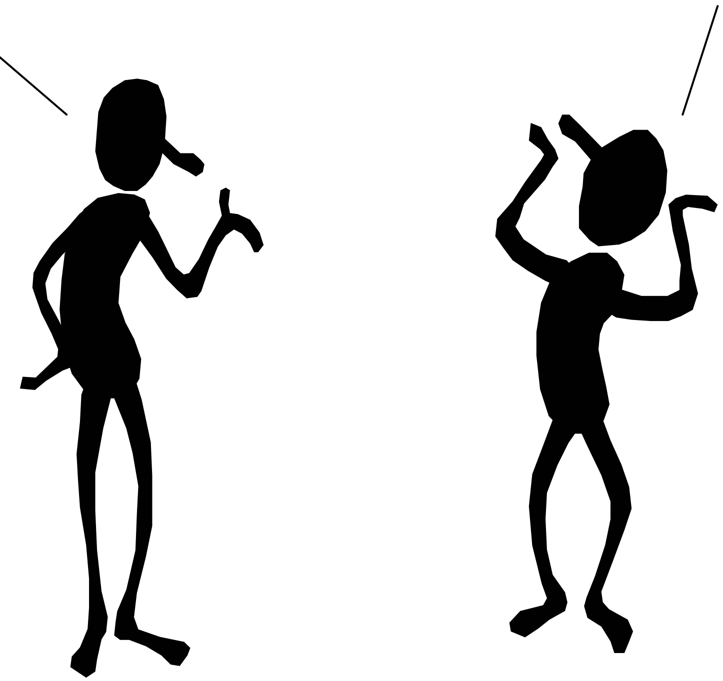
Always blocks can contain case statements, for loops, while loops, even functions – they enable high-level behavioral modeling

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

What about this funny reg statement? Is this how you create a register in Verilog?

No! and whoever decided on the reg syntax really messed things up!



Behavioral : 4-input Multiplexer

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

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

In Verilog a reg is just a variable – when you see reg think variable not hardware register!

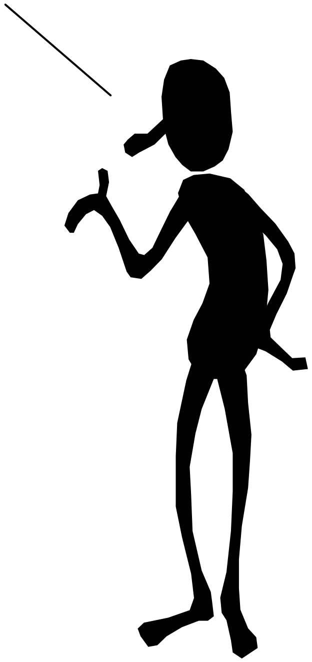
Any assignments in an always block must assign to a reg variable – the reg variable may or may not actually represent a hardware register

If the always block assigns a value to the reg variable for all possible executions then the reg variable is not actually a hardware register

Behavioral : 4-input Multiplexer

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

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

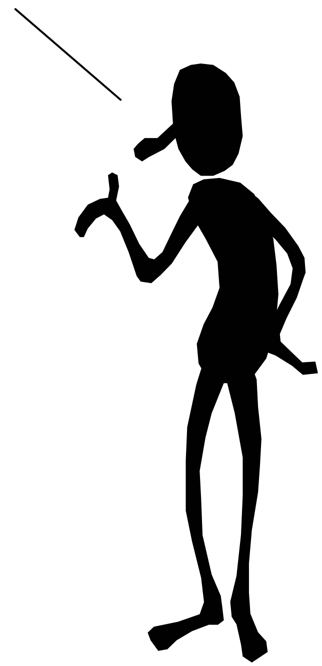
What about in this situation? Will the generated hardware include a latch for out?



Behavioral : 4-input Multiplexer

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

  reg out;

  always @( * )
  begin
    case ( sel )
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

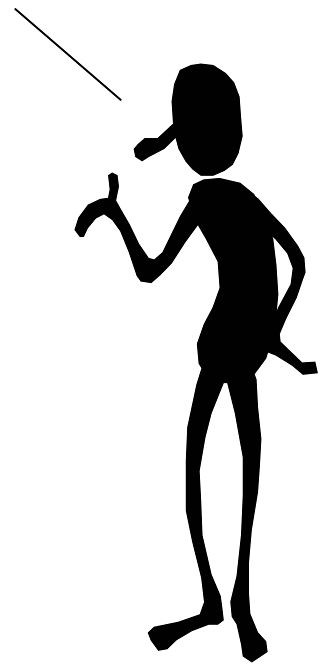
Maybe! What if sel == xx? Then out is unassigned and the hardware must maintain the previous value of out!



Behavioral : 4-input Multiplexer

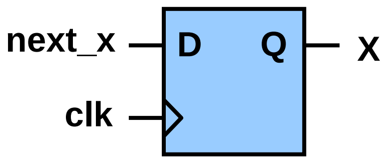
```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    case ( sel )
      default : out = 1'bx;
      0 : out = a;
      1 : out = b;
      2 : out = c;
      3 : out = d;
    endcase
  end

endmodule
```

Fix it with a default clause in the case statement – then no hardware latch is inferred

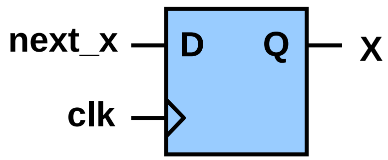


Behavioral Non-Blocking Assignments

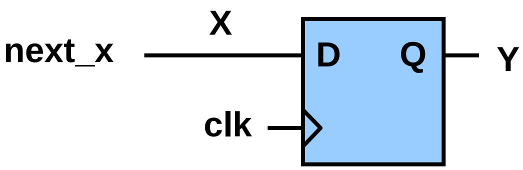
```
always @( posedge clk )
begin
  x = next_x;
end
```



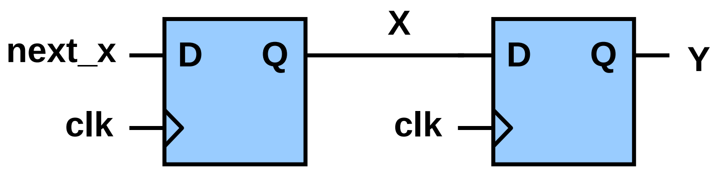

```
always @( posedge clk )
begin
  x <= next_x;
end
```




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




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

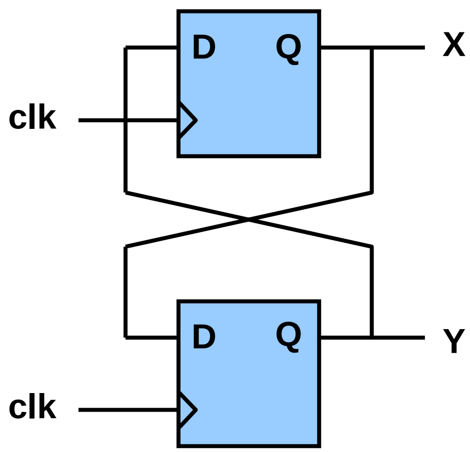


Behavioral Non-Blocking Assignments

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

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

X	Y

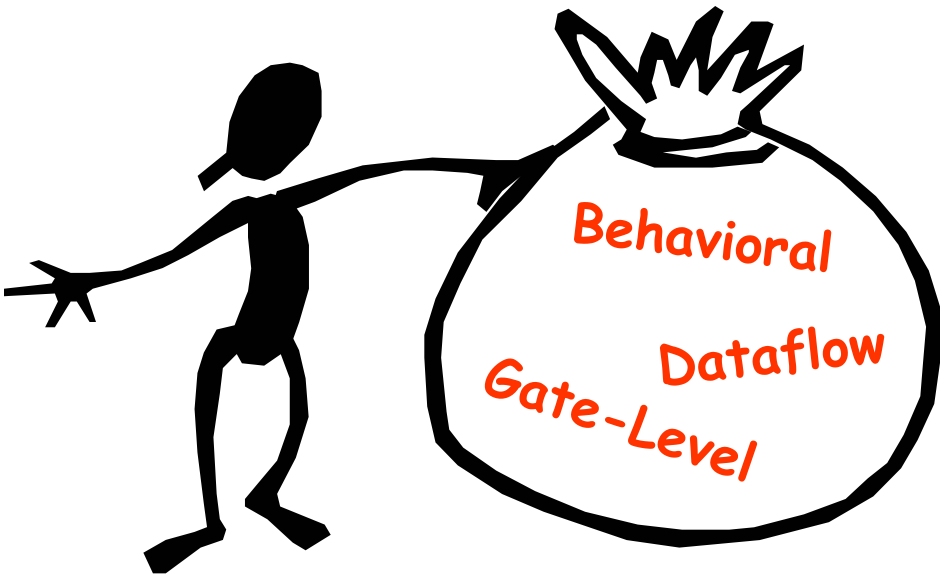


Take Away Point - always ask yourself "Do I need blocking or non-blocking assignments for this always block?"

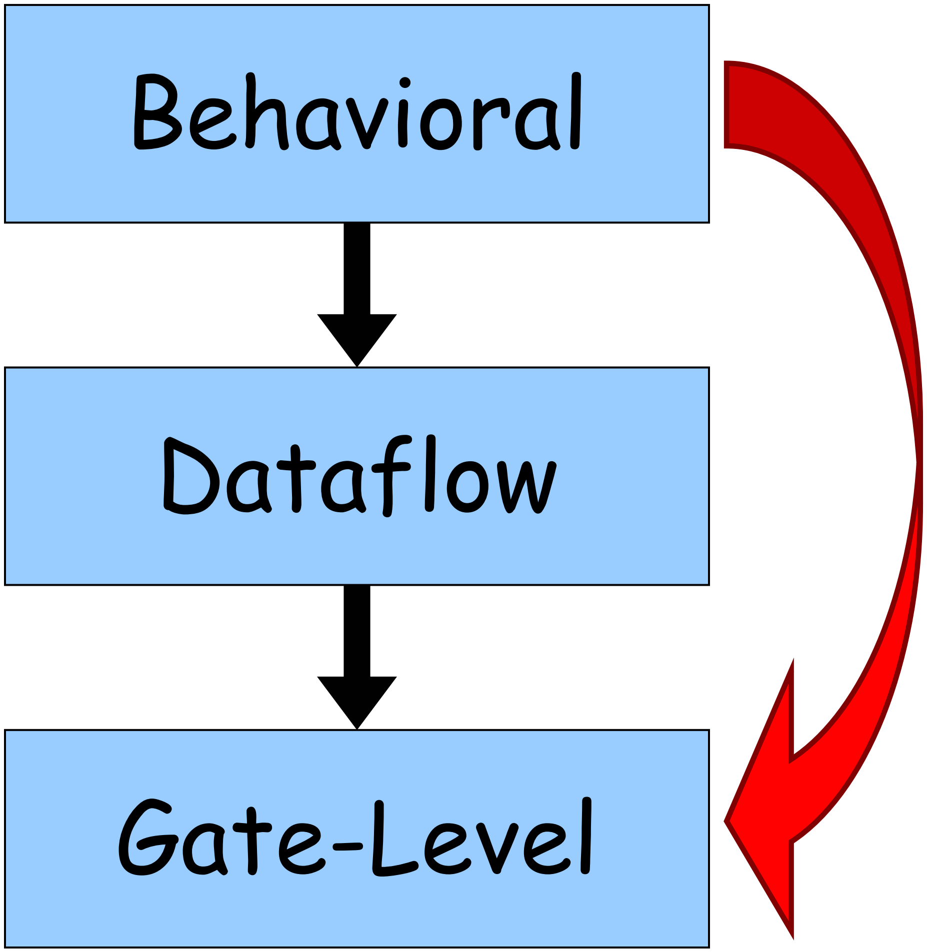
Never mix and match!

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.



Revisiting Logic Synthesis

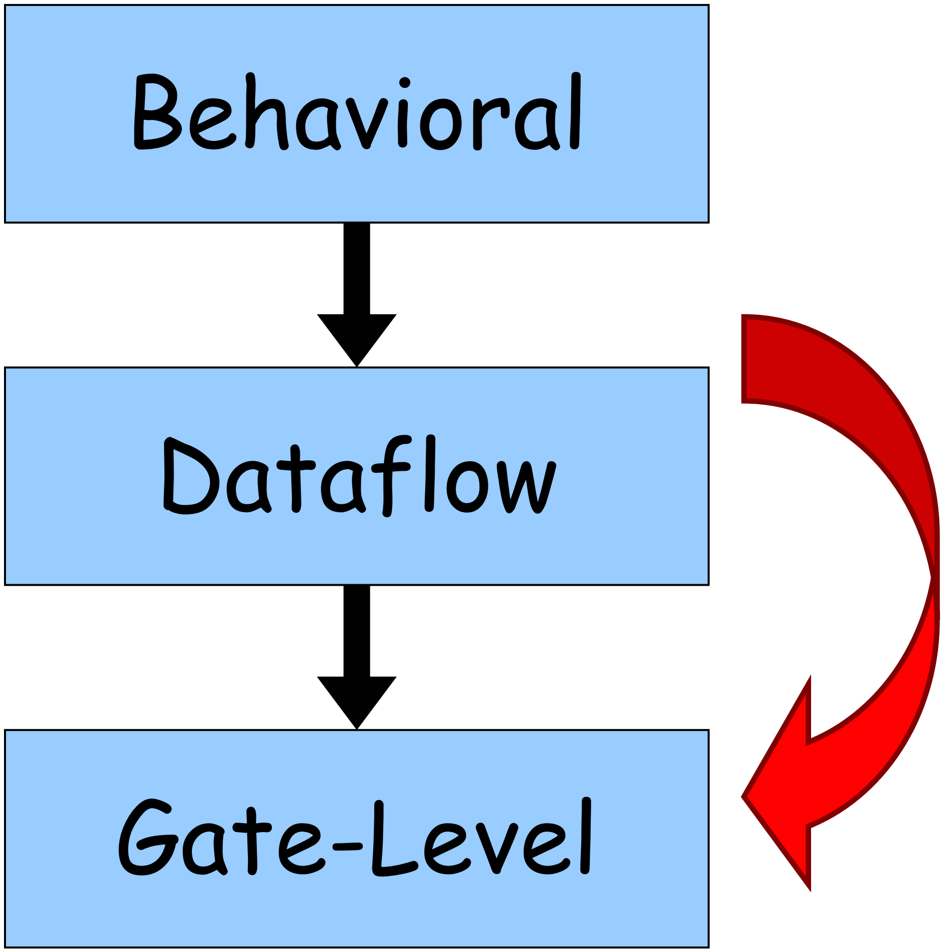


Modern tools are able to synthesize more and more behavioral Verilog code directly to the gate-level

The problem though, is that it is very hard to predict what the generated hardware will look like

This makes it difficult to perform rational design space exploration

Revisiting Logic Synthesis



In this course we will mostly stick to very predictable dataflow to gate-level synthesis – we want to have a good idea what kind of hardware we are generating!

Writing Parameterized Models

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

  ...

endmodule

// Specify parameters at instantiation time
mux4 #( .width(32) )
      alu_mux( .a(op1), .b(bypass), .c(32'b0), .d(32'b1),
               .sel(alu_mux_sel), .out(alu_mux_out) );
```

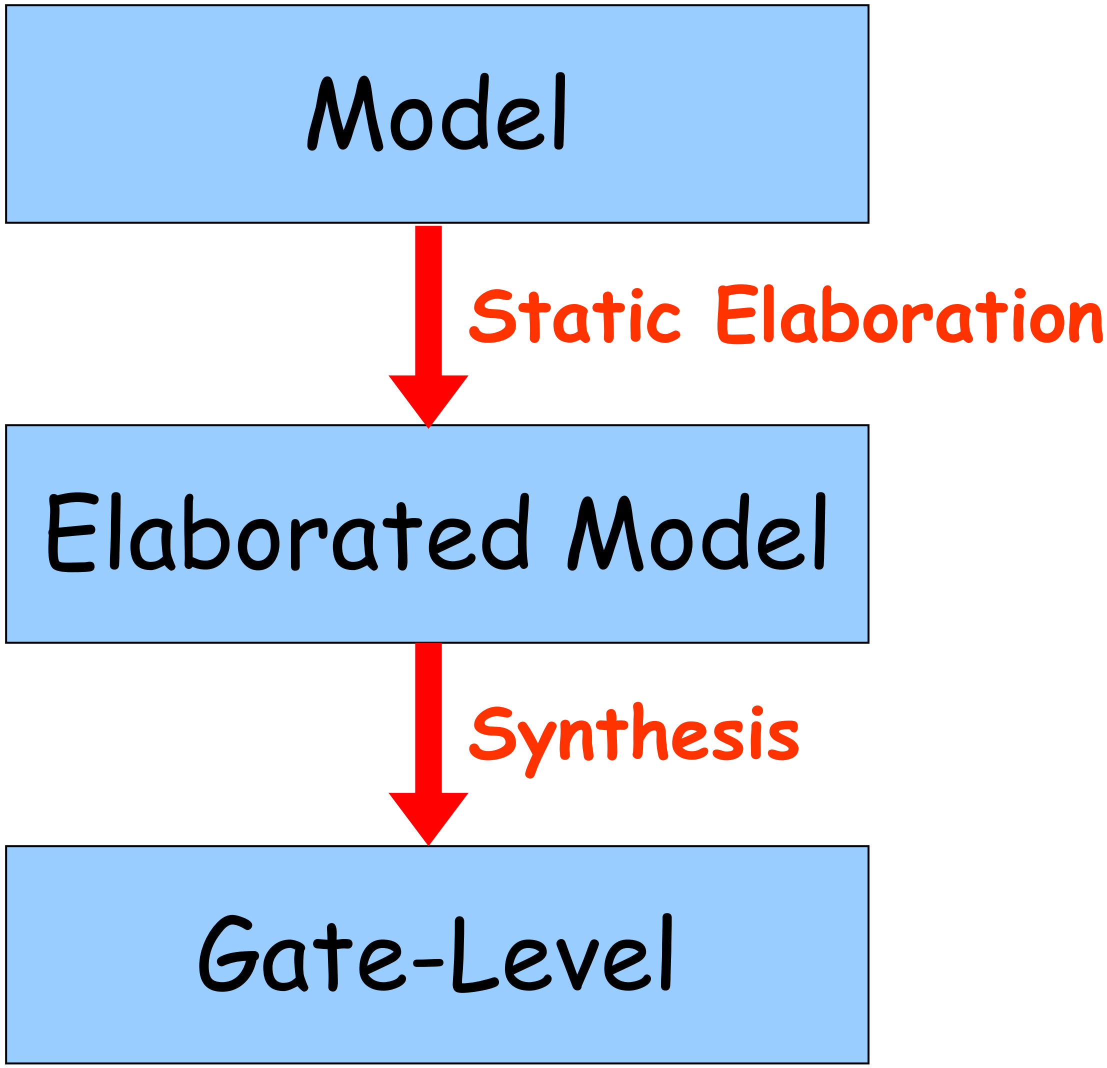
Parameters enable static configuration of modules at instantiation time and can greatly increase the usefulness of your modules

Writing Parameterized Models

```
module adder #( parameter width )
              ( input  [width-1:0] op1,op2,
                output cout,
                output [width-1:0] sum );

  wire [width-1:0] carry;
  assign carry[0] = 0;
  assign cout = carry[width]

  genvar i;
  generate
    for ( i = 0; i < width; i = i+1 )
    begin : ripple
      FA fa( op1[i], op2[i],
             carry[i], carry[i+1] );
    end
  endgenerate

endmodule
```

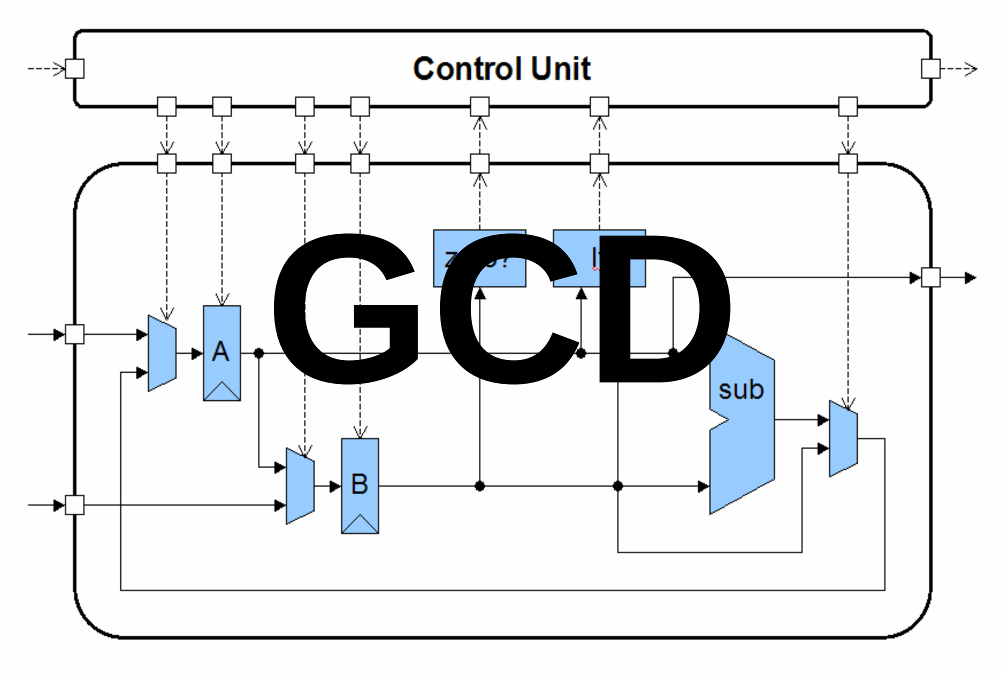
Generate blocks can use parameters to instantiate a variable number of sub-modules or to create a variable number of nets

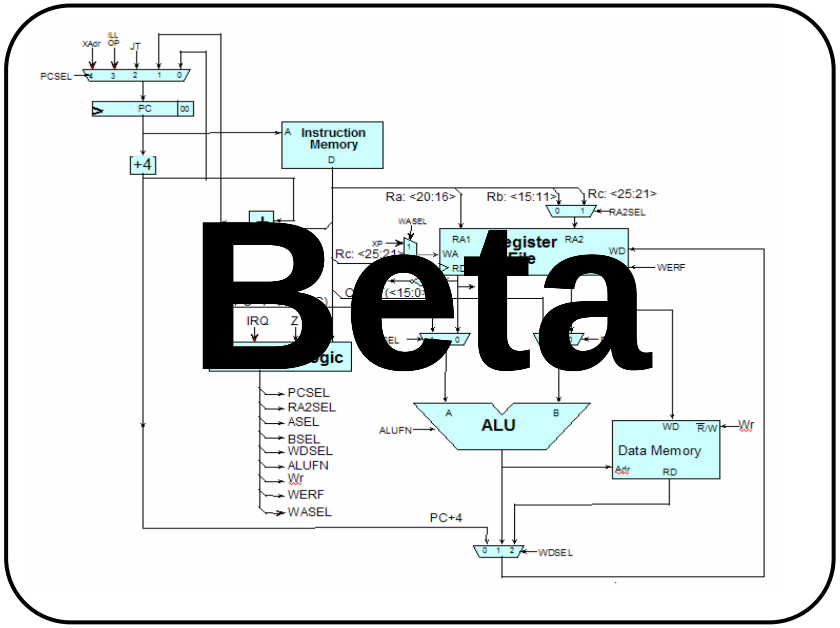
Static Elaboration



Larger Examples

Let's briefly examine two larger digital designs and consider the best way to model these designs in Verilog





GCD Behavioral Example

```
module gcd_behavioral #( parameter width = 16 )
                       ( input  [width-1:0] A_in, B_in,
                         output [width-1:0] Y );

  reg [width-1:0] A, B, Y, swap;
  integer         done;

  always @( A_in or B_in )
  begin
    done = 0;
    A = A_in; B = B_in;

    while ( !done )
    begin
      if ( A < B )
      begin
        swap = A;
        A = B;
        B = swap;
      end
      else if ( B != 0 )
        A = A - B;
      else
        done = 1;
    end

    Y = A;
  end

endmodule
```

We write the general algorithm in an always block using a very C-like syntax

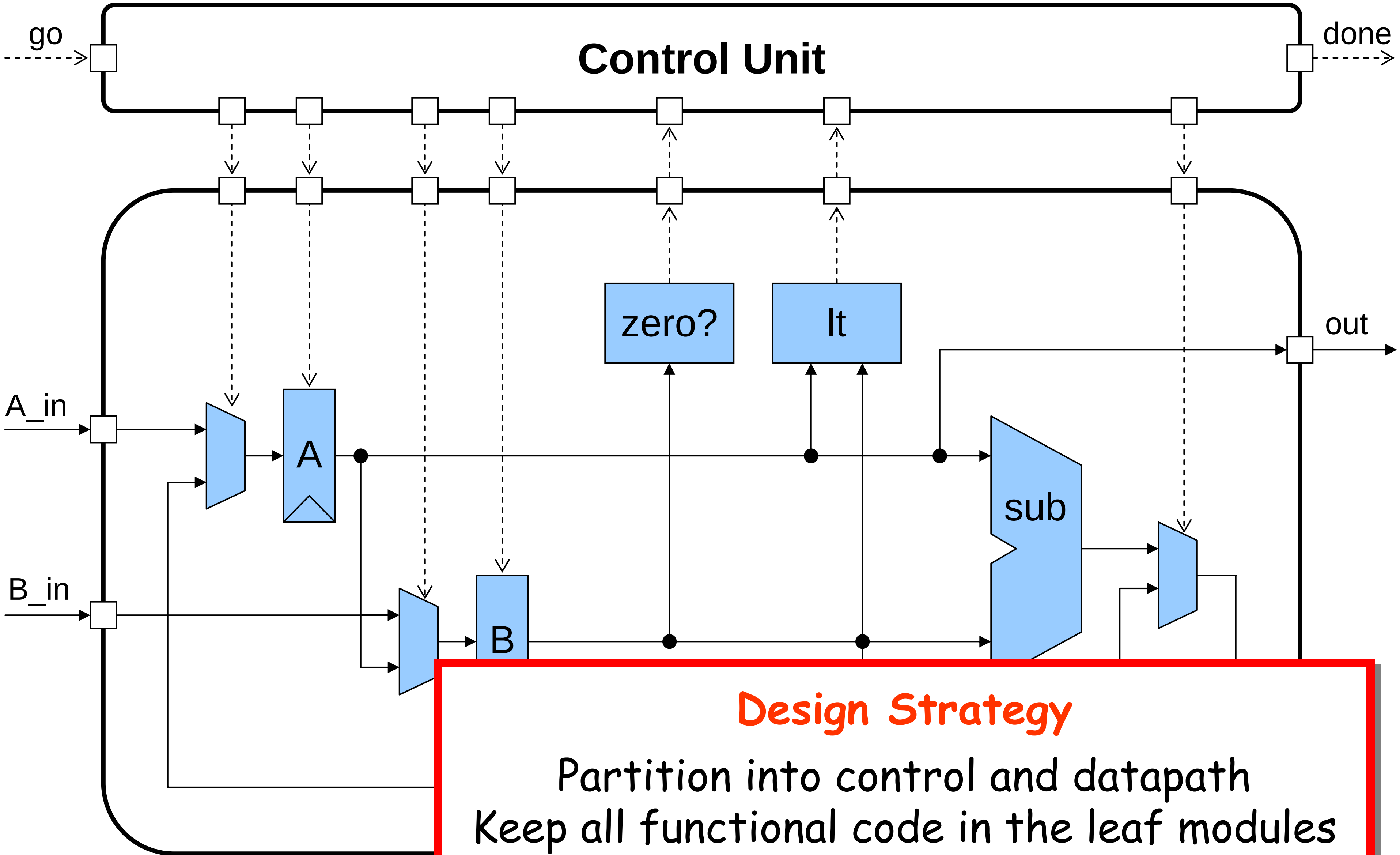
GCD Behavioral Test Harness

```
module gcd_test;
  parameter width = 16;

  reg  [width-1:0] A_in, B_in;
  wire [width-1:0] Y;

  gcd_behavioral #( .width(width) )
                gcd_unit( .A_in(A_in), .B_in(B_in), .Y(Y) );

  initial
  begin

    // Default inputs if cmdline args
    // are not provided
    A_in = 27;
    B_in = 15;

    // Read in cmdline args
    $value$plusargs("a-in=%d",A_in);
    $value$plusargs("b-in=%d",B_in);

    // Let the simulation run
    #10;

    // Output the results
    $display(" a-in    = %d", A_in );
    $display(" b-in    = %d", B_in );
    $display(" gcd-out = %d", Y    );
    $finish;

  end

endmodule
```

We use a test harness to drive the GCD module. The test harness includes an initial block, which is similar to always block except it executes only once at time = 0.

Special directives which begin with $ enable the test harness to read command line arguments, use file IO, print to the screen, and stop the simulation

GCD RTL Example



Design Strategy

Partition into control and datapath
Keep all functional code in the leaf modules

GCD RTL Datapath

```
module gcd_dpath #( parameter width = 16 )
                 ( input  clock,
                   input  A_en, B_en, A_mux_sel, B_mux_sel, out_mux_sel,
                   input  [width-1:0] A_in, B_in,
                   output B_zero, A_lt_B,
                   output [width-1:0] Y );

  reg [width-1:0] A, B;
  assign Y = A;

  // Datapath logic
  wire [width-1:0] out    = ( out_mux_sel ) ? B : A - B;
  wire [width-1:0] A_next = ( A_mux_sel ) ? out : A_in;
  wire [width-1:0] B_next = ( B_mux_sel ) ? A : B_in;

  // Generate output control signals
  wire B_zero = ( B == 0 );
  wire A_lt_B = ( A < B );

  // Edge-triggered flip-flops
  always @( posedge clock )
  begin
   if ( A_en )
     A <= A_next;
   if ( B_en )
     B <= B_next;
  end

endmodule
```

Edge-triggered flip-flops with enables

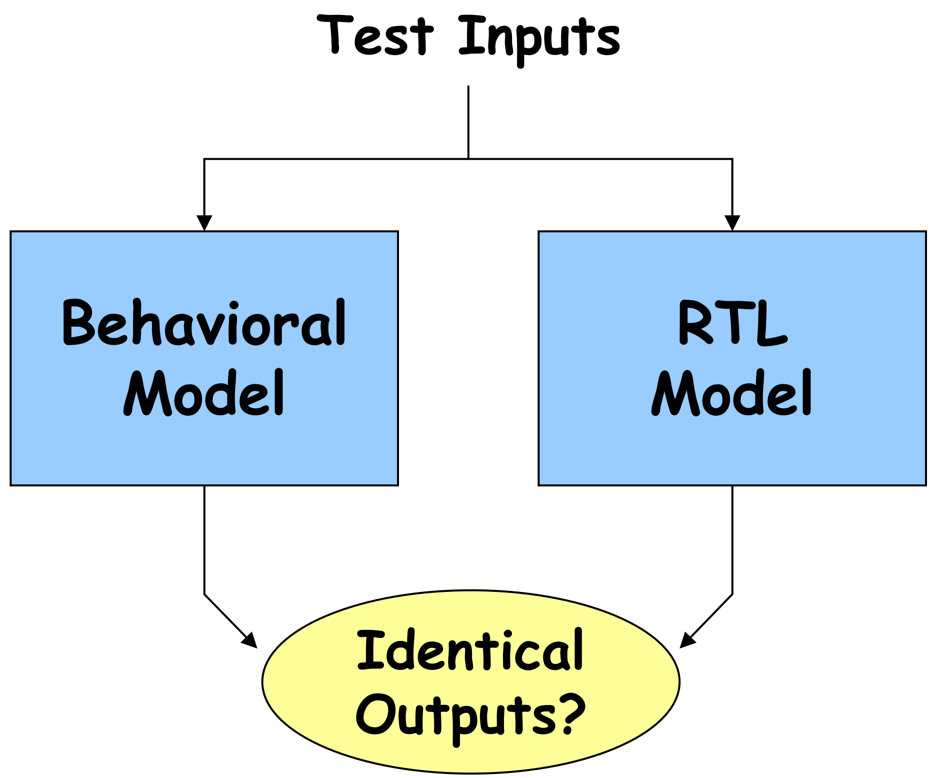
A mix of dataflow and behavioral

GCD RTL Control Unit

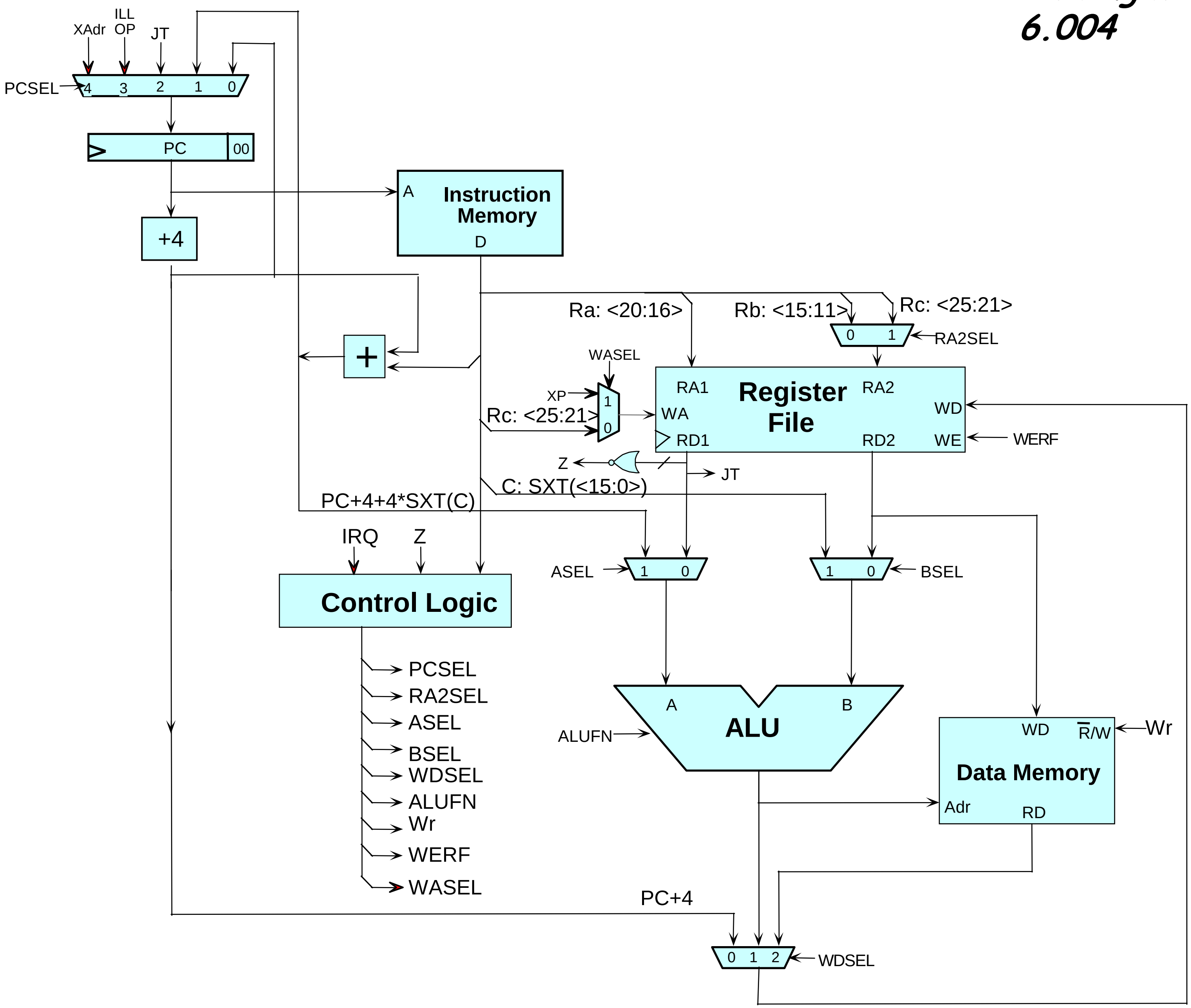
```
module gcd_ctrl ( input  clock, reset, go,
                  input  B_zero, A_lt_B,
                  output A_en, B_en, A_mux_sel, B_mux_sel, out_mux_sel,
                  output done );

  // The running bit is one after go goes high and until done goes high

  reg running = 0;
  always @( posedge clock )
  begin
    if ( go )        running <= 1;
    else if ( done ) running <= 0;
  end

  // Combinational control logic - we group all the control signals
  // onto one bus to make the Verilog more concise

  reg [5:0] ctrl_sig;
  assign { A_en, B_en, A_mux_sel, B_mux_sel, out_mux_sel, done } = ctrl_sig;

  always @(*)
  begin
    if ( !running )       ctrl_sig = 6'b11_00x_0; // Latch in A and B values
    else if ( A_lt_B )    ctrl_sig = 6'b11_111_0; // A <= B and B <= A
    else if ( !B_zero )   ctrl_sig = 6'b10_1x0_0; // A <= A - B and B <= B
    else                  ctrl_sig = 6'b00_xxx_1; // Done
  end

endmodule
```

GCD Testing

We use the same test inputs to test both the behavioral and the RTL models. If both models have the exact same observable behavior then the RTL model has met the functional specification.



Beta Redux


XAdr
ILL OP
JT
PCSEL
4 3 2 1 0
PC
00
+4
Instruction Memory
A
D
Ra: <20:16>
Rb: <15:11>
Rc: <25:21>
0 1
RA2SEL
+
WASEL
XP
Rc: <25:21>
1
0
RA1
Register File
RA2
WA
WD
RD1
RD2
WE
WERF
Z
JT
C: SXT(<15:0>)
PC+4+4*SXT(C)
IRQ
Z
Control Logic
ASEL
1 0
1 0
BSEL
PCSEL
RA2SEL
ASEL
BSEL
WDSEL
ALUFN
Wr
WERF
WASEL
A
B
ALU
ALUFN
WD
R/W
Wr
Data Memory
Adr
RD
PC+4
0 1 2
WDSEL


I thought I already did 6.004

Goals for the Beta Verilog Description

Readable, correct code that clearly captures the architecture diagram – "correct by inspection"

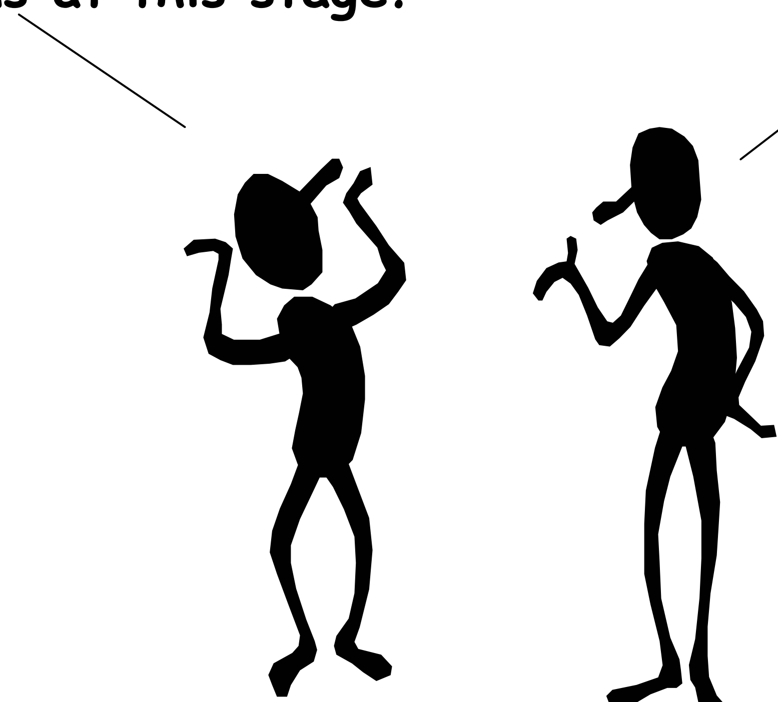
Partition the design into regions appropriate for different implementation strategies. Big issue: wires are "bad" since they take up area and have capacitance (impacting speed and power).

- Memories: very dense layouts, structured wires pretty much route themselves, just a few base cells to design & verify.
- Datapaths: each cell contains necessary wiring, so replicating cells (for N bits of datapath) also replicates wiring. Data flows between columnar functional units on horizontal busses and control flows vertically.
- Random Logic: interconnect is "random" but library of cells can be designed ahead of time and characterized.
- Think about physical partition since wires that cross boundaries can take lots of area and blocks have to fit into the floorplan without wasteful gaps.

Hey! What happened to abstraction?

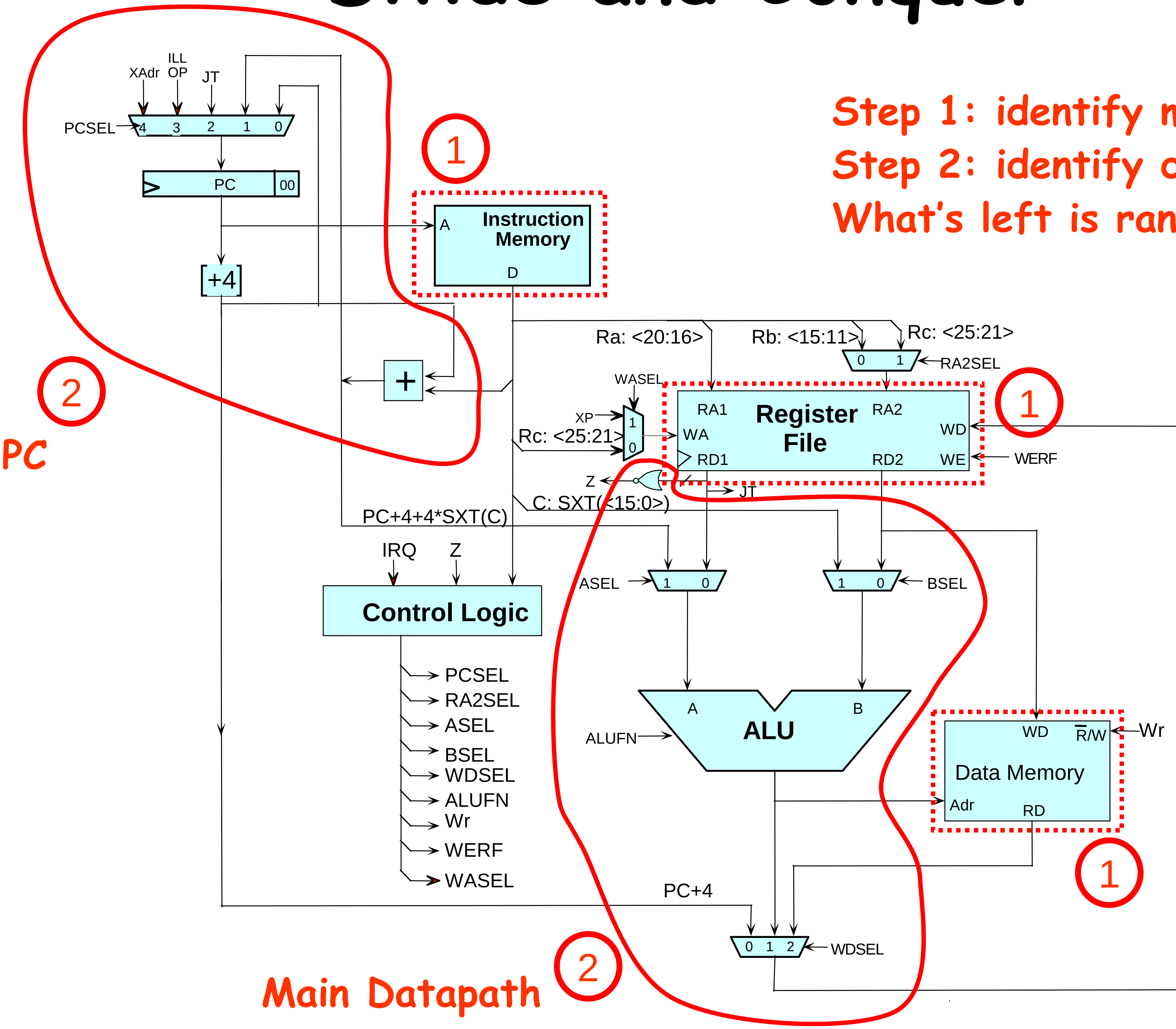
Wasn't the plan to abstract-away the physical details so we could concentrate on getting the functionality right? Why are we worrying about wires and floorplans at this stage?

Because life is short! If you have the luxury of writing two models (the first to experiment with function, the second to describe the actual partition you want to have), by all means! But with a little experience you can tackle both problems at once.



Divide and Conquer

Step 1: identify memories
Step 2: identify datapaths
What's left is random logic …



Take Away Points

Hardware description languages are an essential part of modern digital design

- HDLs can provide an executable functional specification
- HDLs enable design space exploration early in design process
- HDLs encourage the development of automated tools
- HDLs help manage complexity inherent in modern designs

Verilog is not a software programming language so always be aware of how your Verilog code will map into real hardware

Carefully plan your module hierarchy since this will influence many other parts of your design

Laboratory 1

You will be building an RTL model of a two-stage MIPS processor

1. Read through the lab and the SMIPS processor spec which is posted on the website
2. Look over the Beta Verilog posted on the website
3. Try out the GCD Verilog example in 38-301 (or on any Athena/Linux machine)

```
% setup 6.884
% cp –r /mit/6.884/examples/gcd .
% cat gcd/README
```

4. Next week's tutorial will review the Beta implementation and describe how to use Lab 1 toolchain (vcs, virsim, smips-gcc)