Princess Sumaya University for Technology

Computer Organization and Architecture I

Introduction to Verilog Hardware Description Language

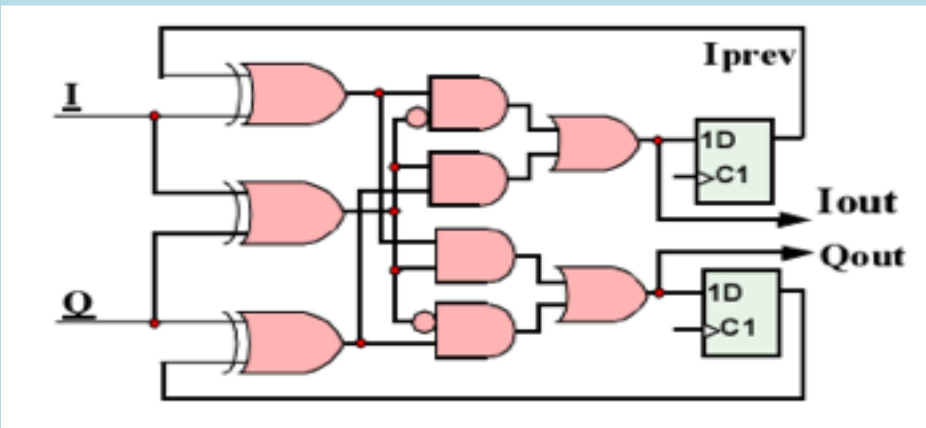
Dr. Esam Al-Qaralleh

Introduction

Purpose of HDL:

1. Describe the circuit in algorithmic level (like c) and in gate-level (e.g. And gate)
2. Simulation
3. Synthesis
4. Words are better than pictures

The best way to describe a circuit?



$$Iout=(\overline{I\oplus Q})(I\oplus Iprev + (I\oplus Q)(Q\oplus Qprev)$$
$$Qout=(\overline{I\oplus Q})(Q\oplus Qprev + (I\oplus Q)(I\oplus Iprev)$$

IQ \ (I,Q)prev	00	01	11	10
00	00	01	11	10
01	10	00	01	11
11	11	10	00	01
10	01	11	10	00

Iout,Qout

If both inputs are 1, change both outputs.

If one input is 1 change an output as follows:

- If the previous outputs are equal change the output with input 0;
- If the previous outputs are unequal change the output with input 1.

If both inputs are 0, change nothing.

Verilog Basics

helloWorld.v

```
module helloWorld ;
    initial
    begin
        $display ("Hello World!!!");
        $finish;
    end
endmodule
```

System calls.

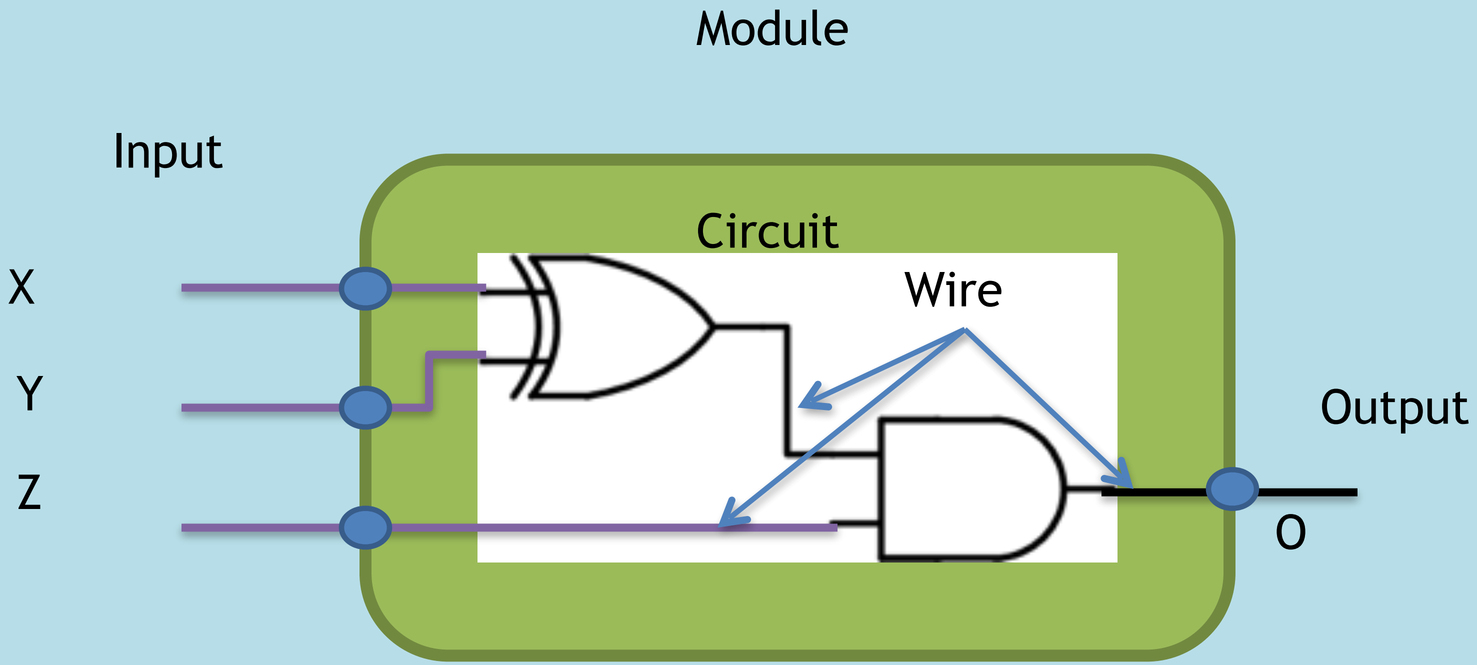
Modules are the unit building-blocks (components) Verilog uses to describe an entire hardware system. Modules are (for us) of three types: *behavioral, dataflow, gate-level*. We ignore the *switch-level* in this course.

This module is behavioral. Behavioral modules contain code in *procedural* blocks.

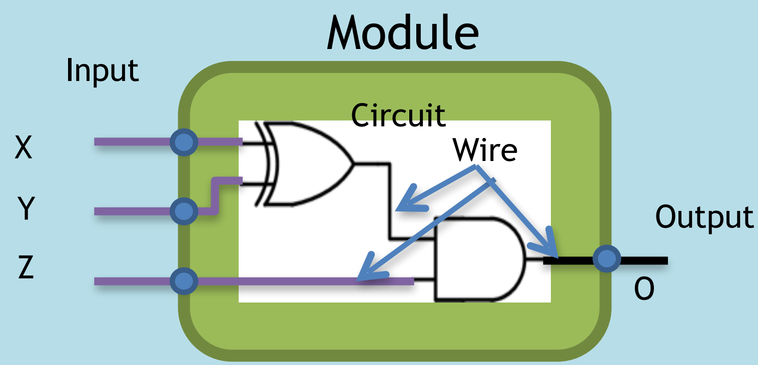
This is a *procedural* block.
There are two types of procedural blocks: *initial* and *always*.

More than one statement must be put in a *begin-end* group.

Module declaration



Module declaration



Module name

```
module  sample (X,Y,Z,O);

input X,Y,Z;
output O;

// Describe the circuit using logic symbols
assign O = (X^Y)&Z;

endmodule
```

Typical Module Components Diagram

Module name, Port list (optional, if there are ports)
Port declarations Parameter list
Declaration of variables (wires, reg, integer etc.)
Instantiation of inner (lower-level) modules
Structural statements (i.e., assign and gates)
Procedural blocks (i.e., always and initial blocks)
Tasks and functions
endmodule declaration

Lexicography

- Comments:

Two Types:

- // *Comment*
- /* ***These comments extend***
  over multiple lines. Good
  for commenting out code */

- Character Set:

0123456789ABCD..YZabcd...yz_$

Cannot start with a number or $

systemCalls.v

```
module systemCalls(clk);
  input clk;
  clockGenerator cg(clk);

  initial
    begin
    #25 $stop;
    #50 $finish;
    end

  initial
    begin
    $write("$write does not ");
    $write("add a new line\n");

    $display("$display does");
    $display("add a new line");

    $monitor("Clock = %d", cg.clk); end
endmodule
```

Compile with the clockGenerator.v module.

Suspends simulation – enters interactive mode.

Terminates simulation.

Similar output calls except $display adds a new line.

$monitor produces output each time a variable changes value.

Data Types

- Nets and Registers
- Vectors
- Integer, Real, and Time Register Data Types
- Arrays
- Memories
- Parameters
- Strings

Nets

- Used to represent connections between HW elements
  - Values continuously driven on nets
- Keyword: `wire`
  - Default: One-bit values
    - unless declared as vectors
  - Default value: `z`
    - For `trireg`, default is `x`
  - Examples
    - `wire a;`
    - `wire b, c;`
    - `wire d=1'b0;`

Registers

- Registers represent data storage elements
  - ❑ Retain value until next assignment
  - ❑ NOTE: this is not a hardware register or flipflop
  - ❑ Keyword: `reg`
  - ❑ Default value: `x`
  - ❑ Example:

```
reg reset;
initial
begin
    reset = 1'b1;
    #100 reset=1'b0;
end
```

Vectors

- Net and register data types can be declared as vectors (multiple bit widths)
- Syntax:
  - `wire/reg [msb_index : lsb_index] data_id;`
- Example

```
wire a;
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg clock;
reg [0:40] virtual_addr;
```

Vectors (cont'd)

- Consider

```
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg [0:40] virtual_addr;
```

- Access to bits or parts of a vector is possible:

```
busA[7]
bus[2:0] // three least-significant bits of bus
// bus[0:2] is illegal.
virtual_addr[0:1] /* two most-significant bits
                   * of virtual_addr
                   */
```

Integer, Real, and Time Register Data Types

- Integer
  - Keyword: `integer`
  - Very similar to a vector of `reg`
    - `integer` variables are signed numbers
    - `reg` vectors are unsigned numbers
  - Bit width: implementation-dependent (at least 32-bits)
    - Designer can also specify a width:

      ```
      integer [7:0] tmp;
      ```
  - Examples:

    ```
    integer counter;
    initial
        counter = -1;
    ```

Integer, Real, and Time Register Data Types (cont'd)

- Real
  - Keyword: `real`
  - Values:
    - Default value: 0
    - Decimal notation: `12.24`
    - Scientific notation: `3e6` (=3x10^6)
  - Cannot have range declaration
  - Example:

```
real delta;
initial
begin
   delta=4e10;
   delta=2.13;
end
integer i;
initial
   i = delta; // i gets the value 2 (rounded value of 2.13)
```

Integer, Real, and Time Register Data Types (cont'd)

- Time
  - Used to store values of simulation time
  - Keyword: `time`
  - Bit width: implementation-dependent (at least 64)
  - `$time` system function gives current simulation time
  - Example:

```
time save_sim_time;
initial
   save_sim_time = $time;
```

Arrays

- Only one-dimensional arrays supported
- Allowed for `reg`, `integer`, `time`
  - Not allowed for `real` data type
- Syntax:

```
<data_type> <var_name>[start_idx : end_idx];
```

- Examples:

```
integer count[0:7];
reg bool[31:0];
time chk_point[1:100];
reg [4:0] port_id[0:7];
integer matrix[4:0][4:0]; // illegal

count[5]
chk_point[100]
port_id[3]
```

- Note the difference between vectors and arrays

Memories

- RAM, ROM, and register-files used many times in digital systems
- Memory = array of registers in Verilog
- Word = an element of the array
  - Can be one or more bits
- Examples:

```
reg membit[0:1023];
reg [7:0] membyte[0:1023];
membyte[511]
```

- Note the difference (as in arrays):

```
reg membit[0:127];
reg [0:127] register;
```

Data Types ~ summary

- Data Values:

0,1,x,z

- Wire

- Synthesizes into wires
- Used in structural code

- Reg

- May synthesize into latches, flip-flops or wires
- Used in procedural code

- Integer

32-bit integer used as indexes

- Input, Output, inout

Defines ports of a module (wire by default)

```
module sample (a,b,c,d);

input a,b;
output c,d;

wire [7:0] b;

reg c,d;

 integer k;
```

4valuedLogic.v

```
module fourValues( a , b, c, d );
  output a, b, c, d ;

  assign a = 1;
  assign b = 0;
  assign c = a;
  assign c = b;
endmodule

module stimulus;
  fourValues X(a, b, c, d);

  initial
    begin
    #1 $display("a = %d b = %d, c = %d, d = %d", a, b, c, d);
    $finish;
    end
endmodule
```

Conflict or *race* condition. Remember this is not a procedural (i.e., sequential) block! These are continuous assign-ments.

4-valued logic:
0 – low
1 – high
x – unknown
z – undriven wire
Now explain output!

Data Values

- Numbers:

Numbers are defined by number of bits

Value of 23:

5’b10111 // Binary

5’d23 // Decimal

5’h17 // Hex

- Constants:

wire [3:0] t,d;

assign t = 23;

assign d= 4’b0111;

- Parameters:

parameter n=4;

wire [n-1:0] t, d;

`**define** Reset_state = 0, state_B =1, Run_state =2, finish_state = 3;

if(state==`Run_state)

numbers.v

```
module numbers;
  integer i, j;
  reg[3:0] x, y;

  initial
    begin
    i = ‘b1101;
    $display( "decimal i = %d, binary i = %b", i, i );
    $display( "octal i = %o, hex i = %h", i, i );

    j = -1;
    $display( "decimal j = %d, binary j = %b", j, j );
    $display( "octal j = %o, hex j = %h", j, j );

    x = 4'b1011;
    $display( "decimal x = %d, binary x = %b", x, x );
    $display( "octal x = %o, hex x = %h", x, x );

    y = 4'd7;
    $display( "decimal y = %d, binary y = %b", y, y );
    $display( "octal y = %o, hex y = %h", y, y );

    $finish;
    end
endmodule
```

Register array.

‘<base>: base can be d, b, o, h

Default base: d

Array of register arrays simulate memory. Example memory declaration with 1K 32-bit words:
reg[31:0] smallMem[0:1023];

Negative numbers are stored in two’s complement form.

Typical format: <size>’<base><number>
size is a *decimal* value that specifies the size of the number in *bits*.

Operators

- Arithmetic:

*,+,-, /,%

- Relational

<,<=,>,>=,==, !=

- Bit-wise Operators
  - Not: ~
  - XOR: ^
  - And : **&**   5'b11001 & 5'b01101 ==> 5'b01001
  - OR: **|**
  - XNOR: ~^   or   ^~
- Logical Operators

Returns 1or 0, treats all nonzero as 1

  - **!** : Not
  - **&&** : AND   27 && -3 ==> 1
  - **||** : OR

```
reg [3:0] a, b, c, d;
wire[7:0] x,y,z;
parameter n =4;

c = a + b;
d = a *n;

If(x==y) d = 1; else d =0;

d = a ~^ b;

if ((x>=y) && (z)) a=1;
  else a = !x;
```

Operators

- Reduction Operators:

Unary operations returns single-bit values

- & : and
- | :or
- ~& : nand
- ~| : nor
- ^ : xor
- ~^ :xnor
- Shift Operators

Shift Left: **<<**

Shift right: **>>**

- Concatenation Operator

{ } (concatenation)

{ n{item} } (n fold replication of an item)

- Conditional Operator

Implements if-then-else statement

(cond) ? (result if cond true) : (result if cond false)

```
module sample (a, b, c, d);
input [2:0] a, b;
output [2;0] c, d;
wire z,y;

assign z = ~| a;
c = a * b;
If(a==b) d = 1; else d =0;

d = a ~^ b;

if ((a>=b) && (z)) y=1;
  else y = !x;

assign d << 2;  //shift left
      twice
assign {carry, d} = a + b;
assign c = {2{carry},2{1'b0}};
// c = {carry,carry,0,0}
```



```
assign c= (inc==2)? a+1:a-1;
```

clockGenerator.v

```
module clockGenerator(clk);
  output clk;
  reg clk;

  initial
    begin
    clk = 0;
    end

  always
    #5 clk = ~clk;
endmodule
```

Port list. Ports can be of three types: *input*, *output*, *inout*. Each must be declared.

Internal register.

Register reg data type can have one of four values: 0, 1, x, z. Registers *store* a value till the next assignment. Registers are assigned values in procedural blocks.

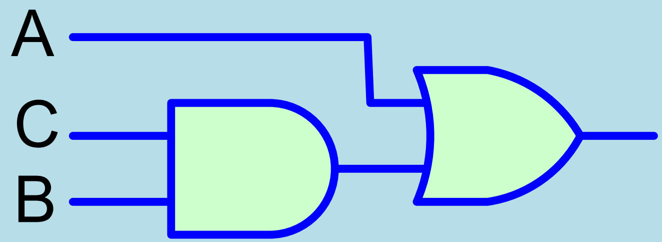
If this module is run stand-alone make sure to add a $finish statement here or simulation will never complete!

The delay is half the clock period.

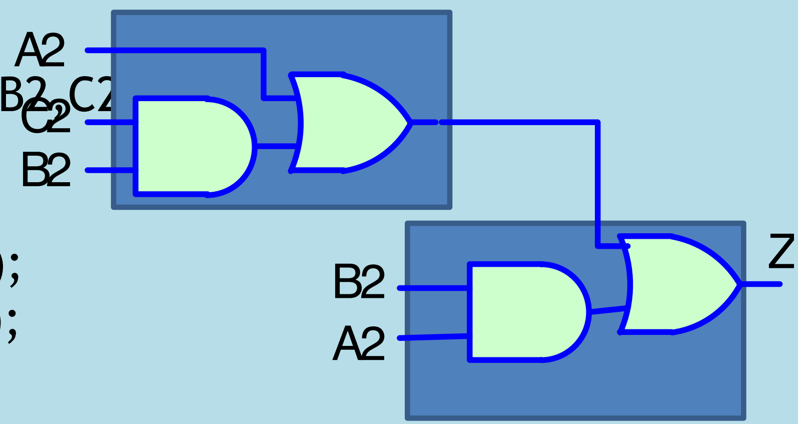
Verilog Structure

- All code are contained in modules
- Can invoke other modules
- Modules cannot be contained in another module

```
module gate(Z,A,B,C);
input A,B,C;
output Z;
assign Z = A|(B&C);
Endmodule
```




```
module two_gates(Z2,A2,B2,C2
input A2,B2,C2;
output Z2;
gate gate_1(G2,A2,B2,C2);
gate gate_2(Z2,G2,A2,B2);
endmodule
```



Structural Vs Procedural

Structural

- textual description of circuit
- order does not matter
- Starts with **assign** statements
- Harder to code
- Need to work out logic

```
wire c, d;
assign c =a & b;
assign d = c |b;
```

Procedural

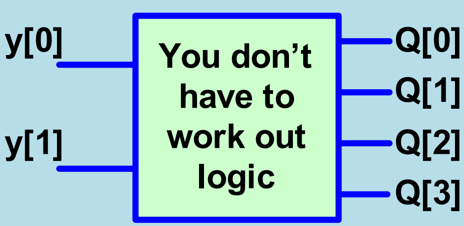
- Think like C code
- Order of statements are important
- Starts with **initial** or **always** statement
- Easy to code
- Can use case, if, for

```
reg c, d;
always@ (a or b or c)
    begin
            c =a & b;
            d = c |b;
    end
```

Structural Vs Procedural

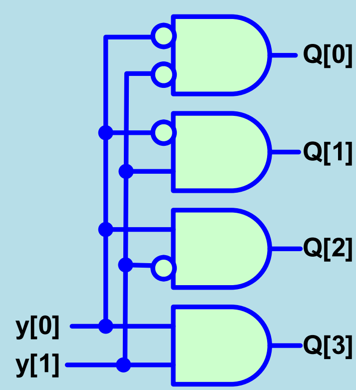
Procedural

```
 reg [3:0] Q;
wire [1:0] y;
always@(y)
  begin
    Q=4'b0000;
    case(y) begin
    2'b00: Q[0]=1;
    2'b01: Q[1]=1;
    2'b10: Q[2]=1;
    2'b11: Q[3]=1;
    endcase
end
```



Structural

```
wire [3:0]Q;
wire [1:0]y;
assign
  Q[0]=(~y[1])&(~y[0]),
  Q[1]=(~y[1])&y[0],
  Q[2]=y[1]&(~y[0]),
  Q[3]=y[1]&y[0];
```



Blocking Vs Non-Blocking

Blocking

- <variable> = <statement>
- Similar to C code
- The next assignment waits until the present one is finished
- Used for combinational logic

Non-blocking

- <variable> **<=** <statement>
- The inputs are stored once the procedure is triggered
- Statements are executed in parallel
- Used for flip-flops, latches and registers

Do not mix both assignments in one procedure

Blocking Vs Non-Blocking

```
Initial
  begin
   #1 e=2;
   #1 b=1;
   #1 b<=0;
        e<=b;   // grabbed the old b
        f=e;   // used old e=2, did not wait
   e<=b
```

blockingVSnba1.v

```
module blockingVSnba1;
  integer i, j, k, l;

  initial
    begin
    #1 i = 3;
    #1 i = i + 1;
    j = i +1;
    #1 $display( "i = %d, j = %d", i, j );

    #1 i = 3;
    #1 i <= i + 1;
    j <= i + 1;
    #1 $display( "i = %d, j = %d", i, j );

    $finish;
    end
endmodule
```

Blocking (procedural) assignment: the whole statement must execute before control is released, as in traditional programming languages.

Non-blocking (procedural) assignment: *all* the RHSs for the current time instant are evaluated (and stored transparently in temporaries) first and, subsequently, the LHSs are updated at the end of the time instant.

blockingVSnba2.v

```
module blockingVSnba2(clk);
  input clk;
  clockGenerator cg(clk);
  integer i, j;

  initial
    begin
    i = 10;
    #50 $finish;
    end

  always @(posedge clk)
    i = i + 1;    // i <= i + 1;
  always @(posedge clk)
    j = i;   // j <= i;

  always @(negedge clk)
    $display("i = %d, j = %d", i, j);
endmodule
```

Compile with clockGenerator.v.

An application of non-blocking assignments to solve a *race* problem.

With blocking assignments we get different output depending on the order these two statements are executed by the simulator, though they are both supposed to execute “simultaneously” at posedge clk - race problem.

Race problem is solved if the non-blocking assignments (after the comments) are used instead - output is unique.

blockingVSnba3.v

```
module blockingVSnba3;
  reg[7:0] dataBuf, dataCache, instrBuf, instrCache;

  initial
    begin
    dataCache = 8'b11010011;
    instrCache = 8'b10010010;

    #20;
    $display("Time = %d, dataBuf = %b, instrBuf = %b", $time, dataBuf, instrBuf);
    dataBuf <= #1 dataCache;
    instrBuf <= #1 instrCache;
    #1 $display("Time = %d, dataBuf = %b, instrBuf = %b", $time, dataBuf, instrBuf);

    $finish;
    end
endmodule
```

The most important application of non-blocking assignments is to model concurrency in hardware systems at the behavioral level.

Both loads from *dataCache* to *dataBuf* and *instrCache* to *instrBuf* happen concurrently in the 20-21 clock cycle.

Replace non-blocking with blocking assignments and observe.

System Tasks and Compiler Directives

System Tasks

- System Tasks: standard routine operations provided by Verilog
  - ❑Displaying on screen, monitoring values, stopping and finishing simulation, etc.
- All start with `$`

System Tasks (cont'd)

- `$display`: displays values of variables, strings, expressions.
  - Syntax: `$display(p1, p2, p3, …, pn);`
  - `p1,…, pn` can be quoted string, variable, or expression
  - Adds a new-line after displaying `pn` by default
  - Format specifiers:
    - `%d, %b, %h, %o`: display variable respectively in decimal, binary, hex, octal
    - `%c, %s`: display character, string
    - `%e, %f, %g`: display real variable in scientific, decimal, or whichever smaller notation
    - %v: display strength
    - %t: display in current time format
    - %m: display hierarchical name of this module

System Tasks (cont'd)

- `$display` **examples:**
  - `$display("Hello Verilog World!");`

    Output: `Hello Verilog World!`

  - `$display($time);`

    Output: `230`

  - `reg [0:40] virtual_addr;`
  - `$display("At time %d virtual address is %h", $time, virtual_addr);`

    Output: `At time 200 virtual address is 1fe000001c`

System Tasks (cont'd)

- `reg [4:0] port_id;`
- `$display("ID of the port is %b", port_id);`

  Output: `ID of the port is 00101`

- `reg [3:0] bus;`
- `$display("Bus value is %b", bus);`

  Output: `Bus value is 10xx`

- `$display("Hierarchical name of this module is %m");`

  Output: `Hierarchical name of this module is top.p1`

- `$display("A \n multiline string with a %% sign.");`

  Output:
  ```
  A
   multiline string with a % sign.
  ```

System Tasks (cont'd)

- `$monitor`: monitors a signal when its value changes
- Syntax: `$monitor(p1, p2, p3, …, pn);`
  - `p1,…, pn` can be quoted string, variable, or signal names
  - Format specifiers just as `$display`
  - Continuously monitors the values of the specified variables or signals, and displays the entire list whenever any of them changes.
  - `$monitor` needs to be invoked only once (unlike `$display`)
    - Only one `$monitor` (the latest one) can be active at any time
    - `$monitoroff` to temporarily turn off monitoring
    - `$monitoron` to turn monitoring on again

System Tasks (cont'd)

- `$monitor` Examples:

```
initial
begin
   $monitor($time, "Value of signals clock=%b, reset=
  %b", clock, reset);
end
```

  - Output:

```
0 value of signals clock=0, reset=1
5 value of signals clock=1, reset=1
10 value of signals clock=0, reset=0
```

System Tasks (cont'd)

- `$stop`: stops simulation
  - Simulation enters interactive mode when reaching a `$stop` system task
  - Most useful for debugging
- `$finish`: terminates simulation
- Examples:

```
initial
begin
  clock=0;
  reset=1;
  #100 $stop;
  #900 $finish;
end
```

Compiler Directives

- General syntax:
  `` `<keyword> ``
- `` `define ``: similar to `#define` in C, used to define macros
- `` `<macro_name> `` to use the macro defined by `` `define ``
- Examples:

```
`define WORD_SIZE 32
`define S $stop

`define WORD_REG reg [31:0]
`WORD_REG a_32_bit_reg;
```

Compiler Directives (cont'd)

- `` `include ``: Similar to `#include` in C, includes entire contents of another file in your Verilog source file
- Example:

```
`include header.v
...
<Verilog code in file design.v>
...
```

Behavior Modeling

simpleBehavioral.v

Sensitivity trigger: when any of a, b or c changes. Replace this statement with "initial". Output?!

```
module aOrNotbOrc(d, a, b, c);
  output d;
  input a, b, c;
  reg d, p;

  always @(a or b or c)
    begin
    p = a || ~b;
    d = p || c;
    end
endmodule
```

Modules are of three types: *behavioral, dataflow*, *gate-level*. Behavioral modules contain code in procedural blocks.

Statements in a procedural block *cannot be re-ordered* without affecting the program as these statements are executed sequentially, exactly like in a conventional programming language such as C.

Ports are of three types: *input*, *output*, *inout*. Each must be declared. Each port also has a data type: either *reg* or *wire* (*net*). Default is wire. Inputs and inouts are always wire. Output ports that hold their value are reg, otherwise wire. More later…

One port register, one internal register.

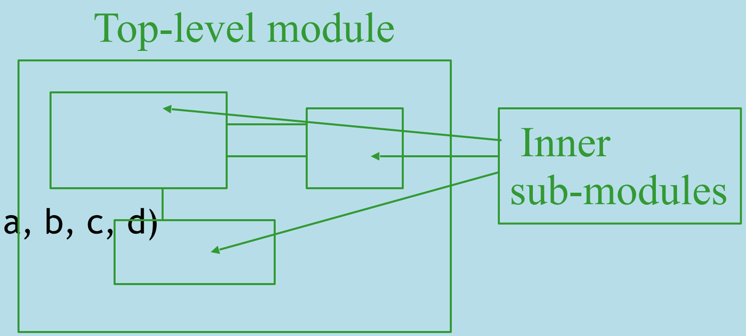
Wires are part of the more general class of nets. However, the only nets we shall design with are wires.

simpleBehavioral.v (cont.)
Top-level stimulus module

```
module stimulus;
  integer i, j, k;
  reg a, b, c;
  aOrNotbOrc X(d, a, b, c);

  initial
    begin
    for ( i=0; i<=1; i=i+1 )
      for ( j=0; j<=1; j=j+1 )
        for ( k=0; k<=1; k=k+1 )
          begin
          a = i;
          b = j;
          c = k;
          #1 $display("a = %d b = %d, c = %d, d = %d", a, b, c, d)
          end
    $finish;
    end
endmodule
```

Instantiation.

<u>Verilog Good Design Principle</u> There is one top-level module, typically called system or stimulus, which is *uninstantiated* and has no ports. This module contains *instantiations* of lower-level (inner) sub-modules. Typical picture below.

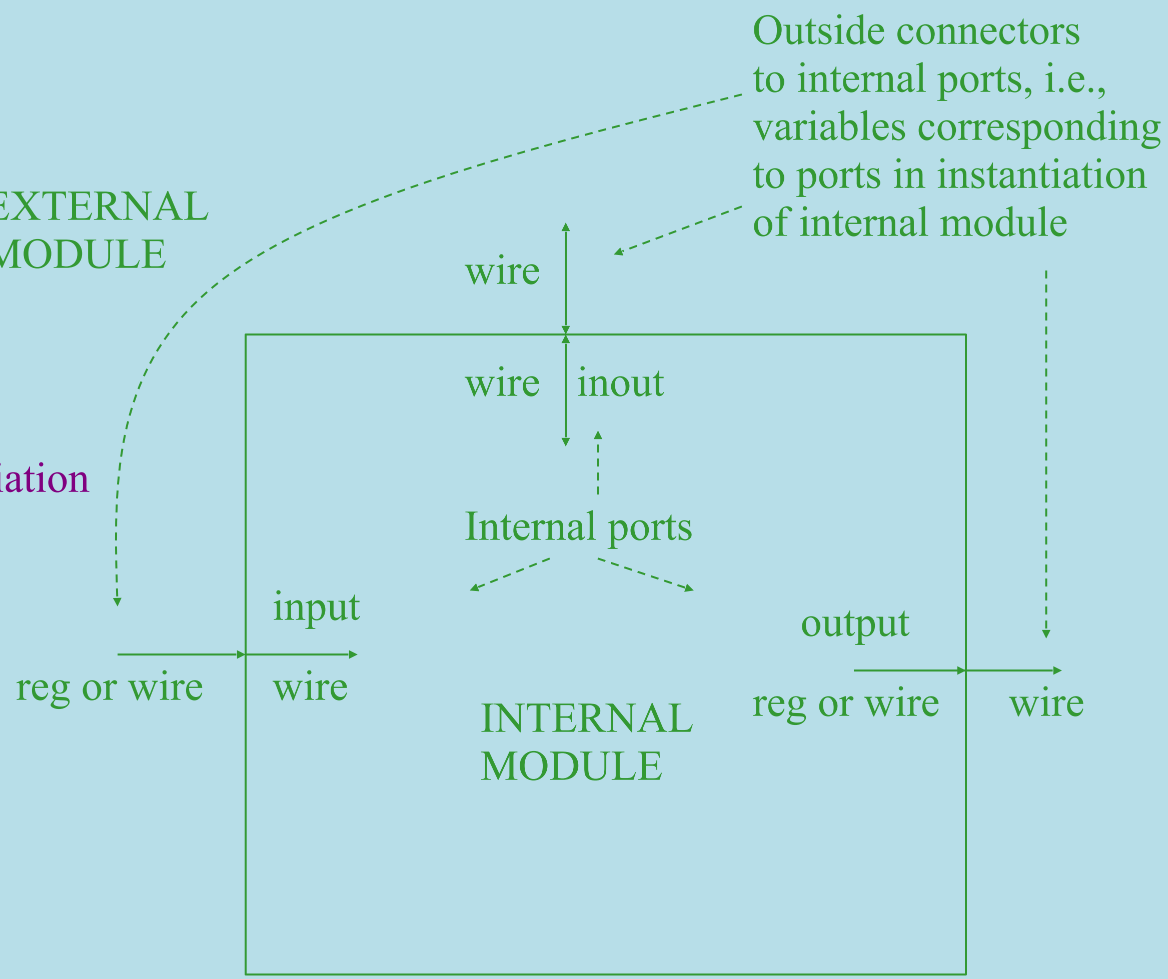


Remove the #1 delay. Run. Explain!

Port Rules Diagram

Example:
```
module external
reg a;
wire b;
internal in(a, b); //instantiation
…
endmodule

module internal(x, y)
input x;
output y;
wire x;
reg y;
…
endmodule
```

port-connector



General rule (with few exceptions) Ports in all modules except for the stimulus module should be wire. Stimulus module has registers to set data for internal modules and wire ports only to read data from internal modules.

If Statements

Syntax

if (*expression*)
begin
...*statements*...
end

else if (expression)
begin
...*statements*...
end
...*more else if blocks*

else
begin
...*statements*...
end

```
if (alu_func == 2'b00)
  aluout = a + b;
else if (alu_func == 2'b01)
  aluout = a - b;
else if (alu_func == 2'b10)
  aluout = a & b;
else // alu_func == 2'b11
  aluout = a | b;
```

Case Statements

Syntax

```
case (expression)
  case_choice1:
  begin
    ...statements...
  end

  case_choice2:
  begin
    ...statements...
  end

  ...more case choices blocks...

  default:
  begin
    ...statements...
  end
endcase
```

```
case (alu_ctr)
  2'b00:  aluout = a + b;
  2'b01:  aluout = a - b;
  2'b10:  aluout = a & b;
  default: aluout = 1'bx; // Treated as don't cares for
endcase                   // minimum logic generation.
```

For loops

Syntax

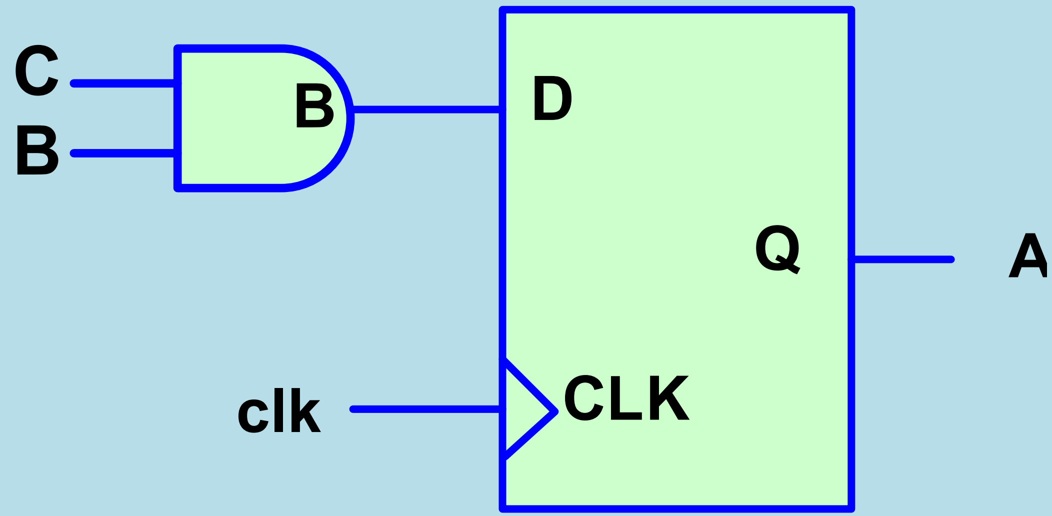
```
for (count= value1;
      count</<=/>/>= value2;
      count=count+/- step)
begin
   ...statements...
end
```

```
integer j;

for(j=0;j<=7;j=j+1)
begin
  c[j] = a[j] + b[j];
end
```

Component Inference

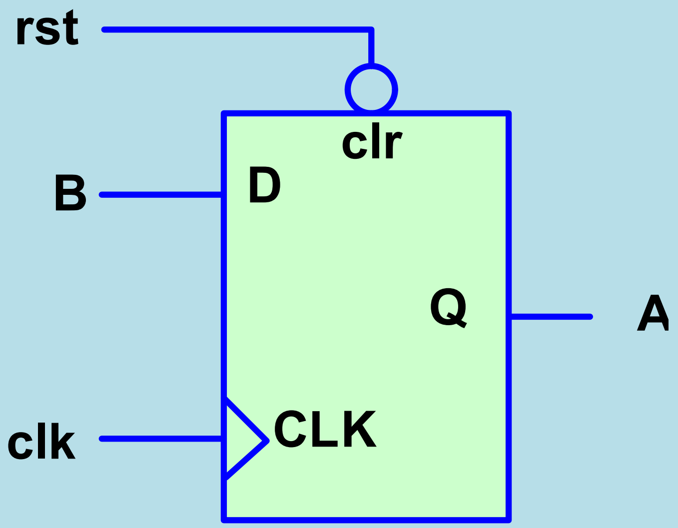
Flip-Flops

```
always@(posedge clk)
begin
  a<=b&c;
end
```



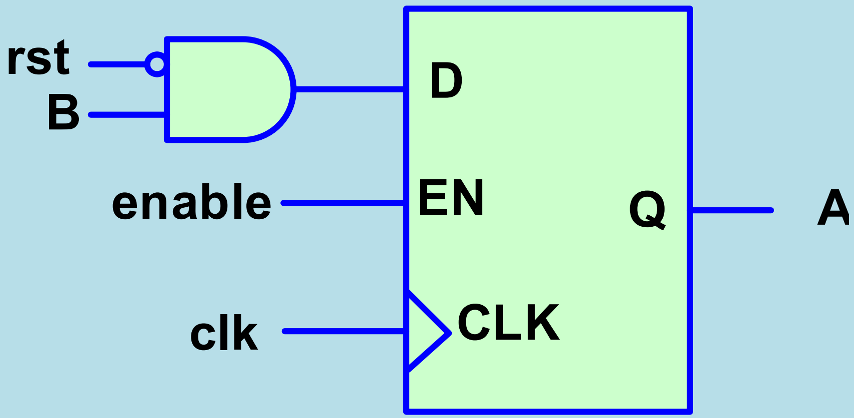
D Flip-Flop with Asynchronous Reset

```
always@(posedge clk or negedge rst)
begin
  if (!rst) a<=0;
  else a<=b;
end
```



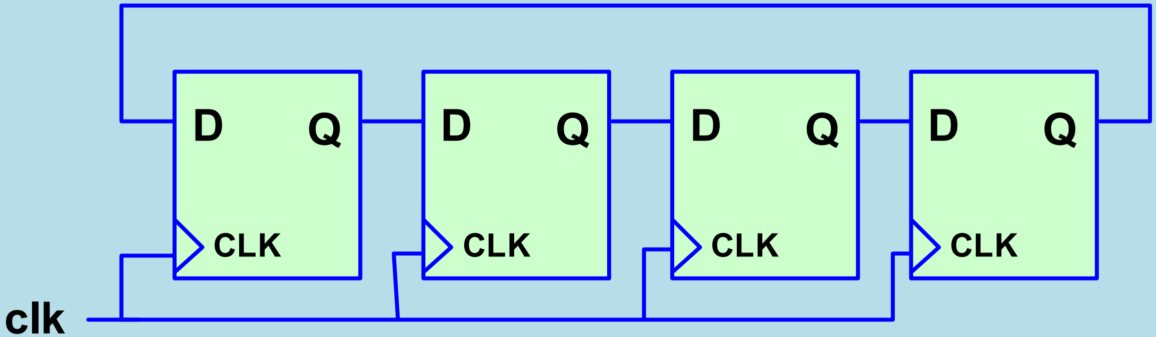
D Flip-flop with Synchronous reset and Enable

```
always@(posedge clk)
begin
  if (rst) a<=0;
  else if (enable) a<=b;
end
```



Shift Registers

```
reg[3:0] Q;
always@(posedge clk or
    posedge rset )
begin
  if (rset) Q<=0;
  else begin
    Q <=Q << 1;
    Q[0]<=Q[3];
   end
```



Multiplexers

Method 1

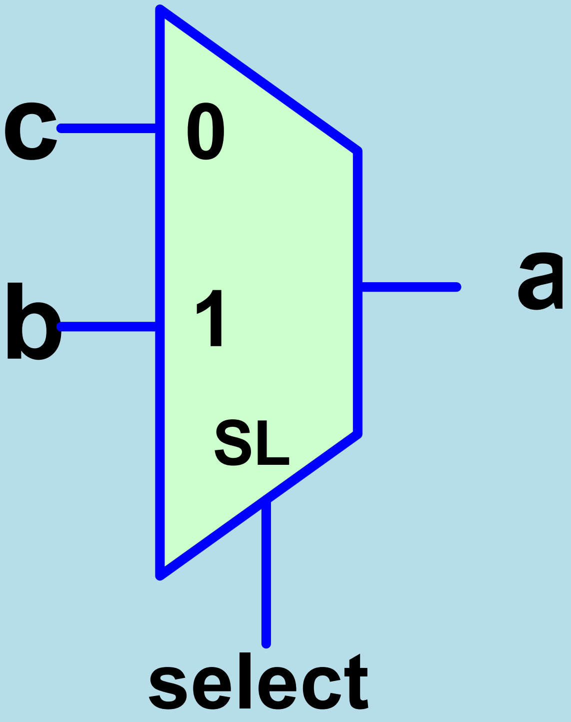
```
assign a = (select ? b : c);
```

Method 2

```
always@(select or b or c) begin
  if(select) a=b;
  else  a=c;
end
```

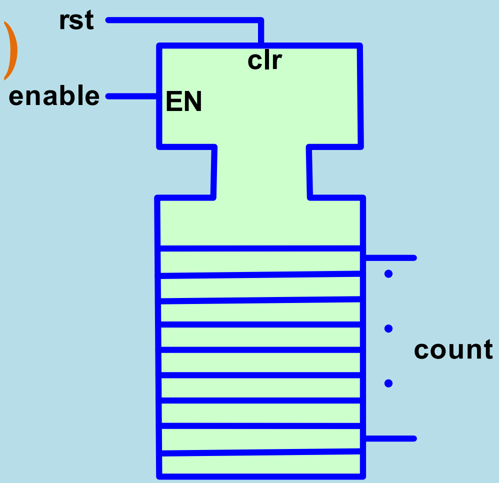
Method 2b

```
case(select)
  1'b1: a=b;
  1'b0: a=c;
endcase
```



Counters

```
reg [7:0] count;
wire enable;
always@(posedge clk or negedge rst)
begin
  if (rst) count<=0;
  else if (enable)
    count<=count+1;
end
```

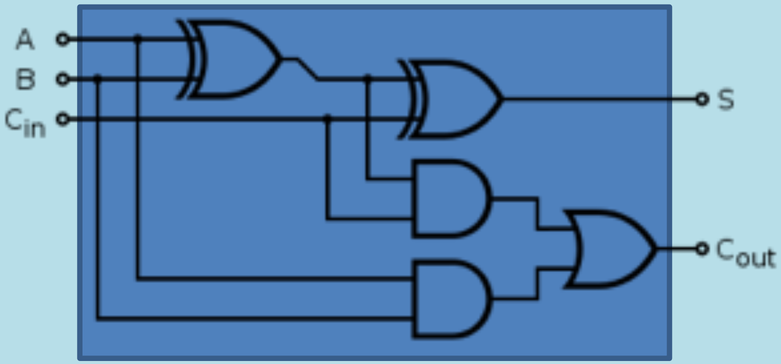


Step by Step
4-bit adder

Princess Sumaya University for Technology
Computer Organization and Architecture I

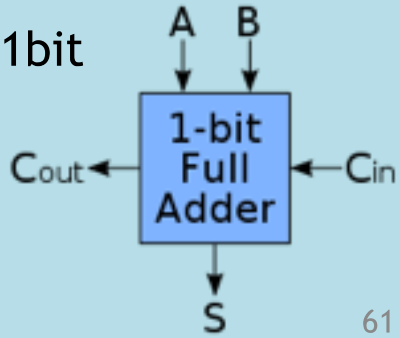
4-bit Adder

- Step 1: build a 1-bit full adder as a module
  - ❑ $S = (a)\ XOR\ (b)\ XOR\ (C_{in})$; ($S = a \hat{}\, b \hat{}\, C_{in}$)
  - ❑ $C_{out} = (a\&b) \,|\, (C_{in}\&(a+b))$



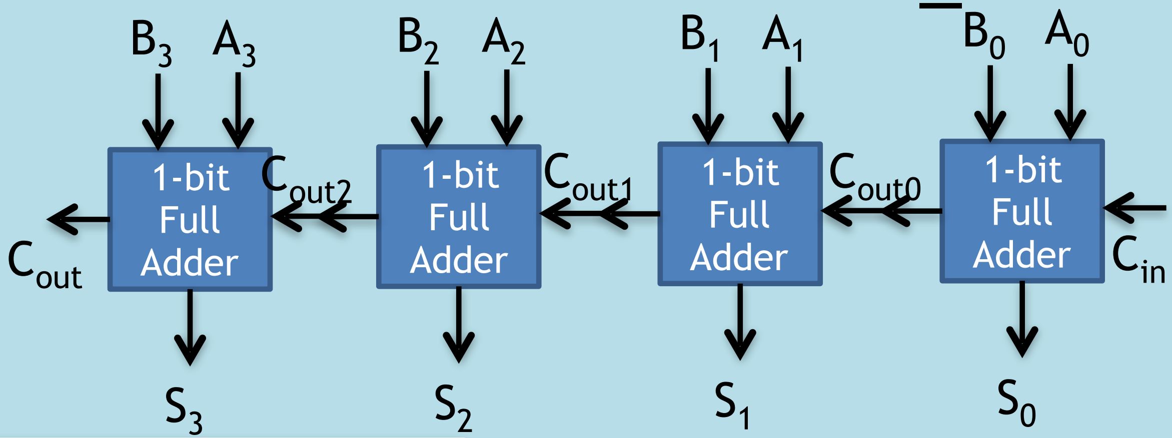

```
module FA_1bit (S,Cout,a,b,Cin);
begin
input a,b,Cin;
Output S, Cout;

        assign Sum = a^b^Cin;
        assign Carry = (a&b) | (Cin&(a^b));

endmodule
```

Module add_1bit



4-bit Adder

- Step 2: initiate 4 instances of FA_1bit module




```
module FA_4bits (S,Cout,A,B,Cin);
begin
            input [3:0] A, B;
            input                   Cin;
            output [3:0]            S;
            output                  Cout
            wire        Cout0, Cout1, Cout2

FA_1bit     FA1(S[0], Cout0,A[0],B[0],Cin);
FA_1bit     FA1(S[1], Cout1,A[1],B[1],Cout0);
FA_1bit     FA1(S[2], Cout2,A[2],B[2],Cout1);
FA_1bit     FA1(S[3], Cout,A[3],B[3],Cout2);
end
endmodule;
```

The inputs and the output are 4-bits wide

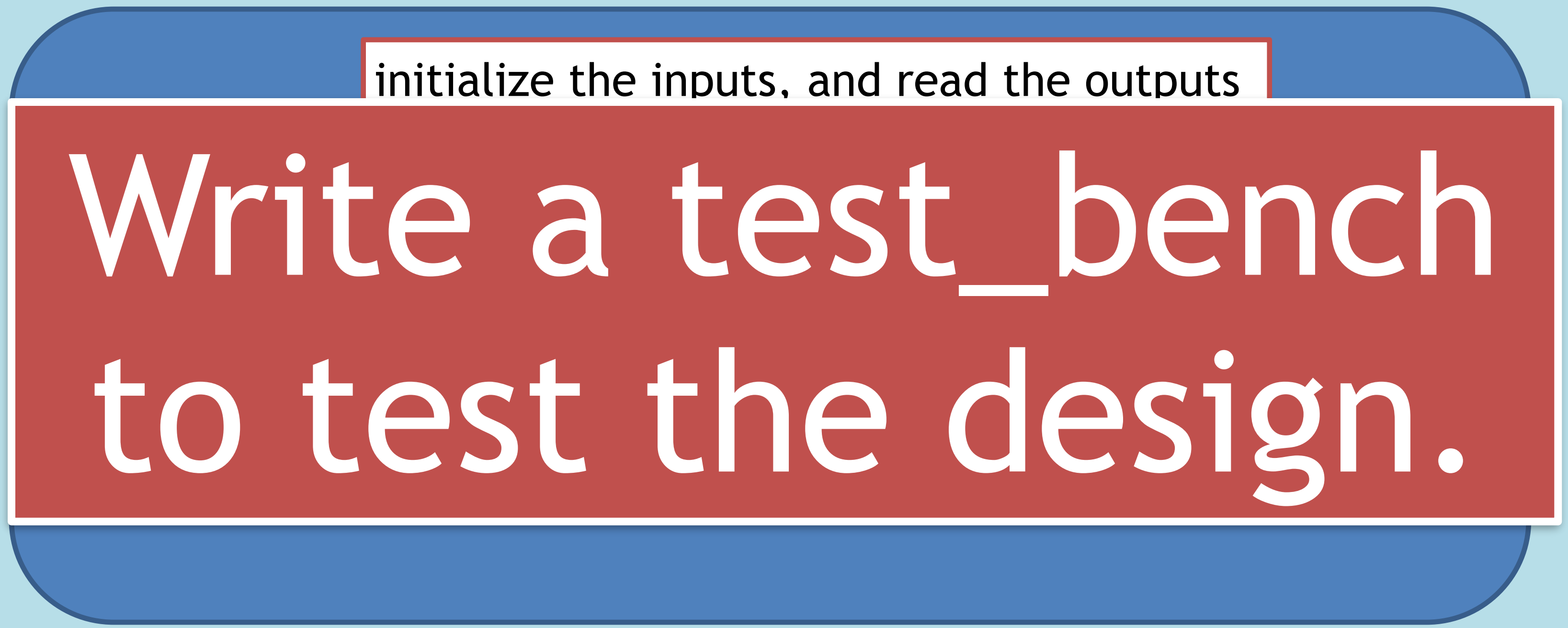
we need wires to propagate the carry from one stage to the next

you may name the instances with any name, but you have to maintain the order of the inputs and outputs

4-bit Adder

- Step 3: write a test-bench to test your design and generate outs using sample inputs.

test_bench



4-bit Adder

```
module test_bench; // you may name it by any name
//define the variables you will use in the design
reg [3:0]  A,B,S;
reg        Cin, Cout
// Create an instance from the full adder
FA_4bits   FA(S[3:0],Cout, A[3:0], B[3:0], Cin);
//initialize the variables once
initial
A = 5; B = 6; S = 0; Cin = 0; Cout = 0;
initial
begin
$display("A=%d, B=%d, the sum = %d, and the carry = %d", A,B,S,Cout)
$finish
end
endmodule
```

4-bit Adder

```
module test_bench; // you may name it by any name
//define the variables you will use in the design
reg [3:0]  A,B,S;
integer    I,j;
reg        Cin, Cout
// Create an instance from the full adder
FA_4bits   FA(S,Cout, A, B, Cin);
//initialize the variables once
 initial begin
      $monitor ("A: %d  B: %d sum: %d  carry: %d", A, B, sum, carry);
      for (i=0; i<16; i=i+1)
         for (j=0; j<16; j=j+1)
             begin
               A = i;
               B = j;
             #1 ;
             end
      $finish;
   end
endmodule
```

System calls.

More Examples

blocksTime1.v

```
module blocksTime1;
   integer i, j;

   initial
      begin
      i = 0;
      j = 3;
      $display( "i = %d, j = %d", i, j );
      $finish;
      end
endmodule
```

Another behavioral module.

Integer data type: other types are time, real and realtime (same as real).

One initial procedural block.

blocksTime2.v

```
module blocksTime2;
  integer i, j;

  initial
    begin
    #2 i = 0;
    #5 j = i;
    $display( "time = %d, i = %d, j = %d", $time, i, j );
    end

  initial
    #3 i = 2;

  initial
    #10 $finish;
endmodule
```

Time *delay* models signal propagation delay in a circuit.

Multiple initial blocks.
Delays add within each block,
but different initial blocks all start
at time $time = 0 and run
in *parallel* (i.e., *concurrently*).

blocksTime3.v

```
module blocksTime3;
  integer i, j;

  initial
    begin
    #2 i = 0;
    #5 j = i;
    $display( "time = %d, i = %d, j = %d", $time, i, j );
    end

  initial
    begin
    #3 i = 2;
    #2 j = i;
    $display( "time = %d, i = %d, j = %d", $time, i, j );
    #1 j = 8;
    $display( "time = %d, i = %d, j = %d", $time, i, j );
    end

  initial
    #10 $finish;
endmodule
```

Important Verilog is a discrete event simulator: events are executed in a time-ordered queue.

Multiple initial blocks. Predict output before you run!

blocksTime4.v

```
module blocksTime4;
  integer i, j;

  initial
    begin
    i = 0;
    j = 3;
    end

  initial
    #10 $finish;

  always
    begin
    #1
    i = i + 1;
    j = j + 1;
    $display( "i = %d, j = %d", i, j );
    end
endmodule
```

Always block is an infinite loop. Following are same:

```
always
  begin
  …
  end
```

```
initial
  begin
   while(1)
     begin
     …
    end
   end
```

```
initial
  begin
   forever
     begin
     …
     end
   end
```

Comment out this delay.
Run. Explain the problem!

clockGenerator.v

```
module clockGenerator(clk);
  output clk;
  reg clk;

  initial
    begin
    clk = 0;
    end

  always
    #5 clk = ~clk;
endmodule
```

Port list. Ports can be of three types: *input*, *output*, *inout*. Each must be declared.

Internal register.

Register reg data type can have one of four values: 0, 1, x, z. Registers *store* a value till the next assignment. Registers are assigned values in procedural blocks.

If this module is run stand-alone make sure to add a $finish statement here or simulation will never complete!

The delay is half the clock period.

useClock.v

Compile with the clockGenerator.v module.

```
module useClock(clk);
   input clk;
   clockGenerator cg(clk);

   initial
      #50 $finish;

   always @(posedge clk)
$display("Time = %d, Clock up!", $time);

   always @(negedge clk) //
      $display("Time = %d, Clock down!", $time);
endmodule
```

Event trigger.

blocksTime5.v

```
// Ordering processes without advancing time
module blockTime5;
   integer i, j;

   initial
      #0
      $display( "time = %d, i = %d, j = %d", $time, i, j );

   initial
      begin
      i = 0;
      j = 5;
      end

   initial
      #10 $finish;
endmodule
```

#0 delay causes the statement to execute after other processes scheduled at that time instant have completed. $time does not advance till after the statement completes.

Comment out the delay.
Run. Explain what happens!

blocksTime6.v

```
module blocksTime6;
  integer i, j;

  initial
    begin
    #2 i = 0;
    j = #5 i;
    $display( "time = %d, i = %d, j = %d", $time, i, j );
    end

  initial
    #3 i = 2;

  initial
    #10 $finish;
endmodule
```

Intra-assignment delay: RHS is computed and stored in a temporary (transparent to user) and LHS is assigned the temporary after the delay.

Compare output with blocksTime2.v.

simpleDataflow.v

```
module aOrNotbOrc(d, a, b, c);
   output d;
   input a, b, c;
   wire p, q;

   assign q = ~b;
   assign p = a || q;
   assign d = p || c;
endmodule
```

A *dataflow* module does not contain procedures.

Statements in a dataflow module *can be re-ordered* without affecting the program as they simply describe a *set of data manipulations and movements* rather than a *sequence of actions* as in behavioral code. In this regard dataflow code is very similar to gate-level code.

Continuous assignment statements: any change in the RHS causes instantaneous update of the wire on the LHS, unless there is a programmed delay.

Use stimulus module from behavioral code.

simpleGate.v

```
module aOrNotbOrc(d, a, b, c);
  output d;
  input a, b, c;
  wire p, q;

  not(q, b);
  or(p, a, q);
  or(d, p, c);
endmodule
```

A *gate-level* module does not contain procedures.

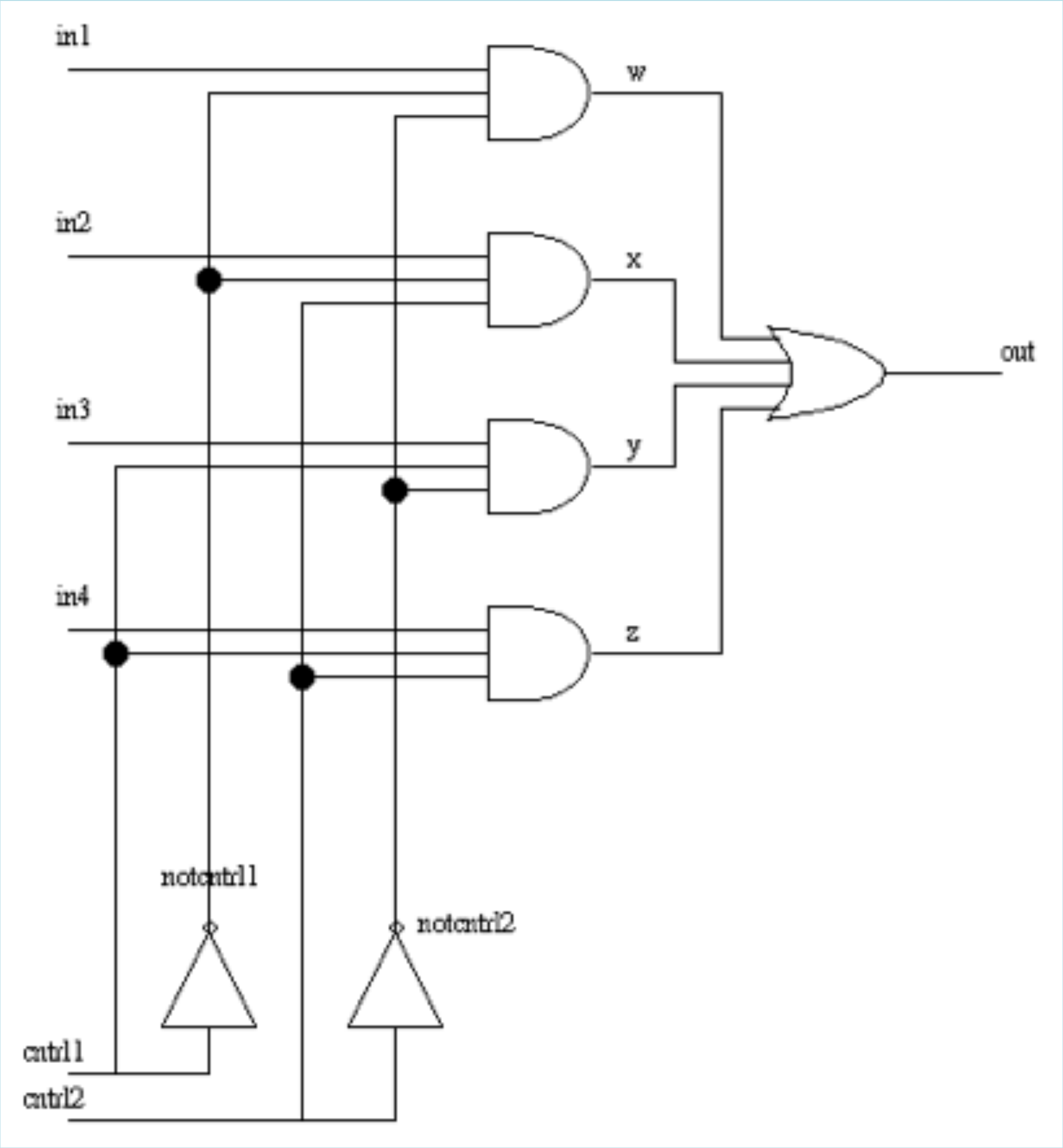
Statements in a gate-level module *can be re-ordered* without affecting the program as they simply describe a *set of connections* rather than a *sequence of actions* as in behavioral code. A gate-levelmodule is equivalent to a *combinational circuit*.

Wire data type can have one of four values: 0, 1, x, z. Wires *cannot store* values – they are continuously *driven*.

Primitive gates. Verilog provides several such, e.g., *and*, *or*, *nand*, *nor*, *not*, *buf*, etc.

Use stimulus module from behavioral code.

4-to-1 multiplexor logic diagram



4-to-1 multiplexor (Folder Multiplexor)

Following are four different Verilog implementations of the same multiplexor.

A stimulus module is shared to test each implementation.

multiplexor4_1Gate.v

```
module multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);
    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;
    wire notcntlr1, notcntrl2, w, x, y, z;

    not (notcntrl1, cntrl1);
    not (notcntrl2, cntrl2);

    and (w, in1, notcntrl1, notcntrl2);
    and (x, in2, notcntrl1, cntrl2);
    and (y, in3, cntrl1, notcntrl2);
    and (z, in4, cntrl1, cntrl2);

    or (out, w, x, y, z);
endmodule
```

Recall default type is wire.

Structural gate-level code based exactly on the logic diagram.

multiplexor4_1Stimulus.v (Folder Multiplexor)

```
module muxstimulus;
      reg IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;
       wire OUT;

       multiplexor4_1 mux1_4(OUT, IN1, IN2, IN3, IN4, CNTRL1, CNTRL2);

       initial
      begin
             IN1 = 1; IN2 = 0; IN3 = 1; IN4 = 0;
             $display("Initial arbitrary values");
             #0 $display("input1 = %b, input2 = %b, input3 = %b, input4 = %b\n",
                    IN1, IN2, IN3, IN4);

             {CNTRL1, CNTRL2} = 2'b00;
             #1 $display("cntrl1=%b, cntrl2=%b, output is %b", CNTRL1, CNTRL2, OUT);
```

Stimulus code that generates test vectors.

Concatenation.

multiplexor4_1Stimulus.v (cont.)

```
            {CNTRL1, CNTRL2} = 2'b01;
            #1 $display("cntrl1=%b, cntrl2=%b output is %b", CNTRL1, CNTRL2, OUT);

             {CNTRL1, CNTRL2} = 2'b10;
            #1 $display("cntrl1=%b, cntrl2=%b output is %b", CNTRL1, CNTRL2, OUT);

            {CNTRL1, CNTRL2} = 2'b11;
              #1 $display("cntrl1=%b, cntrl2=%b output is %b", CNTRL1, CNTRL2, OUT);
      end
endmodule
```

multiplexor4_1Logic.v (Folder Multiplexor)

```
module multiplexor4_1(out, in1, in2, in3 ,in4, cntrl1, cntrl2);
    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;

    assign out =  (in1 & ~cntrl1 & ~cntrl2) |
                  (in2 & ~cntrl1 &  cntrl2)  |
                  (in3 &  cntrl1 & ~cntrl2)  |
                  (in4 &  cntrl1 &  cntrl2);
endmodule
```

RTL (dataflow) code using continuous assignments rather than a gate list.

multiplexor4_1Conditional.v (Folder Multiplexor)


```
module  multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);
      output out;
      input in1, in2, in3, in4, cntrl1, cntrl2;

        assign out = cntrl1 ? (cntrl2 ? in4 : in3) : (cntrl2 ? in2 : in1);
endmodule
```

More RTL (dataflow) code – this time using conditionals in a continuous assignment.

multiplexor4_1Case.v (Folder Multiplexor)

```
module multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);
        output out;
       input in1, in2, in3, in4, cntrl1, cntrl2;
       reg out;


       always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
             case ({cntrl1, cntrl2})
                          2'b00 : out = in1;
                          2'b01 : out = in2;
                          2'b10 : out = in3;
                          2'b11 : out = in4;
                          default : $display("Please check control bits");
             endcase
endmodule
```

Behavioral code: output out must now be of reg type as it is assigned values in a procedural block.

8-to-3 encoder truth table

Input								Output		
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

8-to-3 encoder (Folder Encoder)

Following are four different Verilog implementations of the same encoder.

Each has its own stimulus module.

encoder8_3Behavioral.v

```
module encoder8_3( encoder_out , enable, encoder_in );
	output[2:0] encoder_out;
	input  enable;
	input[7:0] encoder_in;
	reg[2:0] encoder_out;
	always @ (enable or encoder_in)
	begin
		if (enable)
		case ( encoder_in )
			8'b00000001 : encoder_out = 3'b000;
			8'b00000010 : encoder_out = 3'b001;
			8'b00000100 : encoder_out = 3'b010;
			8'b00001000 : encoder_out = 3'b011;
			8'b00010000 : encoder_out = 3'b100;
			8'b00100000 : encoder_out = 3'b101;
			8'b01000000 : encoder_out = 3'b110;
			8'b10000000 : encoder_out = 3'b111;
			default : $display("Check input bits.");
		endcase
	end
endmodule
```

Sensitivity list.

Simple behavioral code using the case statement.

encoder8_3BehavioralStimulus.v

```
module stimulus;
    wire[2:0] encoder_out;
    reg enable;
    reg[7:0] encoder_in;
    encoder8_3 enc( encoder_out, enable, encoder_in );
    initial
    begin
        enable = 1; encoder_in = 8'b00000010;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
                enable, encoder_in, encoder_out);
        #1 enable = 0; encoder_in = 8'b00000001;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
                        enable, encoder_in, encoder_out);
        #1 enable = 1; encoder_in = 8'b00000001;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
                        enable, encoder_in, encoder_out);
        #1 $finish;
    end
endmodule
```

Stimulus for the behavioral code.

Remove this delay.
Run. Explain!

8-to-3 encoder logic equations

$$A0 = D1 + D3 + D5 + D7$$

$$A1 = D2 + D3 + D6 + D7$$

$$A2 = D4 + D5 + D6 + D7$$

encoder8_3structural.v (Folder Encoder)

```
module encoder8_3( encoder_out , encoder_in );
    output[2:0] encoder_out;
    input[7:0] encoder_in;

    or( encoder_out[0], encoder_in[1], encoder_in[3], encoder_in[5], encoder_in[7] );
    or( encoder_out[1], encoder_in[2], encoder_in[3], encoder_in[6], encoder_in[7] );
    or( encoder_out[2], encoder_in[4], encoder_in[5], encoder_in[6], encoder_in[7] );
endmodule
```

Structural code. Why is there no enable wire?! Hint: think storage.

encoder8_3StructuralStimulus.v

```
module stimulus;
    wire[2:0] encoder_out;
    reg[7:0] encoder_in;
    encoder8_3 enc( encoder_out, encoder_in );

    initial
    begin
        encoder_in = 8'b00000010;
        #1 $display("encoder_in = %b, encoder_out = %b", encoder_in, encoder_out);
        #1 encoder_in = 8'b00000001;
        #1 $display("encoder_in = %b, encoder_out = %b", encoder_in, encoder_out);
        #1 $finish;
    end
endmodule
```

Stimulus for the structural code.

encoder8_3Mixed.v

```
module encoder8_3( encoder_out , enable, encoder_in );
        output[2:0] encoder_out;
        input enable;
        input[7:0] encoder_in;
        reg[2:0] encoder_out;
        wire b0, b1, b2;

        or( b0, encoder_in[1], encoder_in[3], encoder_in[5], encoder_in[7] );
        or( b1, encoder_in[2], encoder_in[3], encoder_in[6], encoder_in[7] );
        or( b2, encoder_in[4], encoder_in[5], encoder_in[6], encoder_in[7] );

        always @(enable or encoder_in)
        begin
        if (enable) encoder_out = {b2, b1, b0};
        end
endmodule
```

Mixed structural-behavioral code. Goal was to modify structural code to have an enable wire, which requires register output for storage.

Be careful with mixed design! It’s working may be difficult to understand.

encoder8_3MixedStimulus.v

```
module stimulus;
    wire[2:0] encoder_out;
    reg enable;
    reg[7:0] encoder_in;
    encoder8_3 enc( encoder_out, enable, encoder_in );

    initial
    begin
        enable = 1; encoder_in = 8'b00000010;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
                enable, encoder_in, encoder_out);

        #1 enable = 1; encoder_in = 8'b00000010;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
               enable, encoder_in, encoder_out);
```

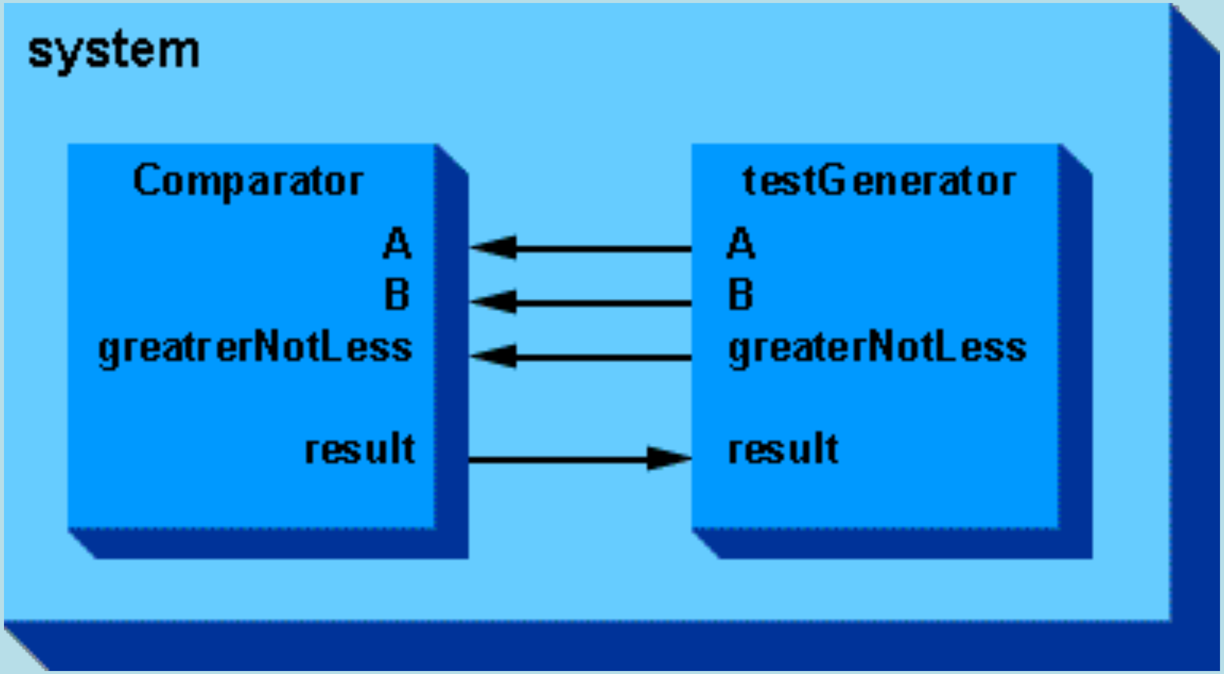
Stimulus for the mixed code.

Output is puzzling! Explain!

encoder8_3MixedStimulus.v

```
        #1 enable = 0; encoder_in = 8'b00000001;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
                    enable, encoder_in, encoder_out);

        #1 enable = 1; encoder_in = 8'b10000000;
        #1 $display("enable = %b, encoder_in = %b, encoder_out = %b",
                    enable, encoder_in, encoder_out);


        #1 $finish;
    end
endmodule
```

Comparator modules scheme



comparator.v

Parameters that may be set when the module is instantiated.

Comparator makes the comparison A ? B where ? Is determined by the input greaterNotLess and returns true(1) or false(0).

```
module comparator (result, A, B, greaterNotLess);
    parameter width = 8;
    parameter delay = 1;
    input [width-1:0] A, B;            // comparands
    input greaterNotLess;              // 1 - greater, 0 - less than
    output result;                     // 1 if true, 0 if false

    assign #delay result = greaterNotLess ? (A > B) : (A < B);

endmodule
```

stimulus.v

Stimulus for the comparator.

```
module system;
    wire greaterNotLess;       // sense of comparison
    wire [15:0] A, B;          // comparand values - 16 bit
    wire result;               // comparison result

    // Module instances
    comparator #(16, 2) comp (result, A, B, greaterNotLess);
    testGenerator tg (A, B, greaterNotLess, result);

endmodule
```

Parameters being set at module instantiation.

testGen.v

```
module testGenerator (A, B, greaterNotLess, result);
   output [15:0] A, B;
   output greaterNotLess;
   input result;
   parameter del = 5;
   reg [15:0] A, B;
   reg greaterNotLess;

   task check;
      input shouldBe;
      begin
         if (result != shouldBe)
            $display("Error! %d %s %d, result = %b", A, greaterNotLess?">":"<",
                     B, result);
      end
   endtask

   initial begin            // produce test data, check results
      A = 16'h1234;
      B = 16'b0001001000110100;
      greaterNotLess = 0;
```

Module that generates test vectors for the comparator and checks correctness of output.

Task definition: a task is exactly like a procedure in a conventional programming language.

testGen.v (cont.)

```
#del
        check(0);
        B = 0;
        greaterNotLess = 1;
    #del
        check(1);
        A = 1;
        greaterNotLess = 0;
    #del
        check(0);
        $finish;
    end
endmodule
```

Task call

Finite State Machines

Standard Form for a Verilog FSM

```
// state flip-flops
reg [2:0] state, nxt_st;
// state definitions
parameter reset=0,S1=1,S2=2,S3=3,..

// NEXT STATE CALCULATIONS
always@(state or inputs or ...)
begin
   …
   next_state= ...
   …
end
```

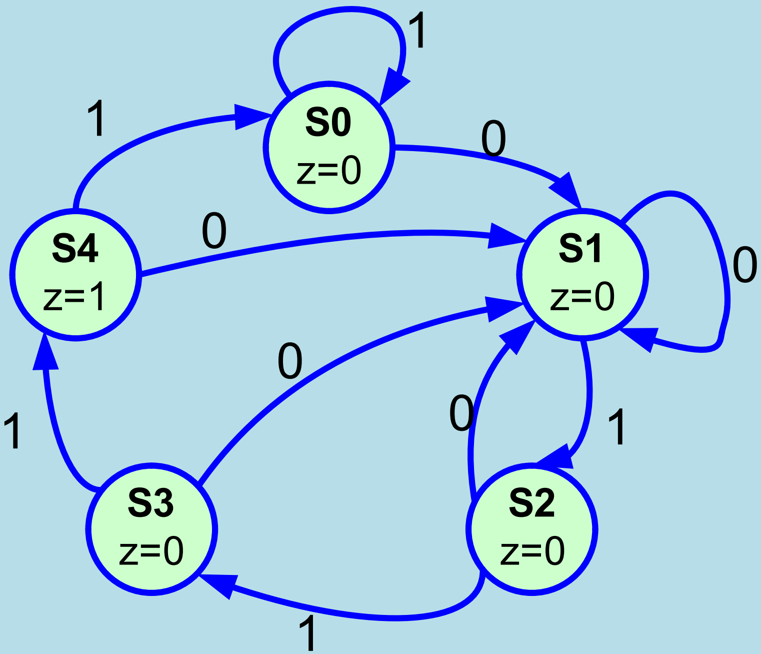
```
// REGISTER DEFINITION
always@(posedge clk)
begin
  state<=next_state;
end

// OUTPUT CALCULATIONS
output= f(state, inputs)
```

Example

```
module myFSM (clk, x, z)
input clk, x;
output z;
// state flip-flops
reg [2:0] state, nxt_st;
// state definition
parameter
    S0=0,S1=1,S2=2,S3=3,S7=7

// REGISTER DEFINITION
always @(posedge clk)
begin
  state<=nxt_st;
end

// OUTPUTCALCULATIONS
assign z = (state==S7);
```

```
// NEXT STATE CALCULATIONS
always @(state or x)
begin
case (state)
  S0: if(x) nxt_st=S1;
        else nxt_st=S0;
  S1: if(x) nxt_st=S3;
        else nxt_st=S2;
  S2: if(x) nxt_st=S0;
        else nxt_st=S7;
  S3: if(x) nxt_st=S2;
        else nxt_st=S7;
  S7:  nxt_st=S0;
  default: nxt_st = S0;
endcase
end


endmodule
```

0111 Sequence Detector



Test Benches

System tasks

- Used to generate input and output during simulation. Start with **$** sign.
- Display Selected Variables:

$display (“format_string”,par_1,par_2,...);

$monitor(“format_string”,par_1,par_2,...);

Example: $display(“Output z: %b”, z);

- Writing to a File:

$fopen, $fdisplay, $fmonitor and $fwrite

- Random number generator: **$random** (*seed*)
- Query current simulation time: $time

Test Benches

Overview

1. Invoke the verilog under design
2. Simulate input vectors
3. Implement the system tasks to view the results

Approach

1. Initialize all inputs
2. Set the clk signal
3. Send test vectors
4. Specify when to end the simulation.

Example

```
'timescale1 ns /100 ps
// timeunit =1ns; precision=1/10ns;
module my_fsm_tb;
reg clk, rst, x;
wire z;

/**** DESIGN TO SIMULATE (my_fsm)
      INSTANTIATION ****/
myfsm dut1(clk, rst, x, z);

/****RESET AND CLOCK SECTION****/
Initial
  begin
clk=0;
rst=0;
#1rst=1; /*The delay gives rst a posedge for
      sure.*/
#200 rst=0; //Deactivate reset after two clock
      cycles +1ns*/
end
always #50clk=~clk; /* 10MHz clock (50*1ns*2)
      with 50% duty-cycle */
```

```
/****SPECIFY THE INPUT WAVEFORM x ****/
Initial begin
  #1 x=0;
  #400 x=1;
  $display("Output z: %b", z);
  #100 x=0;
  @(posedge clk) x=1;

   #1000  $finish;  //stop simulation
   //without this, it will not stop
end
endmodule
```

Modelsim Demonstration