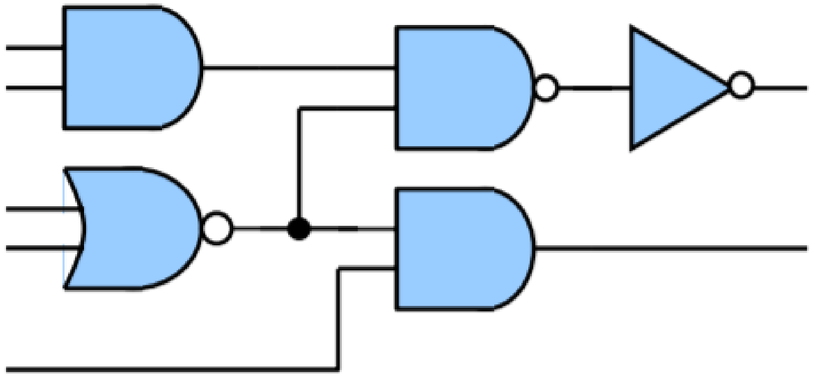
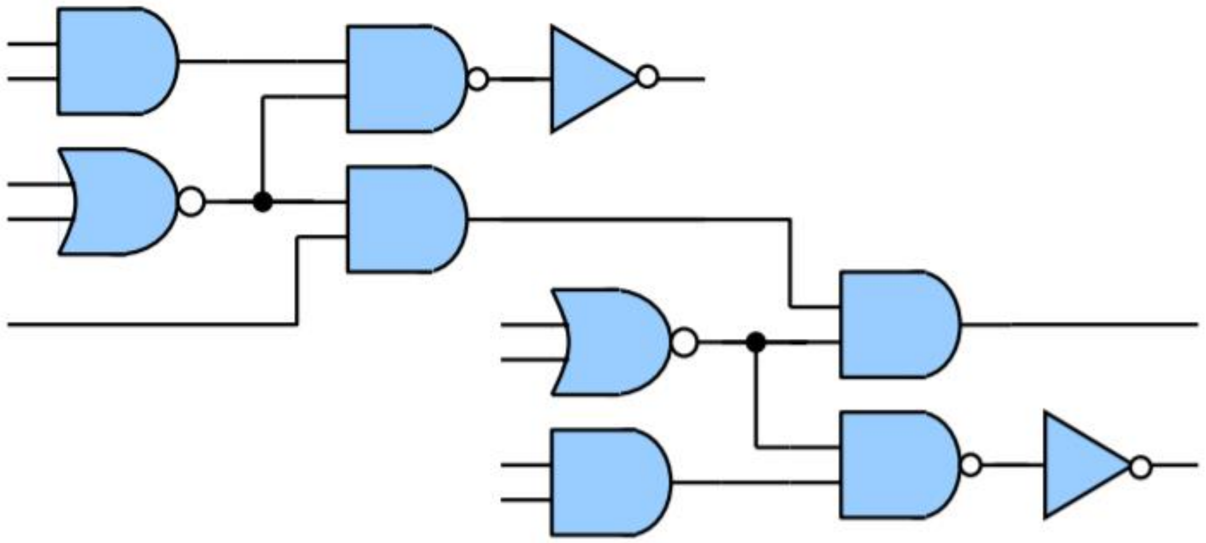
Digital Design Using Verilog

Hardware Description Languages



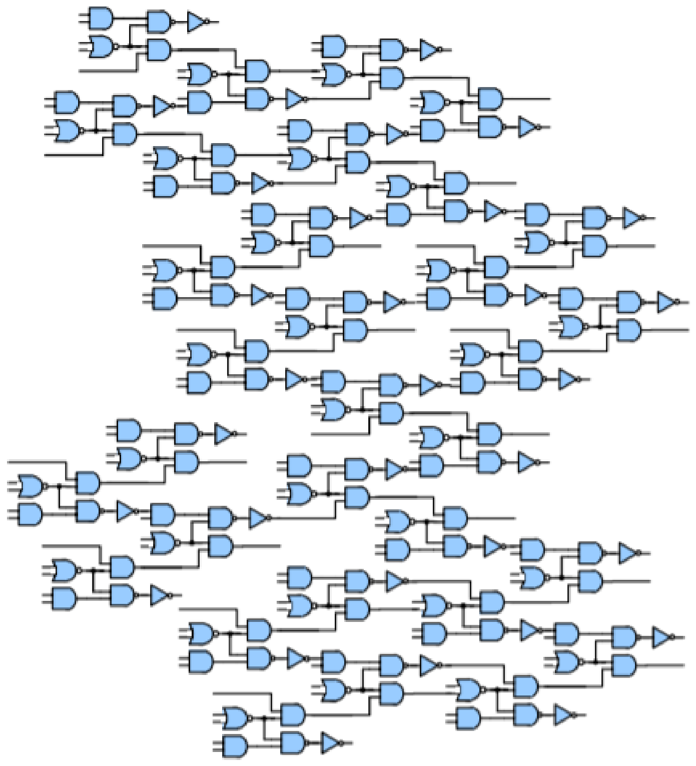
In the beginning designs involved just a few gates, and thus it was possible to verify these circuits on paper or with breadboards

Hardware Description Languages



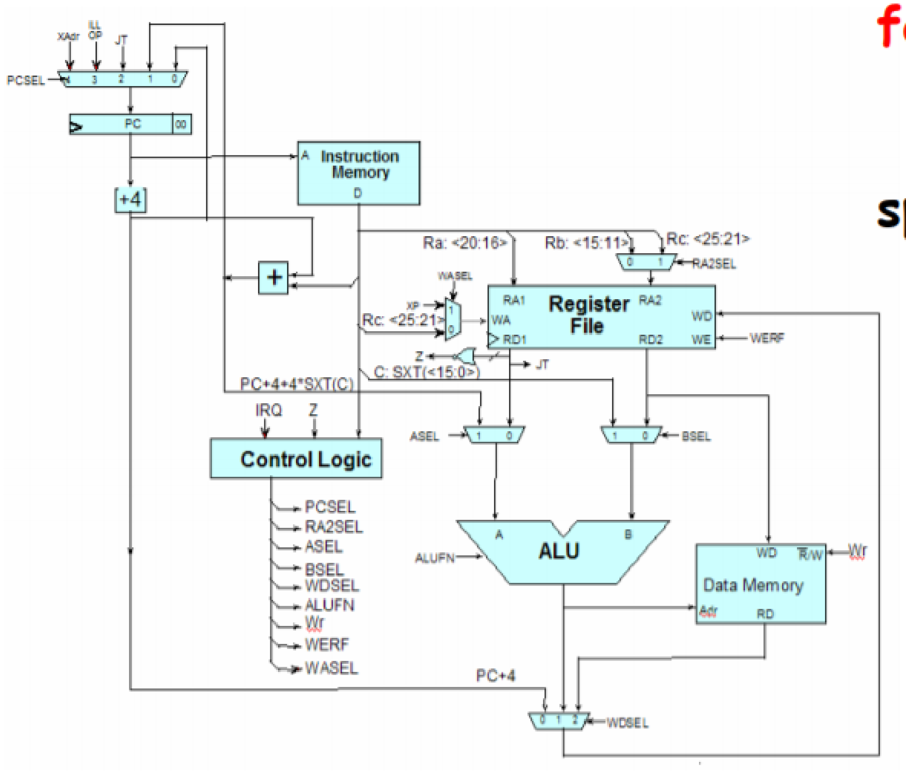
As designs grew larger and more complex, designers began using gate-level models described in a Hardware Description Language to help with verification before fabrication

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

HDLs Overview

- A Hardware Description Language (HDL) **is a language used to describe a digital system**, for example, a computer or a component of a computer.
- A digital system can be described at several levels
  - **Switch level**: wires, resistors and transistors
  - **Gate level**: logical gates and flip flops
  - **Register Transfer Level (RTL):** registers and the transfers of information between registers
- Two Major HDLs in Industry
  - VHDL
  - **Verilog**

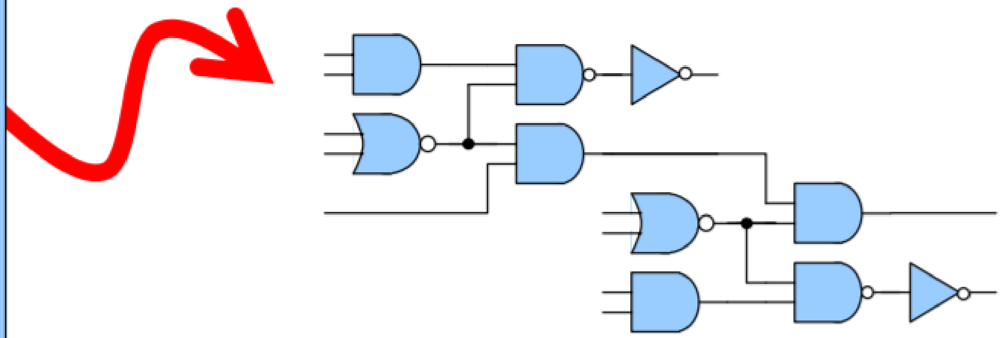
We will use Verilog

- **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

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

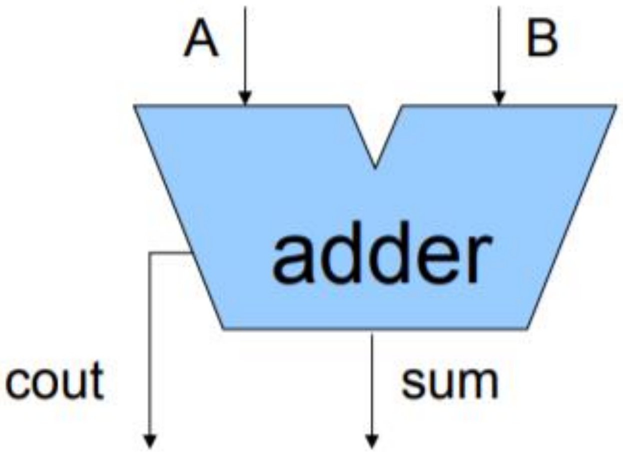
```
module foo(clk,xi,yi,done);
  input [15:0] xi,yi;
  output done;

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



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



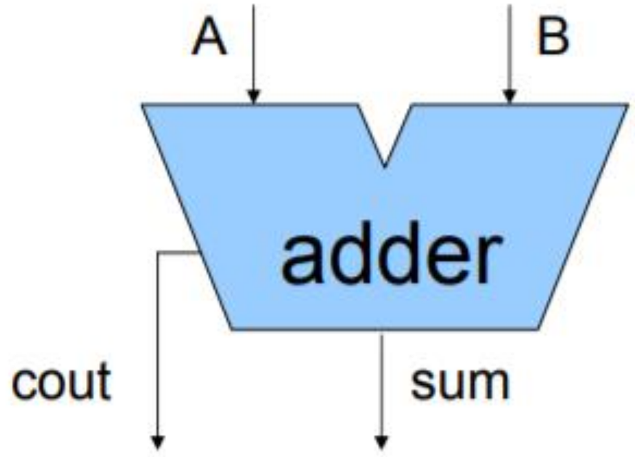
```
module adder( A, B, cout, sum );
  input  [3:0] A, B;
  output       cout;
  output [3:0] sum;

  // HDL modeling of
  // adder functionality

endmodule
```

Don't forget the semicolon!

Modeling with Verilog

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

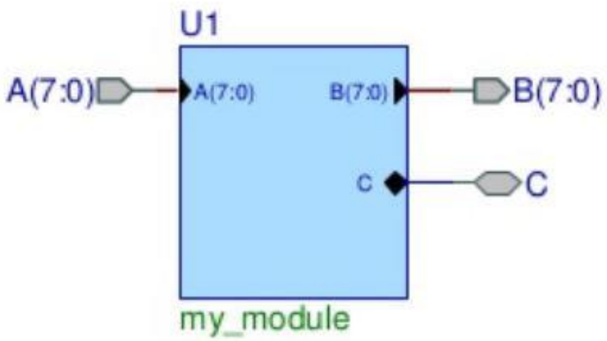
- Must specify direction and bitwidth for each port
- Verilog-2001 introduced a succinct ANSI C style portlist



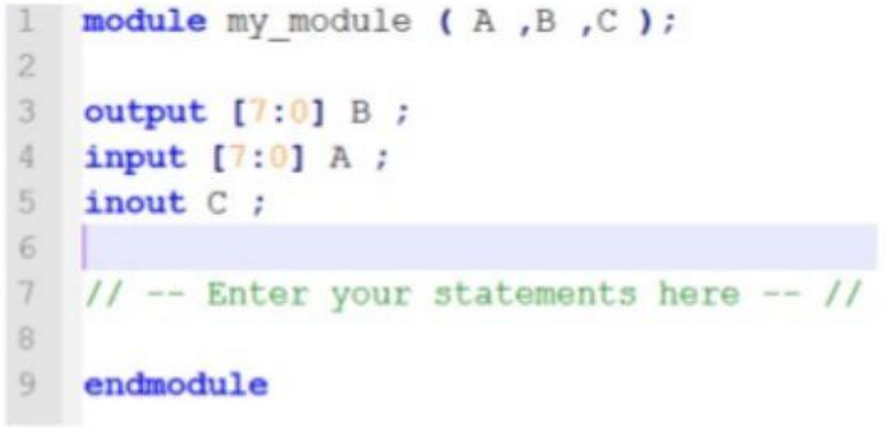
```
module adder( input  [3:0] A, B,
              output       cout,
              output [3:0] sum );

  // HDL modeling of 4 bit
  // adder functionality

endmodule
```

Ports

- There are three different port types:
  - Input
  - Output
  - Inout

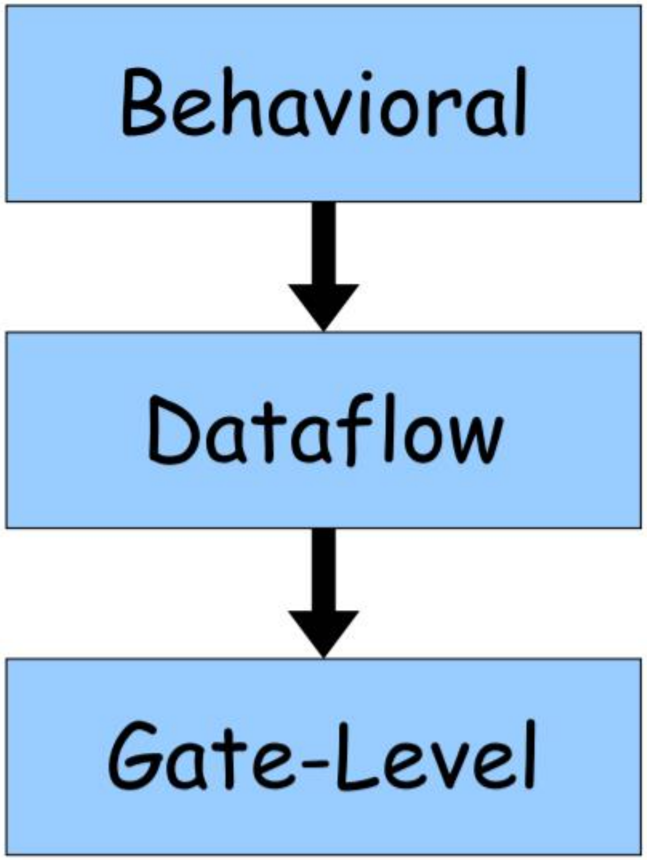


- Port definition:
  - <port type> <port width> <port name>



```
module my_module ( A ,B ,C );

output [7:0] B ;
input [7:0] A ;
inout C ;

// -- Enter your statements here -- //

endmodule
```

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

Common Abstraction Levels Summary

- **Gate level [Net list of gates]**
  - This is implemented using logic gates. The designer here must know the exact gate level diagram of the design.
- **Data flow level [Boolean function assigned to a net]**
  - This is implemented by specifying the data flow, i.e. registers used in the design.
- **Behavioral level**
  - This is highest level of abstraction. It is implemented with design algorithms. Here the designer necessarily not need to know the exact hardware implementation knowledge.
  - A sequential algorithm (quite similar to software) that determines the value(s) of variable(s)

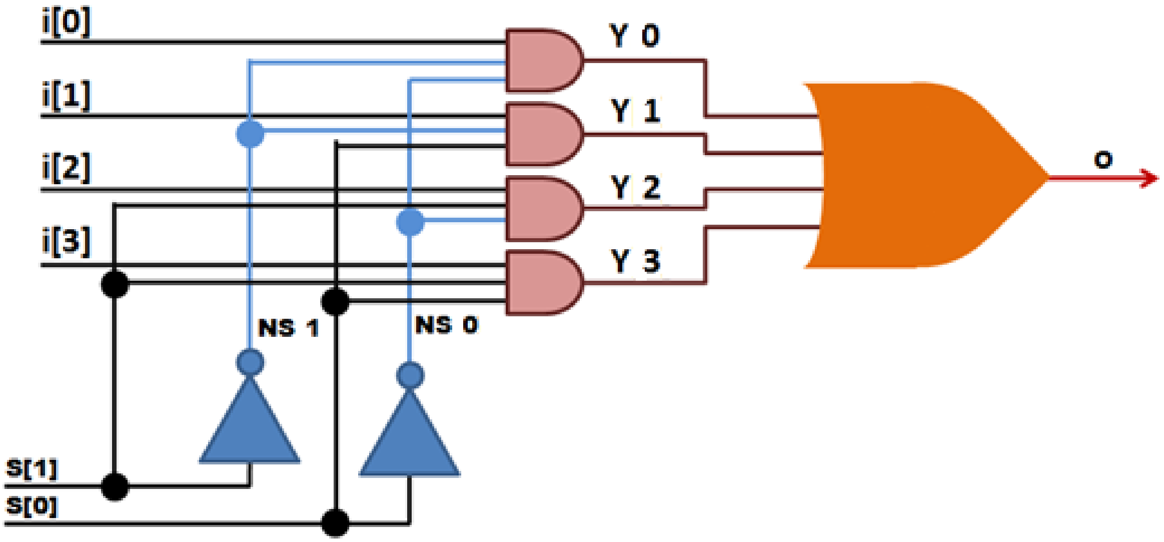
BUILDING GATE-LEVEL MODELS

Gate-Level : 2-input AND

```
module AND2 (in1, in2, out);
    input in1; input in2;
    output out;
    and u1 (out, in1, in2);
 endmodule
```

Gate-Level : 4-input Multiplexer

```
module Mux_4to1_gate(I,S,O);
input [3:0] I;
input [1:0] S;
output O;

wire NS0, NS1;
wire Y0, Y1, Y2, Y3;
not N1(NS0, S[0]);
not N2(NS1, S[1]);
and A1(Y0, I[0], NS1, NS0);
and A2(Y1, I[1], NS1, S[0]);
and A3(Y2, I[2], S[1], NS0);
and A4(Y3, I[3], S[1], S[0]);
or O1(O, Y0, Y1, Y2, Y3);
endmodule
```



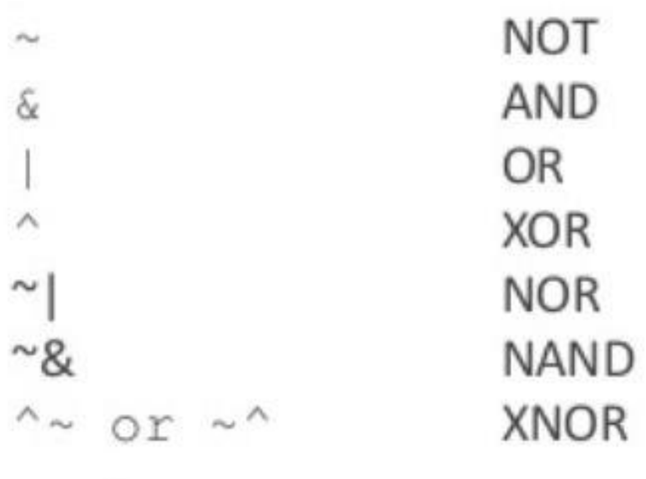
Physical data type: Variables

- Two basic families of data types for variables: Nets and Registers
- Net variables e.g. **wire**
  - Memory less
  - Variable used simply to connect components together
  - Usually corresponds to a wire in the circuit
- Register variable e.g. **reg**
  - Variable used to store data as part of a **behavioural description**
  - Like variable in ordinary procedural languages
- Note
  - The **reg** variables store the last value that was procedurally assigned to them where as the **wire** variables represent physical connections between structural entities such as gates.

BUILDING DATA FLOW MODELS

Operators

- **Bitwise operators**



~	NOT
&	AND
|	OR
^	XOR
~|	NOR
~&	NAND
^~ or ~^	XNOR

- **Logical operators**

    !,&&,||,==,!=,>=,<=,>,<

Numbers

- Numbers are specified using the following form

<size><base format><number>

Size of the number in bits

‘b (binary)
‘o (octal)
‘h (hex)

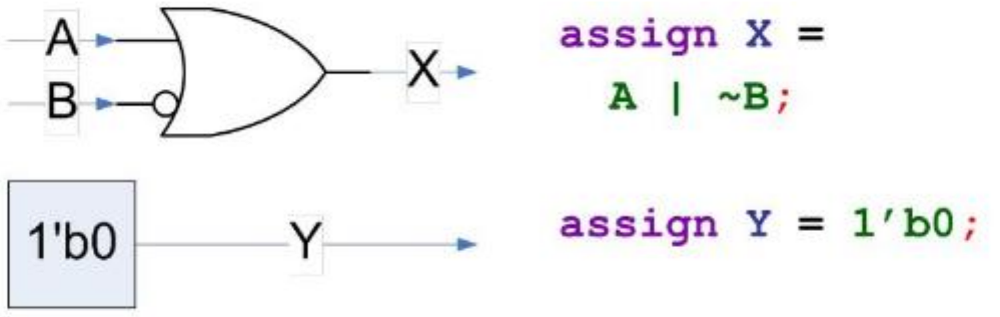
- Example:
  - X=347 //decimal number
  - X=4’b101 // 4-bit binary number 0101
  - X=16’h87f7 16-bit hex number 87f7

Continuous Assignment

- Continuous statement is used to model **combinational logic.**
- A continuous assignment statement is declared as follows:

Assign <net_name>= expression;

- Assign corresponds to a connection
- Target is never a **reg** variable.
- Example:

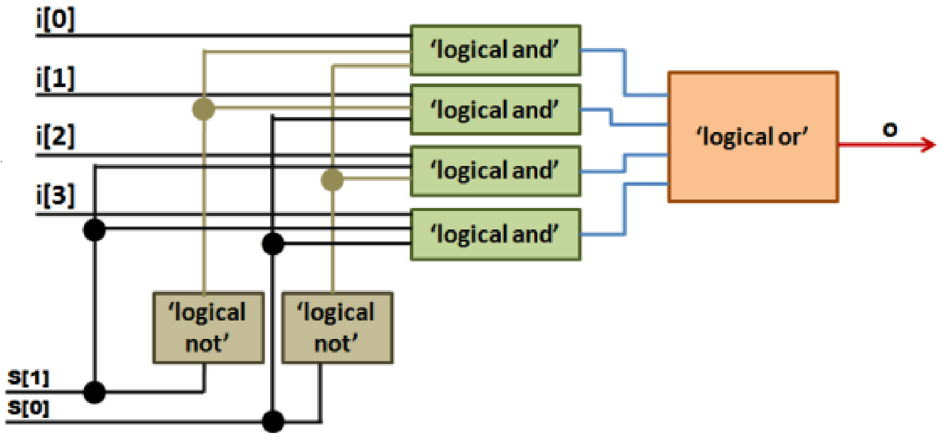


```
assign X =
  A | ~B;

assign Y = 1'b0;
```

Data Flow: 2-input AND

```
module AND2 (in1, in2, out);
    input in1; input in2;
    output out;
    assign out = in1 & in2;
 endmodule
```

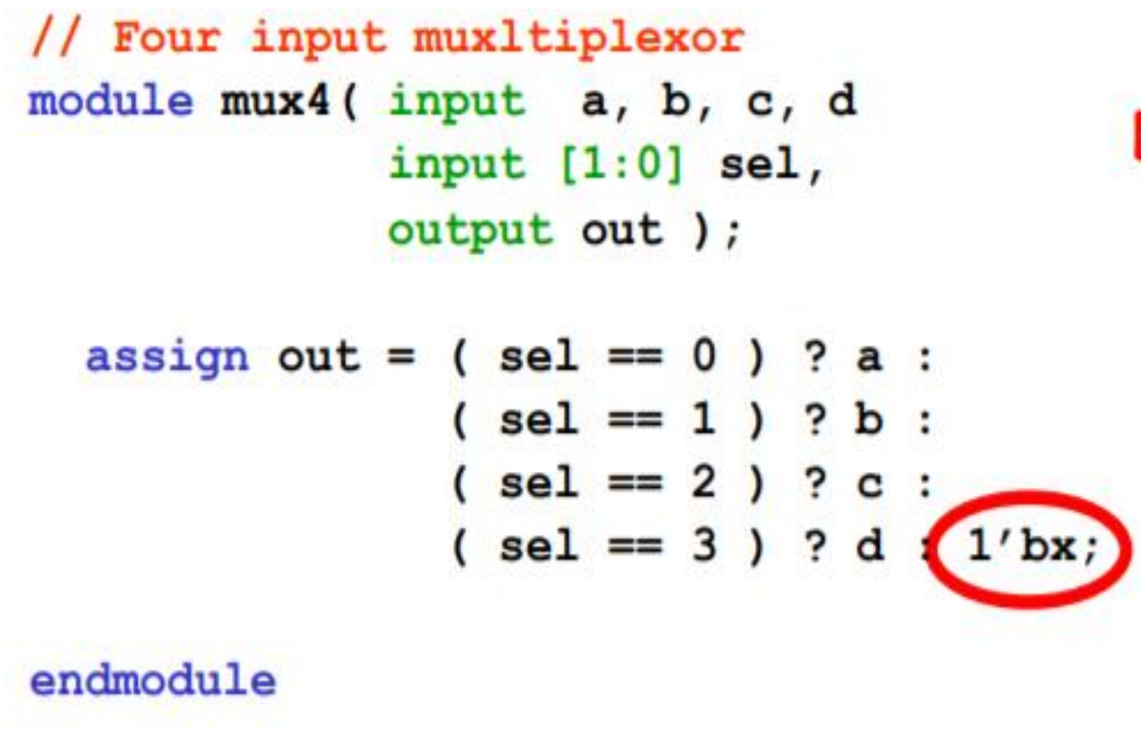
Dataflow : 4-input Multiplexer



```
module mux4 (I,S,O)
    input [3:0]I;
     input [1:0] S;
    output O;
    assign O = (~ S[1] & ~ S[0] & I[0])| (~ S[1] & S[0]
      & I[1])| (S[1] & ~ S[0] & I[2]) | (S[1] & S[0] &
      I[3]);
     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

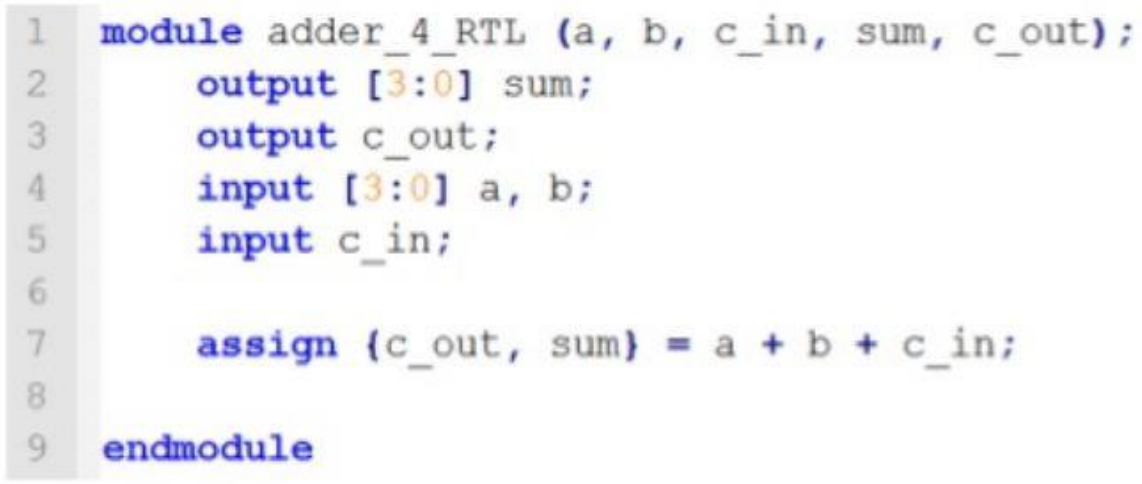


```
// Four input muxltiplexor
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

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

endmodule
```

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

Continuous Assignment

- Verilog code for a 4 bit adder:



```
module adder_4_RTL (a, b, c_in, sum, c_out);
    output [3:0] sum;
    output c_out;
    input [3:0] a, b;
    input c_in;

    assign {c_out, sum} = a + b + c_in;

endmodule
```

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

BUILDING BEHAVIORAL MODELS

Structured Procedures

- Two basic structured procedure statements

  always

  initial

  - All behavioral statements appear only inside these blocks
  - Each always or initial block has a separate activity flow (multithreading, concurrency)
  - Start from simulation time 0
  - No nesting

Procedural Assignment
always

- Used for modeling sequential circuits.
- A procedural assignment statement is declared as follows:

```
always @(event list) begin
<reg_name> = expression;
end
```

- The assignment will be performed whenever one of the events in “event_list” occurs.

Procedural Assignment
initial

- Starts at time 0
- Executes only once during a simulation
- Multiple initial blocks, execute in parallel
  - All start at time 0
  - Each finishes independently
- Syntax:

```
initial
begin
  // behavioral statements
end
```

Procedural Assignment initial [Example]

```
reg A, B, C, D;

  initial
    A= 1'b0;

  initial
  begin
    #5  B=1'b0;
    #25 C=1'b1;
  end

initial
 #10 D= 1'b1;
```

Time	Statement executed
0	A=1’b0
5	B=1’b0
10	D=1’b1
30	C=1’b1

Behavioral Modeling Statements: Conditional Statements

- Just the same as **if-else** in C
- Syntax:

```
if (<expression>) true_statement;

if (<expression>) true_statement;
    else  false_statement;

if (<expression>) true_statement1;
    else if (<expression>) true_statement2;
    else if (<expression>) true_statement3;
    else default_statement;
```

- True is 1 or non-zero
- False is 0 or ambiguous (x or z)
- More than one statement: begin end

Behavioral Modeling Statements: Case Statement

- Similar to **switch-case** statement in C

- Syntax:

```
case (<expression>)
  alternative1: statement1;
  alternative2: statement2;
    ...
  default: default_statement; // optional
endcase
```

- Notes:
  - <expression> is compared to the alternatives in the order specified.
  - Default statement is optional

Behavioral Modeling Statements: Loops

- Loops in Verilog
  - while, for, repeat, forever

- **The while loop** syntax:

```
while (<expression>)
    statement;
```

- Example
- **initial begin**

```
  reg [3:0] i, output;
  i = 0;
  while (i <= 15) begin
    output = i;
    #10 i = i + 1;
  end
  end
```

Behavioral Modeling Statements: Loops

- **The for loop**
  - Similar to C
  - Syntax:

```
    for( init_expr; cond_expr; change_expr)
      statement
```

- **Example:**

```
initial begin
reg [3:0] i, output;
for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end
end
```

Behavioral Modeling Statements: Loops

- **The repeat loop**
  - Syntax:

    repeat (expression) statement;
- The **repeat** instruction executes a given statement a fixed number of times. The number of executions is set by the expression, which follows the repeat keyword.
- **Example:**

```
initial begin
    repeat (10) a = a + ~b;
  end
```

Behavioral Modeling Statements: Loops

- **The forever loop**
  - Syntax:

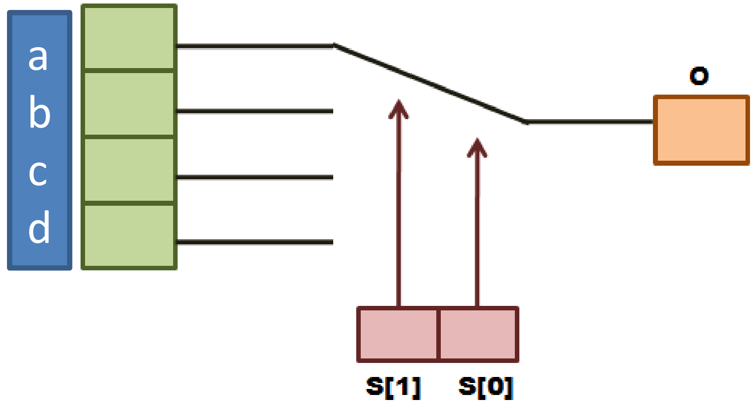
    forever statement;
- The forever instruction continuously repeats the statement that follows it.
- Example:

```
initial begin
#1 clk = 0;
forever begin
#5 clk = ! clk;
end
end
```

Behavioral: 2-input AND

```
module AND2 (in1, in2, out);
    input in1, in2;
    output out;
    reg out;
    always @ (in1 or in2)
    out = in1 & in2;
endmodule
```

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

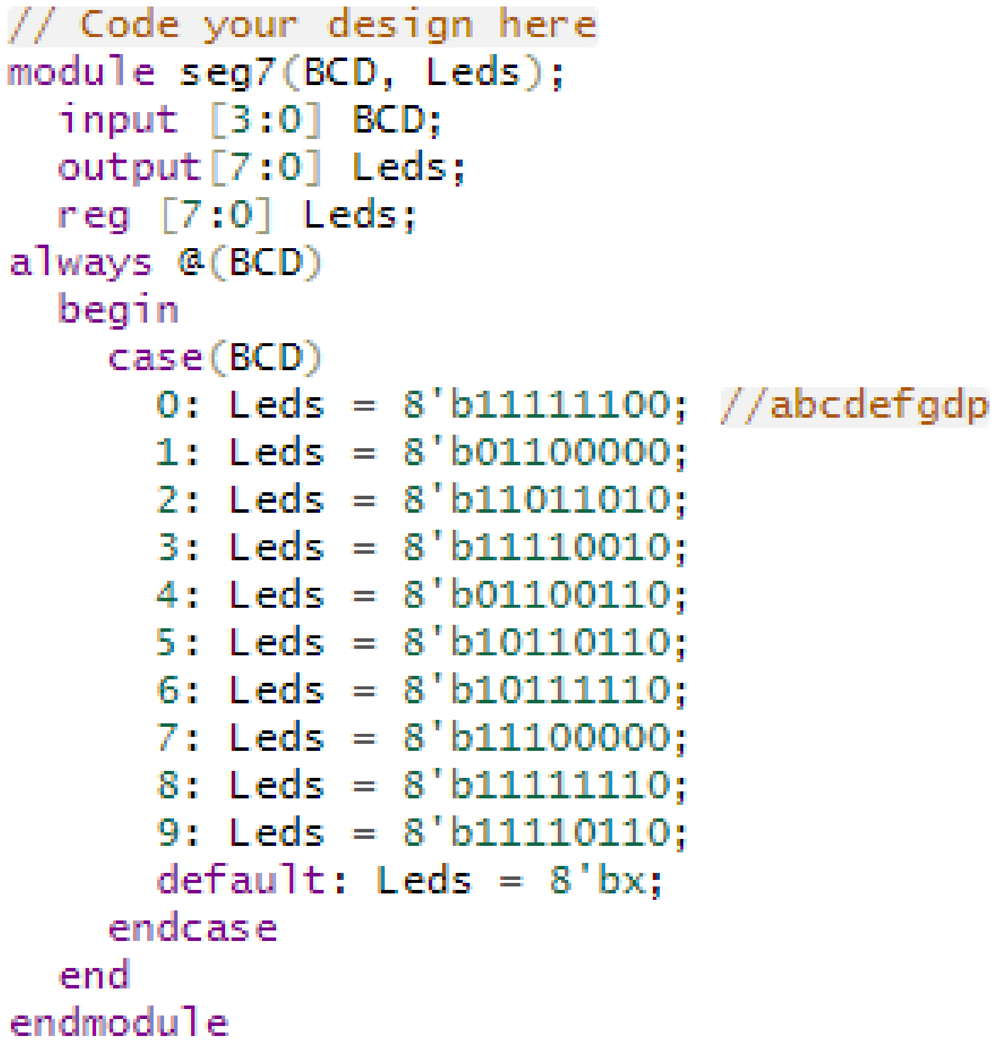
Behavioral: 4-input Multiplexer

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

  reg out;

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

endmodule
```

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

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

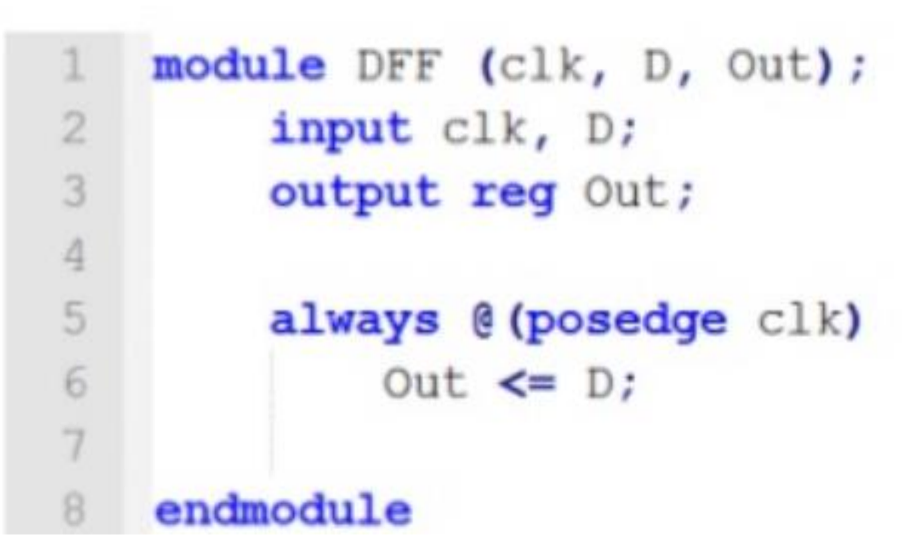
BCD to 7-Segment Display



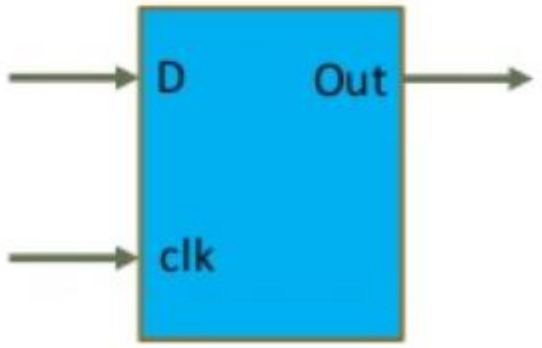
```
// Code your design here
module seg7(BCD, Leds);
  input [3:0] BCD;
  output[7:0] Leds;
  reg [7:0] Leds;
always @(BCD)
  begin
    case(BCD)
      0: Leds = 8'b11111100; //abcdefgdp
      1: Leds = 8'b01100000;
      2: Leds = 8'b11011010;
      3: Leds = 8'b11110010;
      4: Leds = 8'b01100110;
      5: Leds = 8'b10110110;
      6: Leds = 8'b10111110;
      7: Leds = 8'b11100000;
      8: Leds = 8'b11111110;
      9: Leds = 8'b11110110;
      default: Leds = 8'bx;
    endcase
  end
endmodule
```

Procedural Assignment D-FlipFlop

- Example D-FF



```verilog
module DFF (clk, D, Out);
    input clk, D;
    output reg Out;

    always @(posedge clk)
        Out <= D;

endmodule
```



Mixed-Level Representation

- Any Verilog representation must start with the keyword "**module**"
  - The Structural (gate-level) Description
    - Basically net list.
  - The Data flow Description
  - The Behavioral Description
- **Mixed-Level Representation**
  - A combination of behavior, and/or data flow, and/or structure.

A Data Flow 1-bit Adder

```
//Data Flow of 1-bit Full-Adder
module fulladd ( in1, in2, carryin, sum,
  carryout) ;
   input in1, in2, carryin;
   output sum, carryout;
   assign {carryout, sum} = in1 + in2 +
     carryin;
endmodule
```

Mixed-Level 4-bit Adder

```
module adder4 (in1, in2, sum, cout, zero);
   input [3:0] in1, in2;
   output [3:0] sum;
   output cout, zero;
    reg zero;
    wire c1,c2,c3;
     fulladd u1 (in1[0], in2[0], 0, sum[0], c1);
     fulladd u2 (in1[1], in2[1], c1, sum[1], c2);
     fulladd u3 (in1[2], in2[2], c2, sum[2], c3);
     fulladd u4 (in1[3], in2[3], c3, sum[3], cout);
     always @ (*)
       if ( sum == 0 && cout == 0) zero = 1; else zero = 0;
endmodule
```

Summary

- *So to summaries when a circuit is very simple and require only few gates we use* ***gate level modelling****, when a* circuit can be defined on its functionality depending on its truth table *and a bit complex we use* ***dataflow*** *and when required at a highest level of abstraction* ***behavioral modelling*** *is used*

Verilog TestBench

- A module with no I/O ports.
- Usually uses behavioral constructs

  e.g. initial
- initial is executed only once.
- https://www.edaplayground.com/

TestBench Example 1
AND Gate

```
//TestBench
module test_and2; reg       i1, i2;
          wire     o;
 // Instantiation of the module(s) to be tested.
         AND2  u2 ( i1, i2, o);
 initial begin
          i1 = 0; i2 = 0;   #1  $display (“i1 =%b,   i2 =%b,    o =%b”,  i1, i2, o);
          i1 = 0;  i2 = 1;  #1  $display (“i1 = %b,  i2 =  %b , o = %b”, i1, i2, o);
          i1 = 1;  i2 = 0; #1  $display (“i1 = %b,  i2 =  %b , o = %b”, i1, i2, o);
          i1 = 1;  i2 = 1; #1  $display (“i1 = %b,  i2 =  %b , o = %b”, i1, i2, o);
end
endmodule
```

TestBench Example 2
4_1 MUX

```
`timescale 1ns/1ps   //Whatever times you mentioned in Verilog code  will be taken in ns.
module test_MUX();
  reg [3:0]I;
  reg [1:0]S;
  wire Y;
 mux4 u1(I,S,Y);
initial
    begin
I=5;  S=0;  #1 $display("OUPUT=%b",Y);
I=5;  S=1;  #1 $display("OUPUT=%b",Y);
I=5;  S=2;  #1 $display("OUPUT=%b",Y);
I=5;  S=3;  #1 $display("OUPUT=%b",Y);
    end
endmodule
```