A Brief Introduction to

Verilog

Hardware Definition Language (HDL)

Forward

Verilog is a “Hardware Description” language (HDL) that is used to define the structure and/or behavior of digital circuits. Verilog is a “concurrent” language, different than a “procedural” language like C or Java.

Verilog is the most widely used HDL industry today, but VHDL is catching up. Although the two languages are different, they are quite similar in many ways. If you learn Verilog, it will only take a few hours to convert to VHDL.

Concepts: Behavioral vs. Structural Models

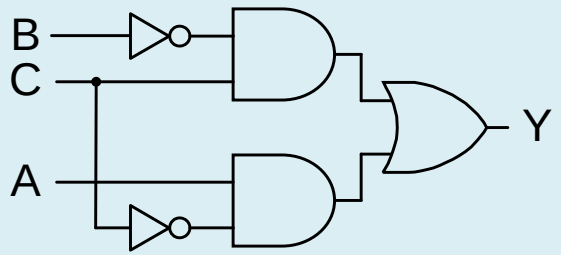
“Behavioral” code defines outputs as functions of inputs, without describing any circuit components or modules that might be used in constructing the circuit.

“Structural” code is a form of netlist, defining a circuit as a collection of subordinate components or modules and their interconnecting wires.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Drive an output signal Y to a ‘1’ whenever input B is not asserted at the same time C is, or when A is asserted when C is not.

Behavioral descriptions define input-to-output relationships, but provide no information about how to construct a circuit



I$001: INV(B,N$001)
I$002: AND2(N$001,C,N$002)
I$003: INV(C,N$003)
I$004: AND2(A,N$003,N$004)
I$005: OR2(N$002,N$004,Y)

Structural descriptions show how lower-level components (like logic gates) are interconnected, but the input/output behavior must be deduced.

Concepts: Behavioral vs. Structural Models

Behavioral descriptions are more abstract, higher-level, quicker and easier to write, easier for others to understand and follow, and largely self documenting.

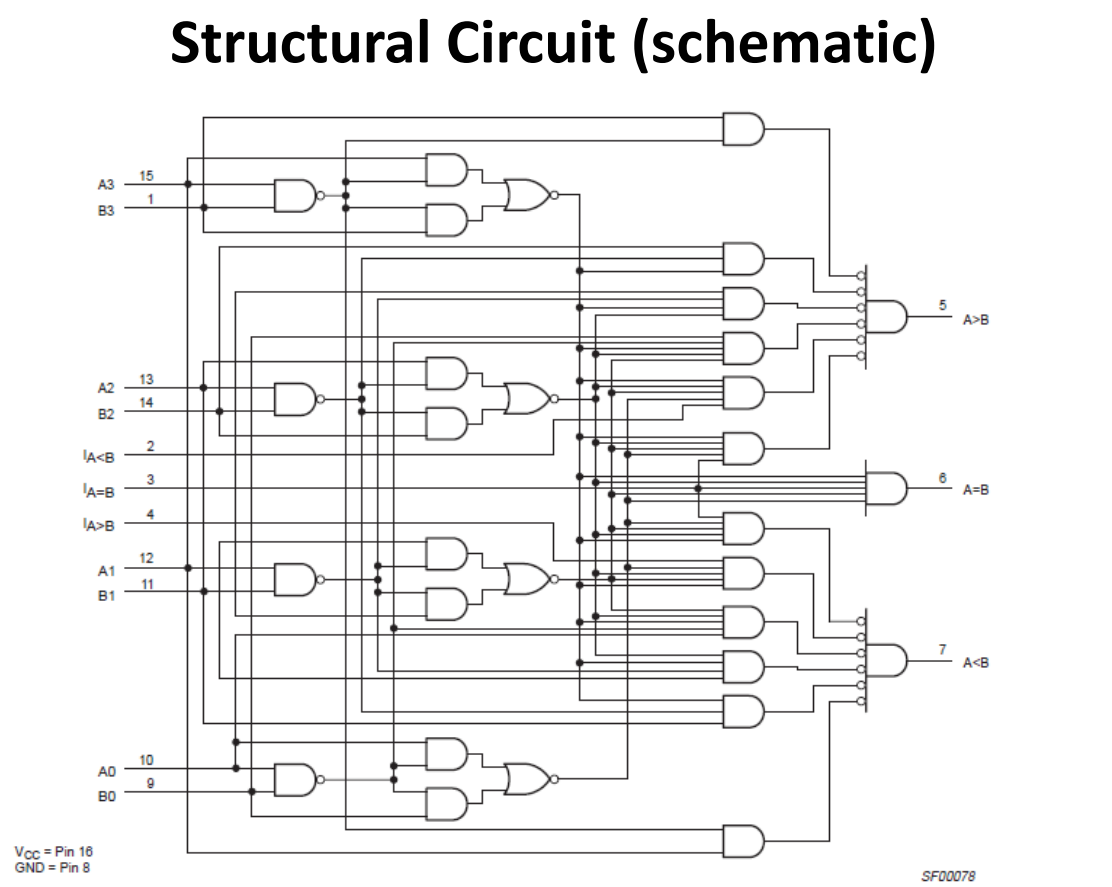
Structural descriptions are often used when existing IP blocks can be reused. They are easier to debug, easier to analyze (for timing and area), and can be easier to optimize.

Most designers write ***behavioral code*** for individual, lower-level circuits, and ***structural code*** when reusing existing IP or connecting lower-level blocks into more complex circuits.

Example: Behavioral vs. Structural Models

This example compares a structural 4-bit comparator schematic a Verilog behavioral description. The Verilog description is far easier and faster to create, its function is clear to the reader, and it is portable between CAD tools.

Structural Circuit (schematic)



Verilog Wire Assignments

```
GT <= (A > B) ? 1'b1 : 1'b0;
LT <= (A < B) ? 1'b1 : 1'b0';
EQ <= (A == B) ? 1'b1 : 1'b0;
```

Example: Behavioral vs. Structural Models

This example compares structural vs. behavioral Verilog for a 4-bit adder. Same advantages!

```
module full_adder(x,y,cin,s,cout);
    input x,y,cin;
    output s,cout;
    wire s1,c1,c2,c3;
    xor(s1,x,y);
    xor(s,cin,s1);
    and(c1,x,y);
    and(c2,y,cin);
    and(c3,x,cin);
    or(cout,c1,c2,c3);
endmodule

module adder_4bit(x,y,cin,sum,cout);
    input [3:0] x,y;
    input cin;
    output[3:0] sum;
    output cout;
    wire c1,c2,c3;
    full_adder fa1(.x(x[0]),.y(y[0]),.cin(cin),.s(sum[0]),.cout(c1));
    full_adder fa2(.x(x[1]),.y(y[1]),.cin(c1),.s(sum[1]),.cout(c2));
    full_adder fa3(.x(x[2]),.Y(y[2]),.cin(c2),.s(sum[2]),.cout(c3));
    full_adder fa4(.x(x[3]),.y(y[3]),.cin(c3),.s(sum[3]),.cout(cout));
endmodule
```

```
module adder_4bit(x,y,sum);
    input [3:0] x,y;
    output[3:0] sum;
    sum = x + y;
endmodule
```

Behavioral Adder

Simpler and easier to read.

Verilog Structural Adder

More work, more detail.

A good thing?

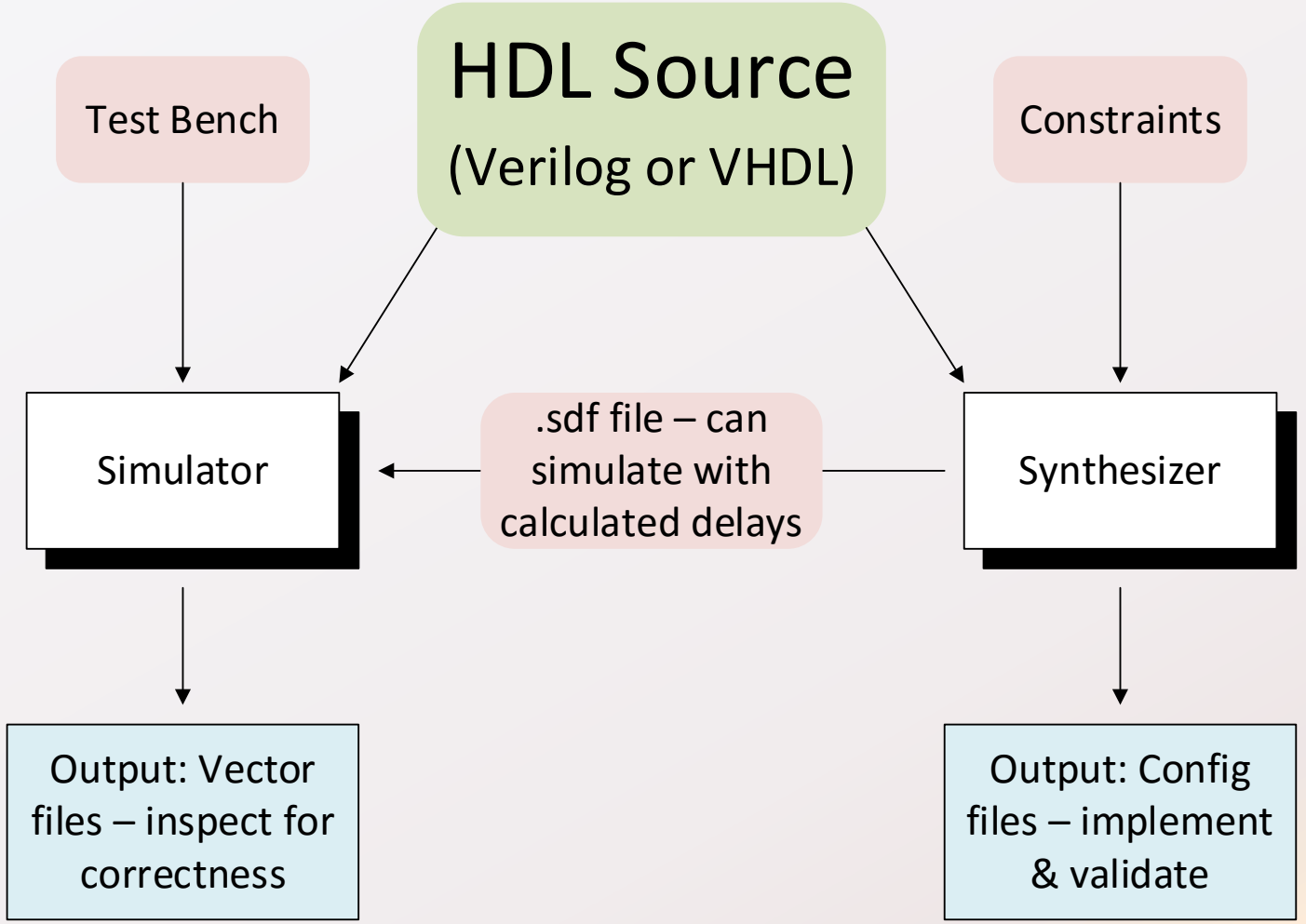
Concept: HDLs can be Simulated and/or Synthesized

Verilog descriptions can be ***simulated*** to check for logical correctness, and ***synthesized*** to automatically create a physical circuit definition.

Simulators were part of the original Verilog release; synthesizers came later.

Simulators are arguably the most important computer tools ever created. Validating circuits and signal timings prior to implementation allows far more complex designs to proceed more quickly and with fewer errors.

Synthesizers create physical circuit descriptions from HDLs. Some descriptions can program an FPGA; others can build custom chips.



Background

- Schematic entry/capture were primary CAD tools until the mid 90’s, but were completely replaced by VHDL and Verilog by the year 2000
  - ✓ Higher-level, behavioral design methods – more complex designs more quickly
  - ✓ Simulation/verification prior to implementation
  - ✓ Synthesis to create physical circuits for a wide range of target technologies
  - ✓ Accurate post-synthesis simulation allows detailed timing analysis
  - ✓ Text-based tools allow platform independent source files (any editor can be used)
- HDLs specify physical circuit behavior, not procedural algorithms
- HDLs can define high-level behavioral or low-level structural models
- HDLs allow designers to easily incorporate IP from a wide range of developers
- Dozens of CAD tools/vendors, many free: Cadence, Mentor, Xilinx, Aldec, etc.

VHDL vs. Verilog

VHDL

- Commissioned/sponsored by US government starting in 1979
- Widespread use by 1987; Specified by IEEE 1076 in 1987
- ADA like
- Strong types, more structure, many features, can be complex

Verilog

- Private company (Gateway Design Automation) starting in 1983
- Widespread use by 1990; Specified by IEEE 1364 in 1995
- C like
- More targeted, less complex

Many free VHDL and Verilog tools are available, and lots of reference and example designs in both languages can be found around the internet

Design Projects

Verilog tools organize the workspace using “Projects”. Projects contain all the source files needed for a given design. External files can also be used by placing them in a library, and making the library visible within a project.

Source files include Verilog modules, constraint files used by the synthesizer to guide implementation, and .sdf files that store signal node timing data.

Verilog source files use ***modules***. Modules are basic Verilog constructs that define circuits. A simple design might use a single module; more complex designs might use several. A project can include any number of modules.

Any Verilog module can be used as a component in any other Verilog source file.

Verilog modules stored in external libraries can be used in other projects.

Verilog code can be written to model and study circuits; to specify circuits and document behavioral requirements; to use/reuse specific pre-existing IP blocks; and to define circuits for synthesis.

Models for Synthesis

Verilog code written to define circuits for synthesis must model physical wires. Wires transport signals between Verilog modules, or between modules and the “outside world”. (*Note*: the outside world is typically a signal connection to a physical pin on an IC package).

Verilog uses “modules” to define circuits and their behaviors.

Verilog uses the data type “wire” to model physical wires; only wires can transport data.

Verilog “wires” use a 4-valued type to model signals: 0, 1, X, Z

Verilog Modules: A first look

Modules are the principle design entity in Verilog. All circuits are defined in modules. A simple design might use a single module; more complex designs might use several. A project can include any number of modules.

The keyword “module” is followed by the module name and the port list. The port specifies all port signals as inputs, outputs, or inout (bi-directional).

Local signals are declared immediately following the module/port statement using the keyword “wire”.

Combinational assignments are made using the assign statement. Sequential assignments are made inside an “always” block. A module can have any number of assign statements and always blocks. Assignment statements are described in the following slides.

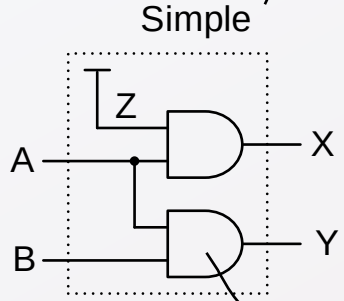
Modules can be instantiated/used in other modules by including the module name and port connection list.

```
module simple(
    input A, B;
    output X, Y;
);

wire Z = 1b’1;

assign X = A & Z;
assign Y = A & B;

end module
```

```
module next(
    input C;
    output K;
);

wire D = 1b’1;
wire F,G;

simple(.A(C),.B(D),.F(X),.G(Y))
assign K = F & G;
end module
```

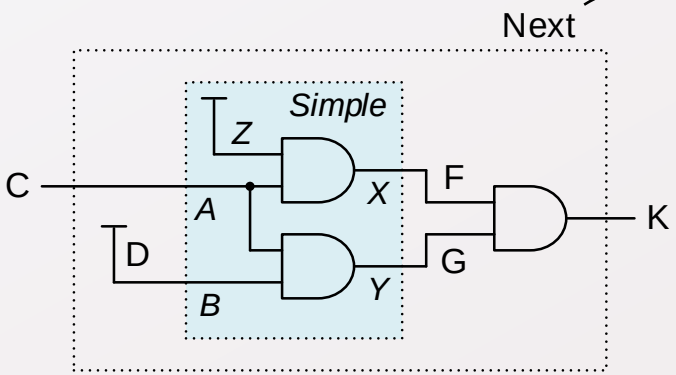
Verilog Modules

The module declaration statement defines the “bounding box”, and the module body statements define the circuit functions




```
module simple(
    input A, B;
    output X, Y;
);

wire Z = 1b’1;
assign X = A & Z;
assign Y = A & B;
end module
```

Any module can be “instantiated” as a component in another module by listing it’s name and port connections. Here, “named association” is used.

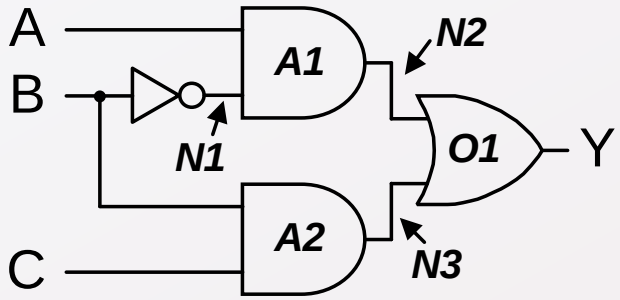


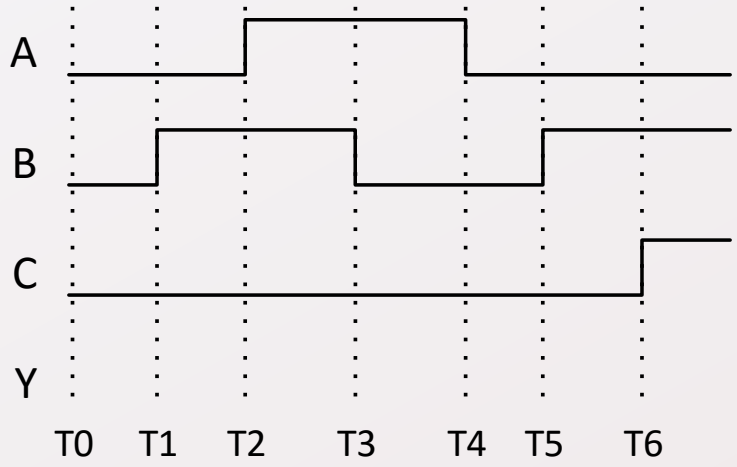
.component_port(topmodule_port)

```
module next(
    input C;
    output K;
);

wire D = 1b’1;
wire F,G;

simple(.A(C),.B(D),.F(X),.G(Y))
assign K = F & G;
end module
```

Concept: Sequential vs. Concurrent Models

- A sequential processing algorithm defines a set of steps that are taken in a specific order
- Concurrent processing steps occur whenever new input data is available, with no implied sequence relationship between separate concurrent processes
- Consider a simulation of a 2-input mux: At what time(s) should gate A1 be simulated? What about gate O1?





- Computer time vs. Simulation time

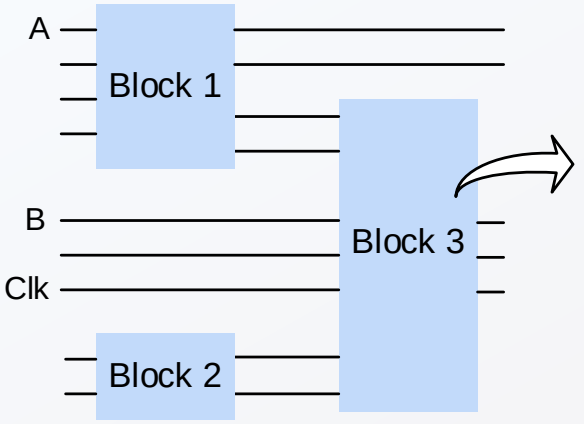
Concurrency: Modelling time

HDL simulators are indispensable tools for designing and learning about digital circuits. They allow every feature, every signal, every critical time to be checked, validated, and studied in detail. You must become proficient at using the simulator to become a skilled designer.

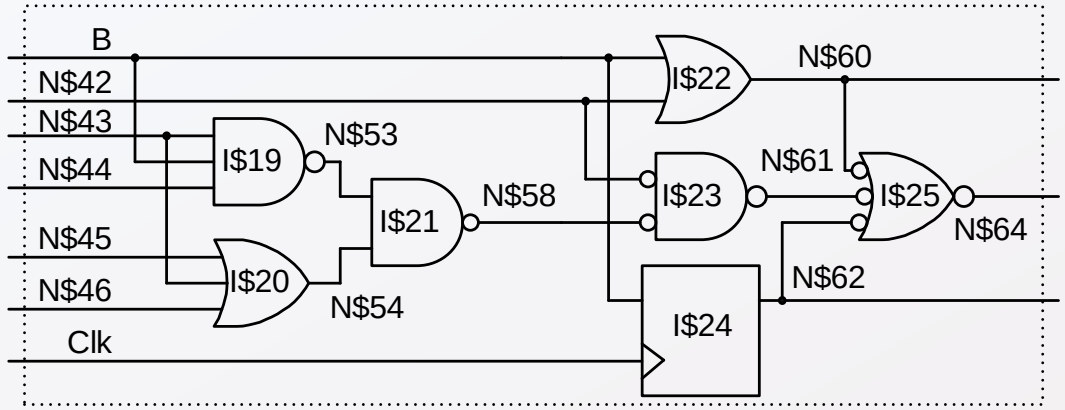
HDL simulators model time by dividing it into arbitrarily small “simulator time steps”, which are the smallest amounts of time that are relevant to the circuit (typically 10ps).

In any given simulator time step, the CPU only simulates circuit components whose inputs have changed. Each simulator time step can take as much CPU time as is needed - during some time steps, many circuit nodes may change, requiring a lot of CPU time. During other steps, very few nodes may change, requiring almost no CPU time.

Concurrency: Modelling time



User Source File



Block 3 Detail

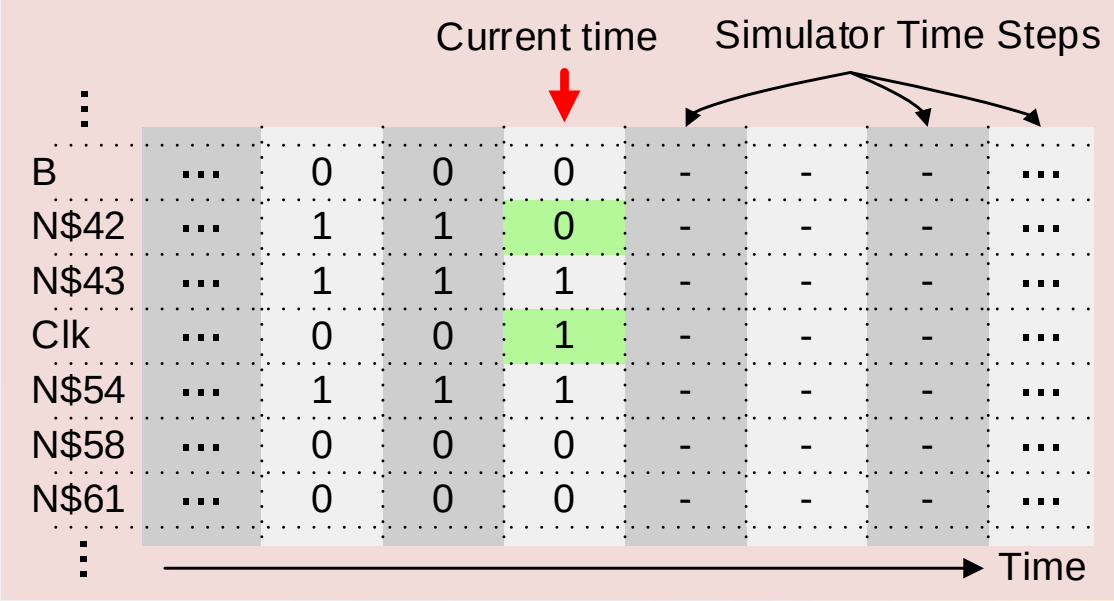
I$19: NAND3(N$43,N$41,N$44,N$53)
I$20: OR3(N$45,N$43,N$46,N$54)
I$21: NAND2(N$53,N$54,N$58)
I$22: OR2(B,N$42,N$60)
I$23: OR2(N$42,N$58,N$61)
I$24: FF(Clk,B,N$62)
I$25: AND3(N$60,N$61,N$62,N$54)

Block 3 Netlist

A Verilog source file is shown as a block diagram, with Block 3 expanded into a schematic and netlist.

All “instances” and “nets” that are not given names in the source file are assigned auto-generated identifiers.

Only some nets will change in any given time step. Here, only N$42 and Clk change. Only instances whose inputs have changed in a given step are simulated. This is called “event driven simulation”.



Concurrency: Modeling time

Assigning new values to output nets requires a variable amount of CPU time for each simulator step, depending on how many different signal assignments must be simulated. “Simulator time” is held constant until all circuit nodes have been simulated and there are no more changes in that step. Only then does simulator time advance.

Verilog combinational assignments are “continuous”. The left hand side is always driven and new output values are assigned as soon as the right hand side is evaluated.

Verilog assign statements are “concurrent”. Assigned outputs are updated immediately after an input changes, regardless of where the assignment statement appears in the source file.

In the code shown, Y changes immediately after A or B change; X changes immediately after A or Z change. The assign statements could swap places in the source file, and there would be no difference in the outputs produced:

```
module simple(
    input A, B;
    output X, Y;
);

wire Z = 1b’1;

assign X = A & Z;
assign Y = A & B;

end module
```

```
assign X = A & Z;
assign Y = A & B;
```

```
assign Y = A & B;
assign X = A & Z;
```

Verilog Concurrency Model

No combinational net should be assigned more than once, because assignment statements are concurrent and the results will not be determinant. The assign statement on the left will have the expected results; the example on the right will not.

```
assign sum = a + b + c;
```

```
assign sum = a + b;
assign sum = sum + c;
```

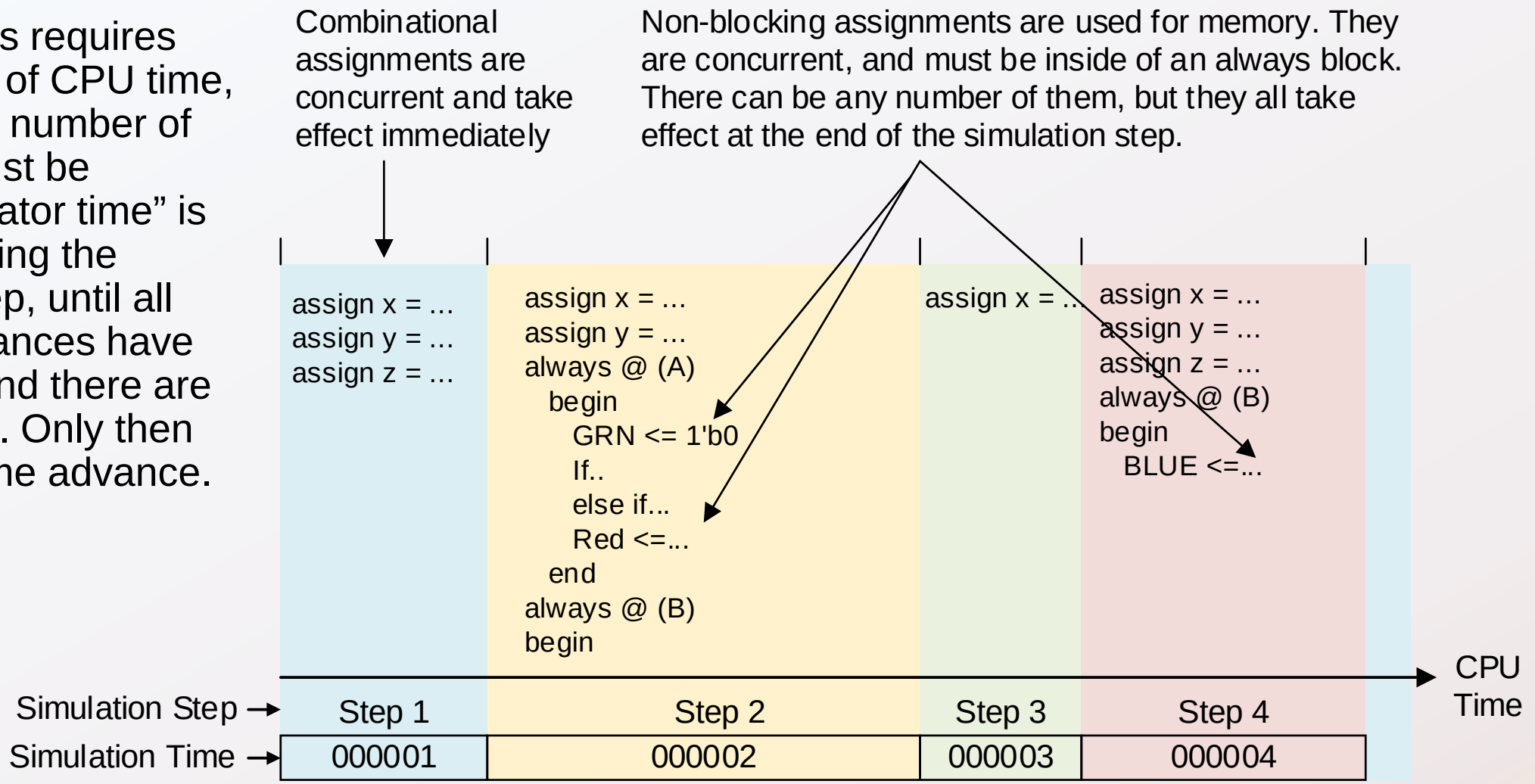
Memory requires a “procedural assignment” so that a timing signal can be checked. Memory assignments use the “non-blocking” operator that updates outputs at the end of the simulation step. Non-blocking procedural assignments can only occur inside an “always” block (described later).

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

In Verilog, flip-flop outputs are updated after a “procedure” checks for a clock edge. If an edge has occurred in the current time step, any required output changes take effect at the end of the time step

Verilog Concurrency Model

Evaluating outputs requires variable amounts of CPU time, depending on the number of instances that must be simulated. “Simulator time” is held constant during the simulator time step, until all circuit nodes/instances have been simulated and there are no more changes. Only then does simulator time advance.



CPU time per simulation step (represented by colored box size) depends on how much processing is required; it will vary for every step. Simulation Time is fixed during each Simulation Step, and only increments at the end of the step.

The Verilog Language

Verilog Syntax: Identifiers and Numbers

Identifiers: space-free upper and lower case letters, numbers, _, and $. Can't begin with a digit; limited to 1024 characters. Verilog ***is*** case sensitive.

Numbers: <size>' <base> <value>

size: decimal number that specifies number of bits

base: ' character followed by b (binary), d (decimal), h (hex)

value: set of digits

Examples: `x = 4'b0101  y = 4'd12  z = 16'hAB12`

White Space: Spaces, tabs, and new lines and form feeds can separate words and are ignored.

Verilog Syntax: Constants and Parameters

Constants: A **parameter** statement defines local parameters (constants) inside of a module. Parameters are not visible outside the module. They can be changed without resynthesizing under certain conditions. Typical uses include defining a bus width, or specifying a number of modules to synthesize in a variable bus-width design.

Localparam is the same, but constants cannot be changed. Localparam is often used to define numbers or state identifiers (Localparam is most commonly used).

Global constants (and macros) can be defined using the **‘define** keyword. They are visible outside the module, throughout the project.

Examples:

```
parameter width = 8;    //“width” can be used throughout module

localparam s0 = 2’b00; //now FSM states can use
localparam s1 = 2’b01; //s0 – s3 as identifiers
localparam s2 = 2’b10;
localparam s3 = 2’b11;

‘define RED 4’b0101    //RED and BLU are visible outside module
‘define BLU 4’d12
```

Verilog Syntax: Operators

+	Add	!	Negation	>	Greater than
-	Subtract	~	Bit-wise negation	<	Less than
*	Multiply	&	Bit-wise And	>=	Greater or equal
/	Divide	|	Bit-wise Or	<=	Lesser or equal
%	Modulus	^	Bit-wise Xor	==	Case equality
<<	Shift left	&&	Logical And	!=	Case inequality
>>	Shift right	||	Logical Or	?	Conditional
[ ]	Bit select	{ }	Concatenation	{{ }}	Replication

Wires and Assignments

Wires transport signals between modules and I/O pins. Wires can connect to other wires, to ‘0’ or ‘1’ logic values, to modules, or to I/O pins.

“assign” statements assign values to wires, but not to memory devices – that requires an “always block” (more on that in a later slide). “assign” statements are “continuous assignment” statements, which means they are always active and take effect immediately. Wires must be declared prior to use with a wire statement. Examples:

```
wire         a, b, c;        //multiple wires can be defined at once
wire         d = 1'b0;       //declares and assigns wire d to 0
wire [3:0]   bus4;           //bus definition
assign a = 1'b1;             //a is assigned to a 1
assign bus4 = {a, 3'b100}    //bus4[3] gets a, bus4[2:0] gets 100
assign b = bus4_A;           //b gets assigned bus4[3]
assign a = b                 //Error! a was already assigned
assign c = b & c | ~d;       //wires can be assigned logic
```

All module input signals (ports) must be wires. Wires declared inside a module are not visible outside that module.

Registers and Assignments

Verilog supports two variable types (wire and reg) and two assignment types: “wire assignments” (pervious slide), and “procedural assignments”. Procedural assignments store outputs in registers (reg) types. Verilog has two procedural assignments: “initial”, run once at the start of simulation, and “always”, which always runs.

Registers/memory device outputs can only be assigned within an always block. Wires cannot be assigned in an always block.

Registers store the last value that was assigned to them (wires store nothing). Reg types are required for instantiating memory. Registers are declared prior to use with a reg statement. Examples:

```
reg         q1, q2;             //multiple OK
reg[3:0]    RA, RB;             //declare two busses
```

Registers can connect to module outputs, but not to inputs (only wires can connect to module inputs).

Possible values for reg and wire variables are 0, 1, X (unknown), and Z (high impedance)

Always Procedural Block

Procedural assignments define sequential behavior – that is, events that happen only under certain conditions.

Always blocks are procedural assignment statements that are ***always*** active throughout the simulation. They execute anytime a signal in the sensitivity list changes. A sensitivity list is a list of signals following the “always” keyword.

The sensitivity list may also contain function calls, like “posedge(clk)”. Posedge(sig) will return “true”, and cause the always block to execute, whenever a positive edge is detected on “sig”.

An always block must be used to infer memory (e.g., flip-flops and latches).

```
module dff(
    input clk, rst, D;
    output Q
);

always @ (posedge(clk), posedge(rst))
begin
   if (rst == 1) Q <= 1'b0;
   else Q <= D;
end

endmodule
```

Example: D Flip-flop

Procedural Assignments: Blocking vs. Non-blocking

Always blocks assign values to variables of type “reg”. Wires cannot be assigned new values.

Reg variables are assigned new values with “blocking” or “non-blocking” assignments. Blocking assignments (=) take effect immediately, but non-blocking assignments (<=) take effect at the end of the simulation step. Non-blocking assignments are ***concurrent***.

Flip-flops are inferred (created) using an always block that checks for a posedge on clk (and/or rst). Output values are assigned using a non blocking (<=) assignment.

Always blocks can define combinational logic, sequential logic, or a mixture of both. If ***only*** combinational logic is defined, then use blocking assignments; otherwise, use non-blocking.

Do not make assignments to the same variable from more than one always block.

A module can have as many procedural (always) blocks as necessary. Statements in a procedural block are executed in order, but the blocks themselves are concurrent to other blocks.

“Initial” procedural blocks are similar to always blocks, but they execute only once at the very start of a simulation. Initial blocks are used to setup initial conditions for simulation.

Procedural blocks: If-then and Case statements

If-then and case statements can only appear in procedural blocks (always blocks). They allow conditions to be checked prior to taking action, and use C-like syntax.

If and case statements can be used to infer memory, by incompletely specifying all possible outcomes of a condition check. *If not all possibilities are specifically covered, synthesis will generate extra latches.*

If and case statements can also be used to define purely combinational logic, but care must be taken to avoid creating unwanted memory.

If statements can have one “else” block, but any number of “else if” blocks. If only one assignment is needed, the *begin… end* keywords are not required.

End case statements with “default”, or risk latches.

```
module if_example(
    input A, B;
    output X;
);

always @ (A)
begin
   if (A) X <= 1'b0;
end

endmodule
```

Procedural blocks: If-then and Case statements

Can you see the difference in the multiplexors?

Will one be purely combinational?

Will one generate latches?

```
reg[1:0]

always @ (Sel, A, B, C, D)
begin
   if (Sel == 2'd0) Y = A;
   else if (Sel == 2'd1) Y = B;
   else if (Sel == 2'd2) Y = C;
   else if (Sel == 2'd3) Y = D;
end

     ***** Versus ******

always @ (Sel, A, B, C, D)
begin
   if (Sel == 2'd0) Y = A;
   else if (Sel == 2'd1) Y = B;
   else if (Sel == 2'd2) Y = C;
   else Y = D;
end
```

Procedural blocks: If-then and Case statements

Case statement example

```
Module decoder_3_8(
   input[2:0] I,
   output[7:0] Y
);
reg [7:0] Y;
always @(I)
begin
   case(I)
      3'd0: Y = 8'd1;
      3'd1: Y = 8'd2;
      3'd2: Y = 8'd4;
      3'd3: Y = 8'd8;
      3'd4: Y = 8'd16;
      3'd5: Y = 8'd32;
      3'd6: Y = 8'd64;
      3'd7: Y = 8'd128;
      default: 8'd0;
   endcase
end;
```

Modules

Modules are the principle design entity in Verilog. The keyword “module” is followed by the module name and the port list.

The port list names and specifies all port signals as inputs, outputs, or inout (bi-directional). Locally declared wires and regs are typically declared immediately following the module/port statement.

Combinational assignments are made using the assign statement. Sequential assignments are made inside an always block. A module can have any number of assign statements and always blocks.

After a module is declared, it can be used instantiated (i.e., used) in other modules by including the module name and port connection list. “Named port association” is recommended as shown in the example:

“.modulepinname(localname)”

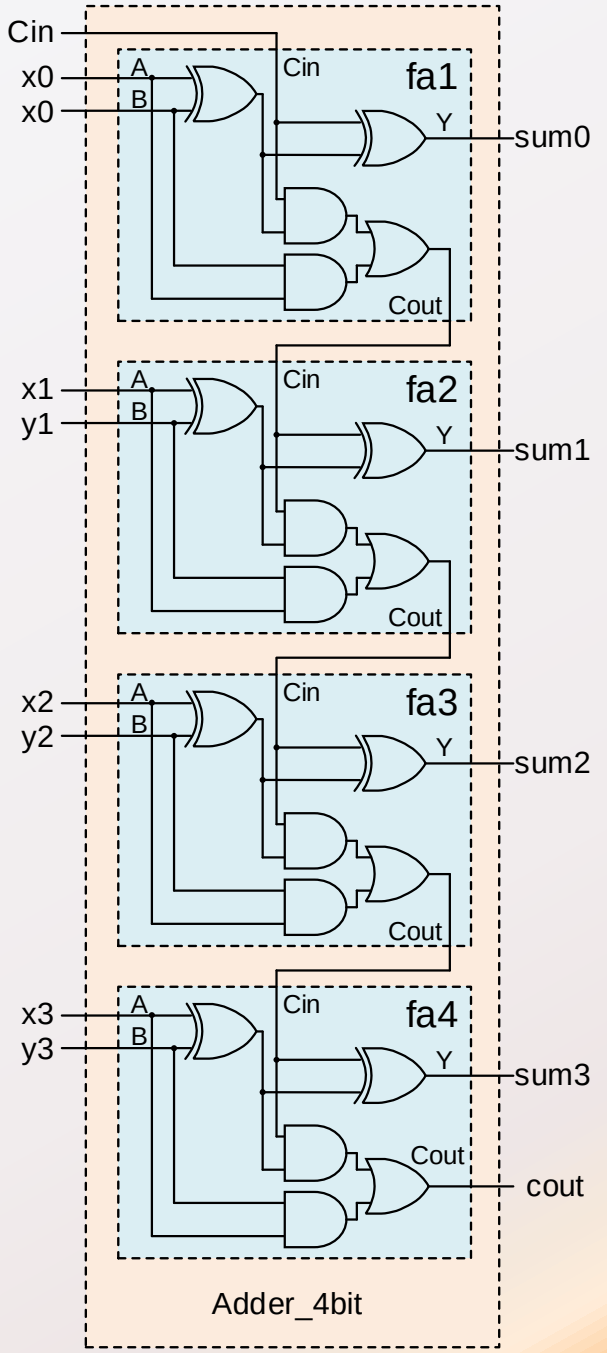
```
module simple(
    input A, B;
    output X, Y;
);

wire Z = 1b’1;

assign X = A & Z;
assign Y = A & B;

end module
```

```
module next(
    input C;
    output K;
);

wire D = 1b’1;
wire F,G;

simple(.A(C),.B(D),.F(X),.G(Y))
assign K = F & G;
end module
```

Structural Verilog: Using modules as building blocks

```
module full_adder(
    input a,b,cin;
    output y,cout
);
    wire s1,c1,c2;
    xor(s1,a,b);
    xor(y,cin,s1);
    and(c1,s1,cin);
    and(c2,a,b);
    or(cout,c1,c2);
endmodule

module adder_4bit(
    input [3:0] x,y;
    input cin;
    output[3:0] sum;
    output cout
);
    wire c1,c2,c3;
    full_adder fa1(.a(x[0]),.b(y[0]),.cin(cin),.y(sum[0]),.cout(c1));
    full_adder fa2(.a(x[1]),.b(y[1]),.cin(c1),.y(sum[1]),.cout(c2));
    full_adder fa3(.a(x[2]),.b(y[2]),.cin(c2),.y(sum[2]),.cout(c3));
    full_adder fa4(.a(x[3]),.b(y[3]),.cin(c3),.y(sum[3]),.cout(cout));
endmodule
```

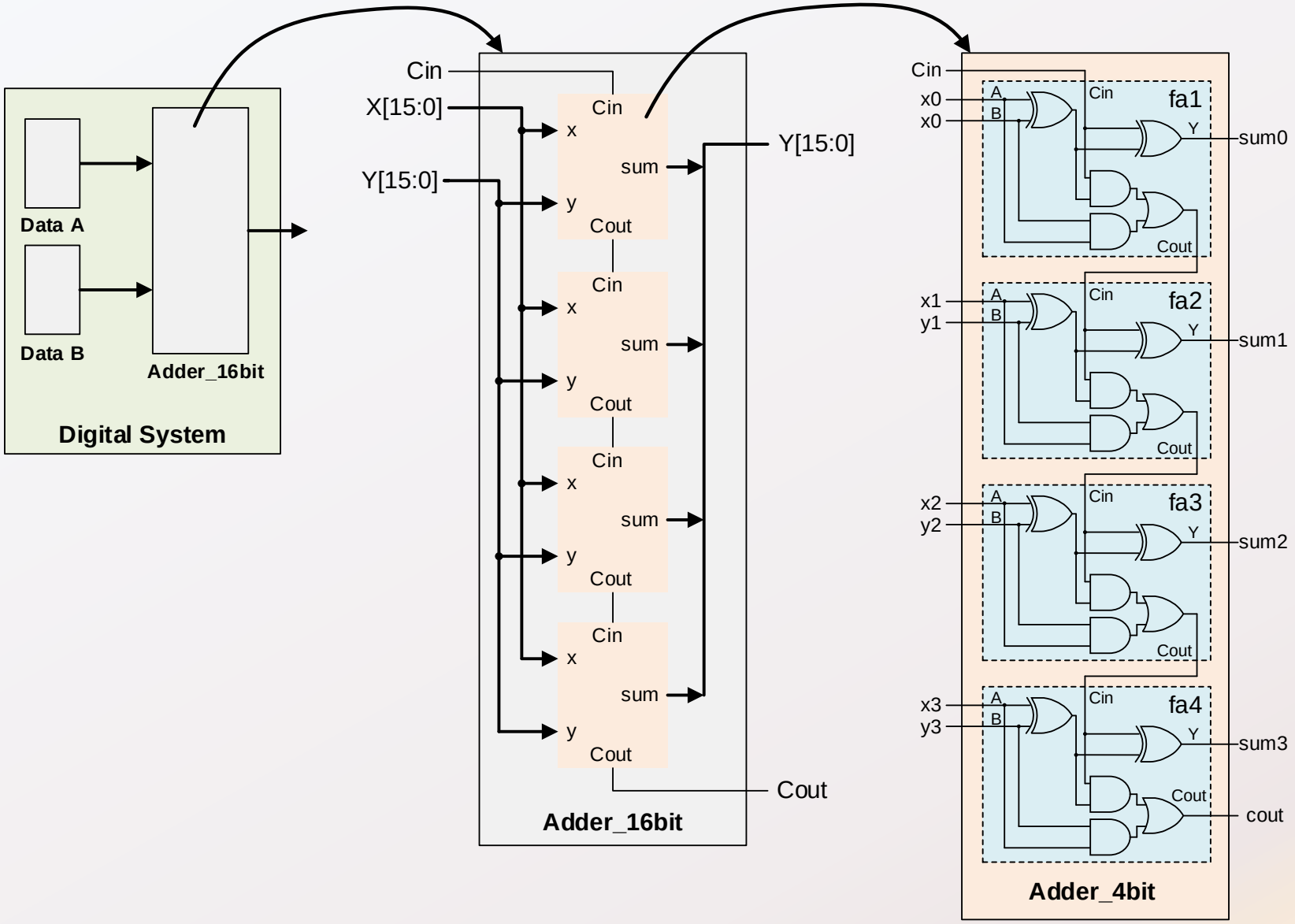
This example uses the Verilog “built in” logic modules of and, or, and xor. These module instantiations use “positional association”, which means the order of signals must match in the module and in the instantiation – they are matched left to right. In this case, the format is output name, followed by inputs.



Hierarchical design

Larger digital systems use hierarchical design, with a top level module that instantiates lower level modules.

Each module at every level is designed and tested independently. Leaf modules typically use behavioral design, while higher levels typically use structural design.



Multiplexor Examples

```
module mux_4_4(
   input [3:0] I0,I1,I2,I3,
   input [1:0] Sel,
   output [3:0] Y
);
```

Using assign statment

```
assign Y=(Sel==2'd0) ? I0 :
         ((Sel==2'd1) ? I1 :
         ((Sel==2'd2) ? I2 : I3));
```

Using a Case statement

```
reg [1:0] Y;

always @ (Sel, I0, I1, I2, I3) begin
   case (Sel)
      2'd0: Y = I0;
      2'd1: Y = I1;
      2'd2: Y = I2;
      2'd3: Y = I3;
      default: Y = 2'd0;
    endcase
end
```

Using if statement

```
reg [1:0] Y;

always @ (Sel, I0, I1, I2, I3) begin
   if       (Sel == 2'd0) Y = I0;
   else if (Sel == 2'd1) Y = I1;
   else if (Sel == 2'd2) Y = I2;
   else                  Y = I3;
end
```

ALU Example

```
module alu(
   input [7:0] A,B,
   input [3:0] Sel,
   output [7:0] Y
);

reg [7:0] Y;

always @ (Sel, A, B) begin
 case (Sel)
      3'd0: Y = A + B;
      3'd1: Y = A - B;
      3'd2: Y = A + 1;
      3'd3: Y = A - 1;
      3'd4: Y = A | B;
      3'd5: Y = A & B;
      3'd6: Y = A ^ B;
      3'd7: Y = ~A;
      default: Y = 7'd0;
   endcase
end
```

Flip-Flops and Latch

```
module dlatch(
  input gate, rst, D;
  output Q
);

always @ (gate or rst or D)
   if (rst == 1) Q <= 1'b0;
   else if (gate) Q <= D;

endmodule
```

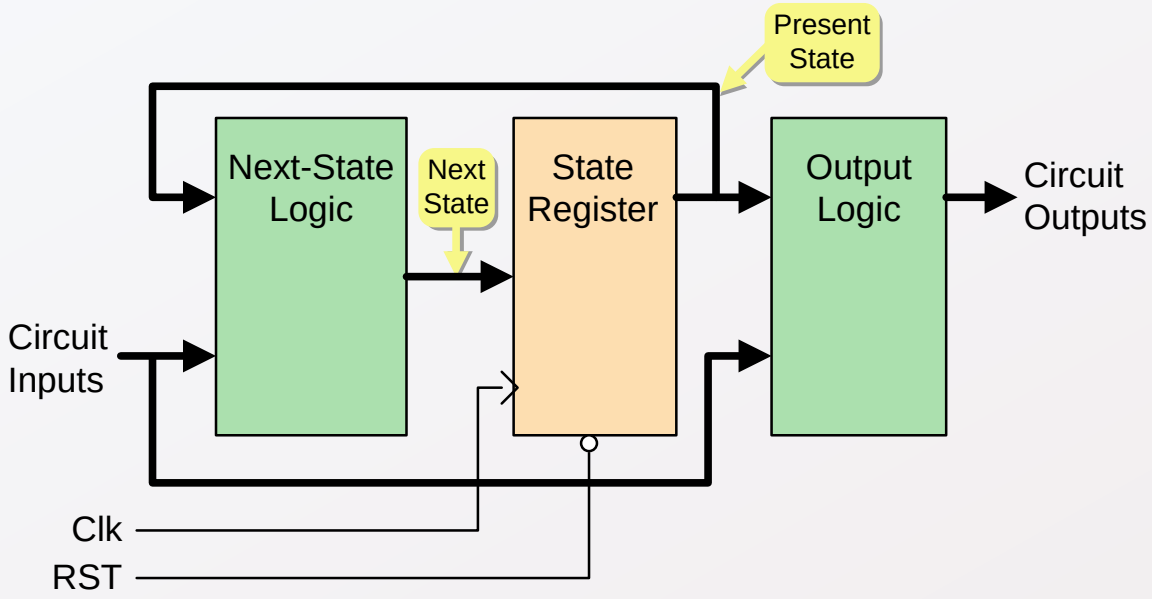
D Flip-flop
Asynch reset

```
module dff(
  input clk, rst, D;
  output Q
);

always @ (posedge(clk), posedge(rst))
begin
   if (rst == 1) Q <= 1'b0;
   else Q <= D;
end

endmodule
```

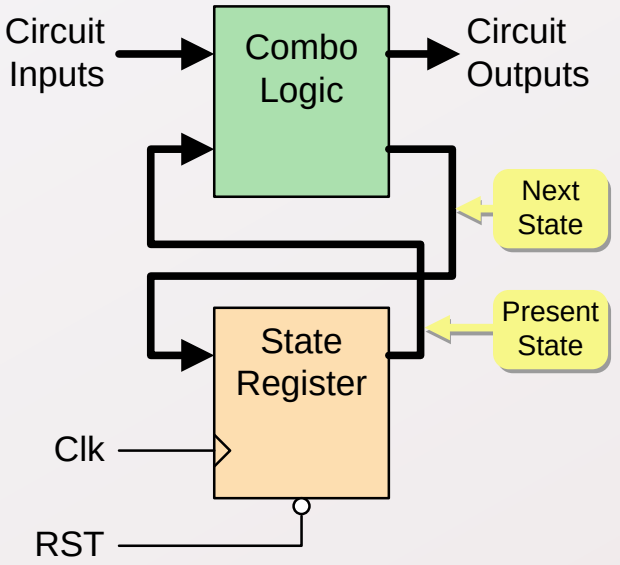
D Flip-flop
Synch reset

```
module dff(
  input clk, rst, D;
  output Q
);

always @ (posedge(clk))
begin
   if (rst == 1) Q <= 1'b0;
   else Q <= D;
end

endmodule
```

State Machine Models

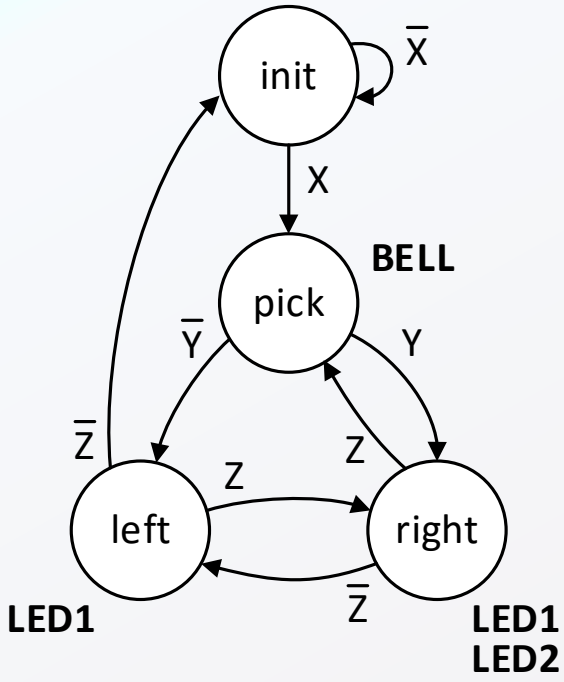


Mealy Model State Machine



Combined Model State Machine

State Machines

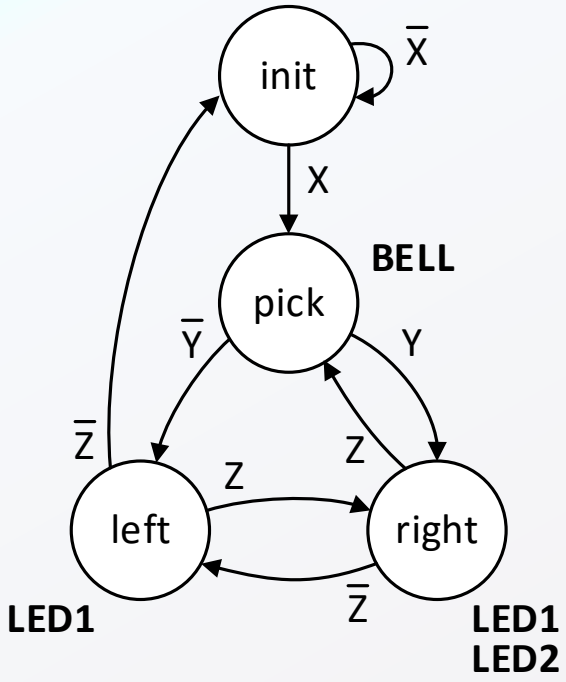


This is a common (and good!) way to code state machines:

1) Choose state codes, define state localparams, and define ps and ns busses; 2) Use two always blocks – one for next-state logic (combinational), and one for the state register; 3) assign outputs.

```
module fsm1 (
   input x, y, z, clk, rst;
   output bell, led1, led2;
);

localparam init = 2'b00; // define state identifiers
localparam pick = 2'b01;
localparam left = 2'b10;
localparam right = 2'b11;

Reg [1:0] ps, ns          // define ps, ns bus signals

always @(ps, x, y, z)
   case (ps)              // One always block used to
                          // define next-state logic
   endcase;               // (see next page)
end

always @(posedge(clk),posedge(rst))
  if (rst) ps <=0;
  else ps <= ns;         // 2nd always block used to
end                      // define state register

assign bell = (ps == pick);
assign led1 = (ps == left || ps == right);
assign led2 = (ps == right);

end module
```

State Machine (con't)



Case statement for FSM on previous slide

```
case (ps)
   init: begin
      if (x == 1) then ns = pick;
      else ns = init;
   pick:
      if (y == 1) then ns = right;
      else ns = left;
   left:
      if (z == 1) then ns = right;
      else ns = init;
   right:
      if (z == 1) then ns = pick;
      else ns = left;
   default:
      ns = init;
endcase
```

Testbench

A testbench is a separate Verilog source file that is written specifically to test other Verilog modules. It uses Verilog structures, syntax, and rules, plus a few extra items that are intended specifically to drive the simulator, and not to define a circuit.

A testbench is a module. It starts with a module declaration like any other Verilog source file (it is typical to name the module after the module being tested. We're testing the 4bit adder presented earlier, so we added "_tb" to the original module's name). The testbench module has no inputs or outputs – it doesn't need them. Instead, it will connect to the inputs and outputs of the module being tested.

Next, we instantiate the module to be tested by proving the module name (adder_4bit) and giving it an instance name (here we use "dut" for "device under test"), and connect the ports. It is typical to use the same port names in the testbench module as were used in the dut.

The remainder of the testbench source file drives inputs into the simulator.

```
module adder_4bit(
    input [3:0] x,y;
    input cin;
    output[3:0] sum;
    output cout
);
```

4-bit adder module statement from earlier example (reproduced for reference)

```
module adder_4bit_tb;

wire [3:0] x,y,sum;
wire cin,cout;

adder_4bit dut (
   .x(x),.y(y),.cin(cin),
   .sum(sum),.cout(cout)
);
```

Testbench con’t

A typical testbench uses an “initial block” to drive input signals with changing logic values over time. Initial blocks are only used in testbenches – they are not synthesizable.

The characters “ #n “ (where n is a number) that appear prior to inputs specify simulation delays before signals take the assigned values. Delays are specified as “n” time scale units that default to some number – you can optionally set the time scale by including “ ’timescale 1ns/1ps” in your file (this sets the time step to 1ns, and the precision to 1ps).

It is often more efficient to use a loop in the testbench to automatically create larger numbers of test patterns. For example, to create every possible combination of the 4-bit “x” adder inputs, the loop shown to the right is an efficient method.

```
initial
begin
     x = 0; y = 0; cin = 0;
     #10 x = 5;
     #10 cin = 1;
     #10 y = 3;
end;
```

```
integer k;

initial

begin
   for (k=0; k<16; k=k+1)
     #10 x = k;
   #10 finish;
end;
```

Miscellaneous

Memory Arrays: Verilog models memory as an array of regs. Memories are accessed by providing an array access. Examples:

```
reg[7:0]  mem1[0:255]   // a 256 byte memory with 8-bit bytes
reg[31:0] mem2[0:1023]  //1K memory with 32-bit words

temp = mem1[20]         // load temp from mem1 location 20
```

Other types: “Integer”, “time”, and “real” are legal Verilog variable types that are used in behavioral modeling and in simulation, but rarely in sauce code intended for synthesis.