Verilog Tutorial

Adopted from

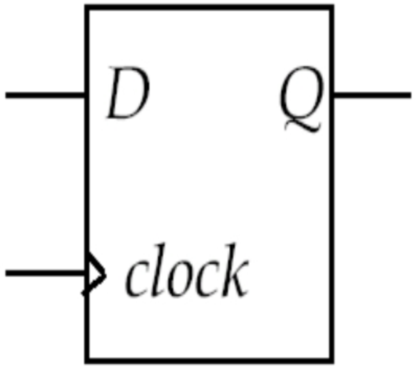
- Abdul-Rahman Elshafei's slides
- Paul Franzon slides
- Steve Ko
- Digital Design, 4th Edition, Morris Mano, Micheal Ciletti

Verilog Basics

Basic Verilog Constructs

- Flip-Flop
  - Behavior: For every positive edge of the clock Q changes to become equal to D
- Write behavior as "code"

```
always@(posedge clock)
  Q <= D
```



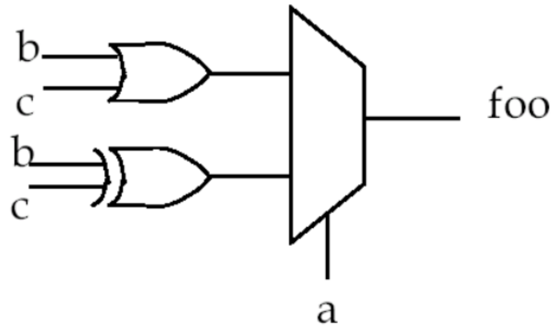
- **always@( )**
  - Triggers execution of following code block
  - ( ) called "sensitivity list"
    - Describes when execution triggered

Verilog Combinational Logic

- Combinational Logic Example
- How would you describe the behavior of this function

in words?

```
If a = 1, then foo = b xor c,
else foo = b or c
```



- And in Code?

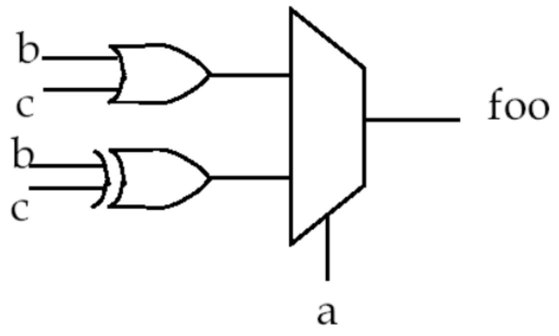
```
always@(a or b or c)
if (a) foo = b ^ c;
else foo = b | c;
```

foo can change when any of a,b or c changes SO code must be rerun whenever any of these change

Alternative Coding Style for CL

- Verilog has a short hand way to capture combinational logic
- Called “continuous assignment”

```
assign foo = a ? b^c : b | c;
```



- LHS re-evaluated whenever anything in RHS changes

```
f = a ? d : e; same as “if (a) f=d else f=e;
```

Verilog Module for Flip-flop

```
module flipflop (a,b,c,clock,Q);
input a,b,c,clock;
Output Q;
reg Q;
wire D;
always@(posedge clock)
 begin
  Q <= D;
 end
assign D = a ? b^c : b | c;
endmodule
```

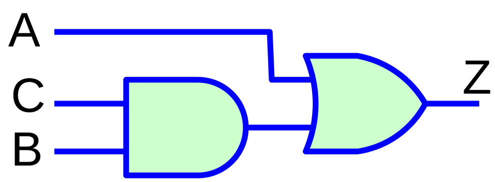
Module Name
Connected Ports
Port Declarations
Local Variable Declarations
Code Segments
endmodule

RTL Coding Styles

- That's it!
- Three coding styles
  - **`always@(???edge clock)`** → FFs and input logic
  - **`Always@(*)`** → Combinational logic (CL)
  - **`assign a = …`** → Continuous Assignment CL
- The hard part is NOT coding but DESIGN

Verilog Syntax

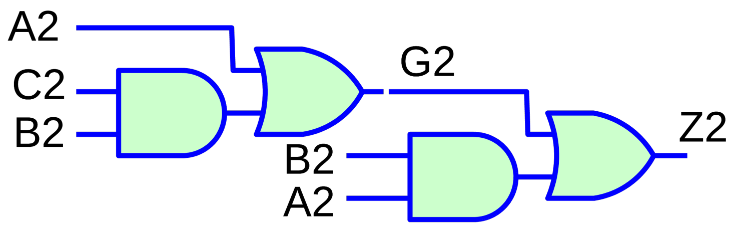
Verilog Structure



- All code are contained in modules
- Can invoke other modules
- Modules cannot be contained in another module
- A Verilog model is a descriptive model

```
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;
wire G2;
gate gate_1(G2,A2,B2,C2);
gate gate_2(Z2,G2,A2,B2);
endmodule
```



Lexicography

- Comments:

Two Types:

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

- Character Set:

0123456789ABCD..YZabcd...yz_$

Identifiers cannot start with a number or $

Data values for a single bit: 0,1,x,z

- x: an unknown value
- z: a high impedance
  - A high impedance condition occurs
    - at the output of three stage gates that are not enabled
    - if a wire is left unconnected

Truth table for predefined primitive **and** gate

and	0	1	**x**	**z**
0	0	0	0	0
1	0	1	**x**	**x**
x	0	**x**	**x**	**x**
z	0	**x**	**x**	**x**

Data Types

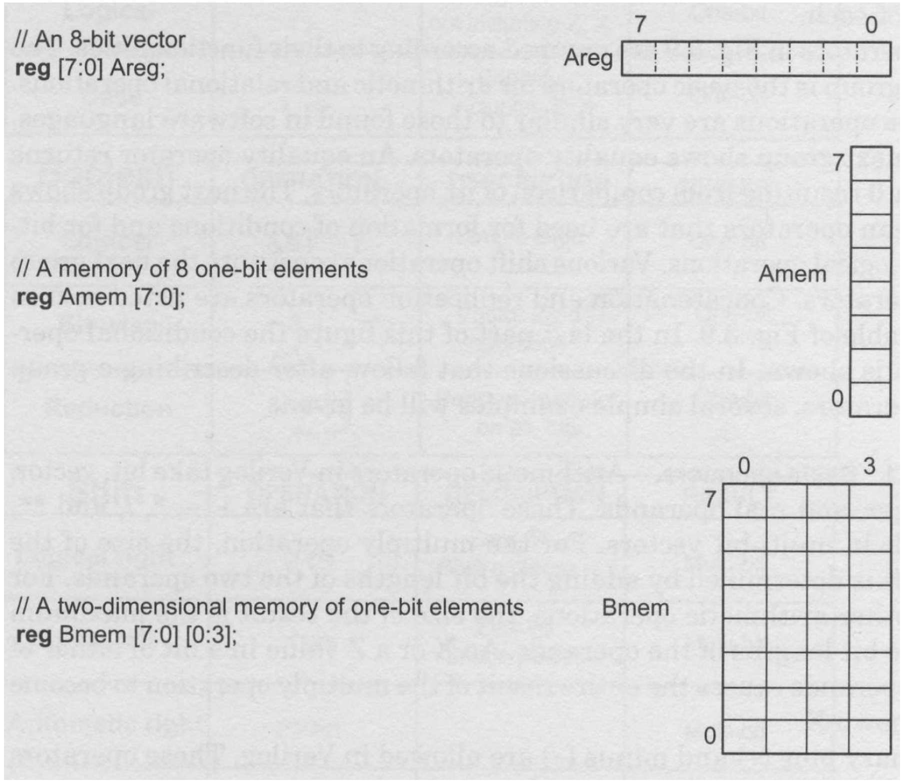
- wire
  - Used for connecting two points.
  - Used in structural code
  - Synthesizes into wires
- reg
  - Used for storing values.
  - Used in procedural code
  - May synthesize into latches, flip-flops or wires
- integer

  32-bit integer used as indexes
- input, output, inout

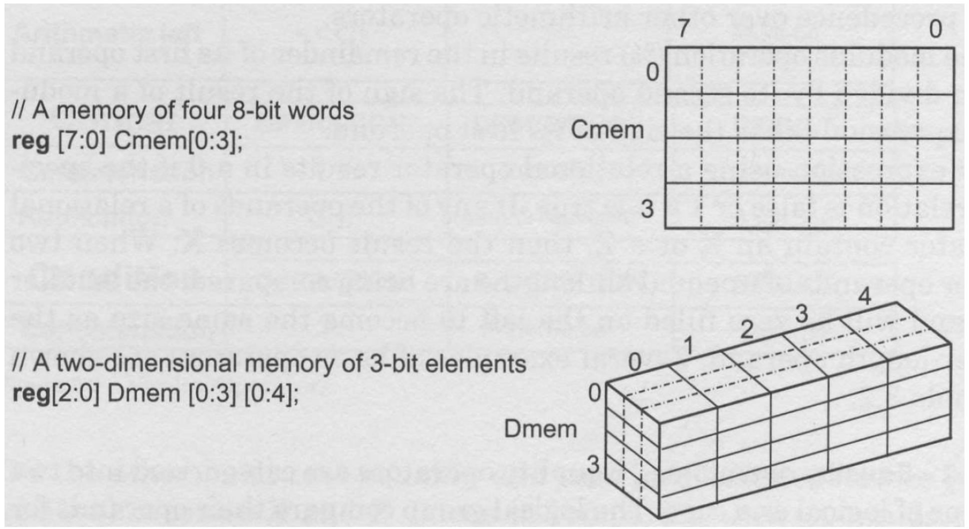
  Defines ports of a module (wire by default)

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

input a,b;
output c,d;

wire [7:0] b;

reg c,d;

integer k;
```

Vectors and arrays: declaration

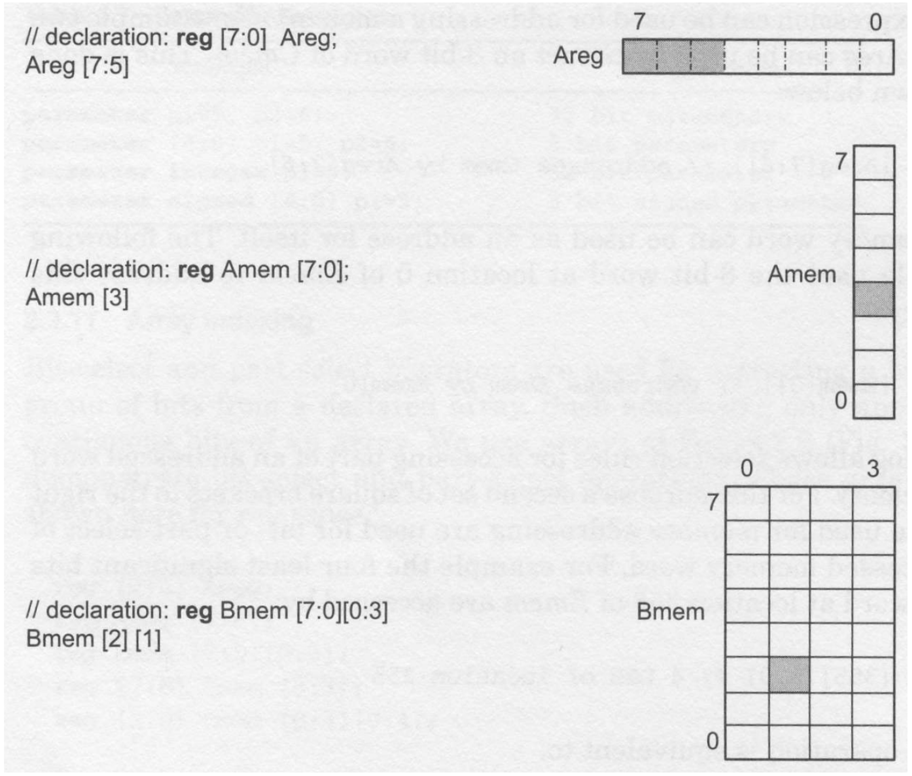


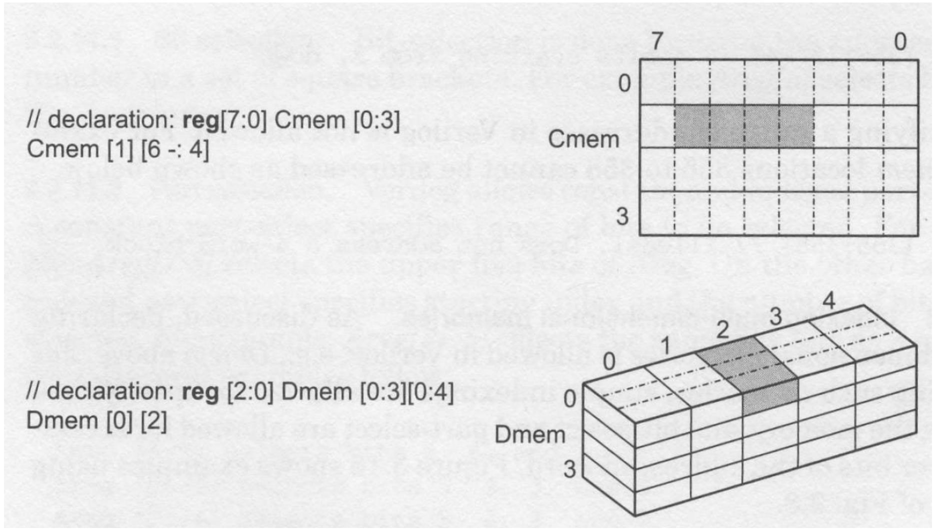
```
// An 8-bit vector
reg [7:0] Areg;

// A memory of 8 one-bit elements
reg Amem [7:0];

// A two-dimensional memory of one-bit elements
reg Bmem [7:0] [0:3];
```



```
// A memory of four 8-bit words
reg [7:0] Cmem[0:3];

// A two-dimensional memory of 3-bit elements
reg[2:0] Dmem [0:3] [0:4];
```

Vectors and arrays: selection


// declaration: reg [7:0] Areg;
Areg [7:5]
Areg
7
0
// declaration: reg Amem [7:0];
Amem [3]
Amem
7
0
// declaration: reg Bmem [7:0][0:3]
Bmem [2] [1]
Bmem
0
3
7
0


// declaration: reg[7:0] Cmem [0:3]
Cmem [1] [6 -: 4]
Cmem
7
0
0
3
// declaration: reg [2:0] Dmem [0:3][0:4]
Dmem [0] [2]
Dmem
0
1
2
3
4
0
3

Data Values

- Numbers:

Numbers are defined by number of bits

Value of 23:

```
5'b10111
5'd23
5'h17
```

- Constants:

```
wire [3:0] t,d;
assign t = 23;
assign d= 4'b0111;
```

- Parameters:

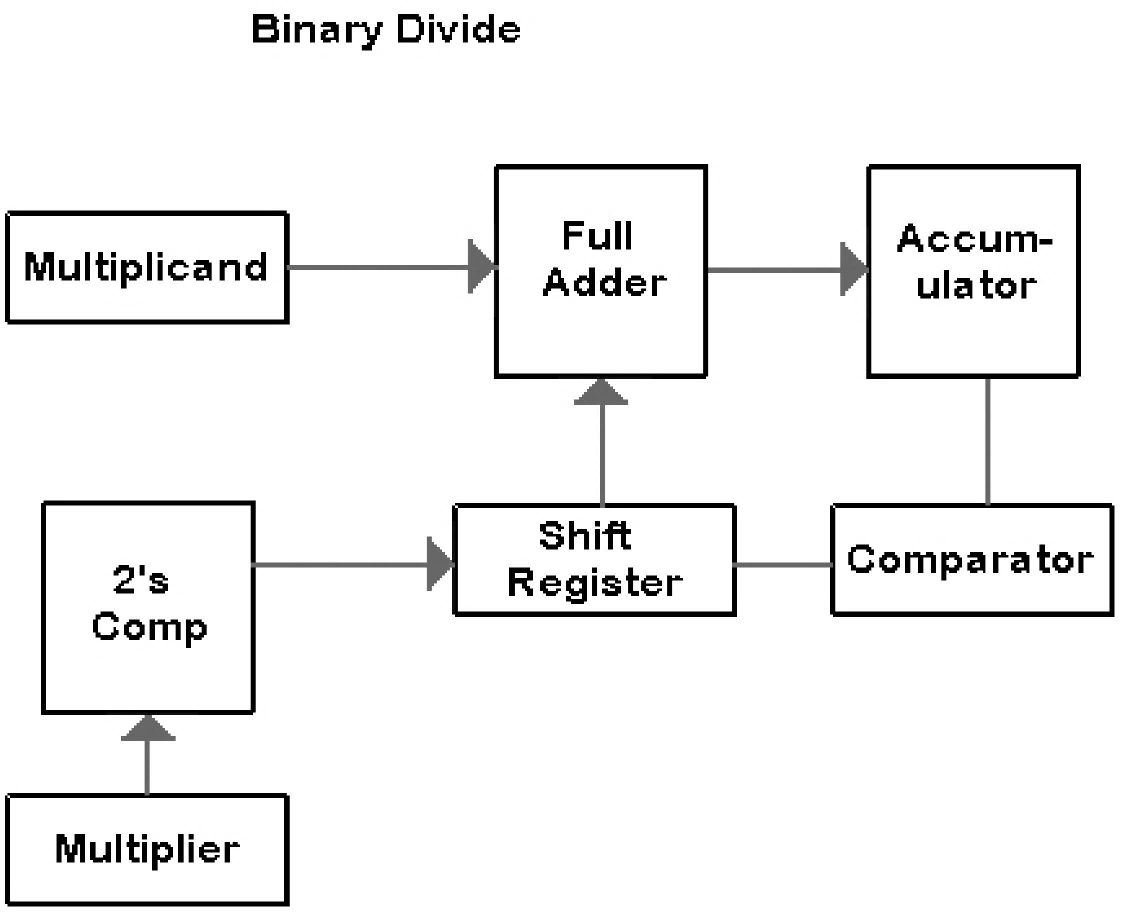
```
parameter n=4;
wire [n-1:0] t, d;

`define Reset_state = 0, state_B =1,
    Run_state =2, finish_state = 3;
if(state==`Run_state)
```

Operators

- Arithmetic:
  - *   (multiplication)
  - + (addition)
  - -   (subtraction)
  - /   (division)
  - % (modulus)
- Relational
  - <   (less than)
  - <= (less than or equal)
  - >   (greater than)
  - >= (greater than or equal)
  - == (equality)
  - != (inequality)

```
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;
```

Binary Divider Circuit



All of them are large circuits. Resulting circuit is too large for the small memory chips. You have to think all operators by their circuit designs before using them.

Operators

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

Returns 1 or 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

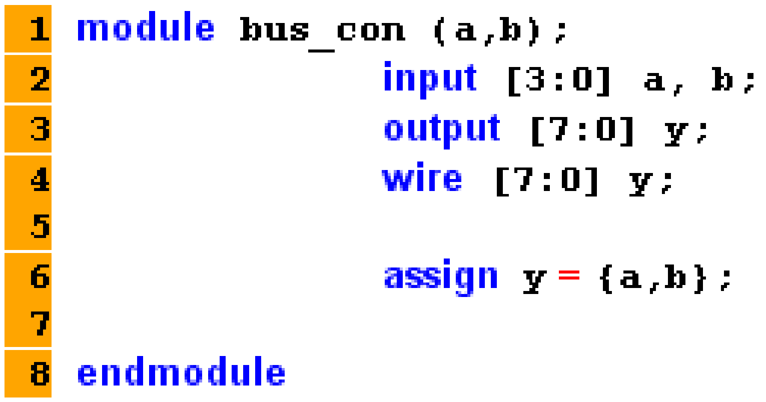
{ } (concatenation)
{ n{item} } (n fold replication of an item)

- Conditional Operator

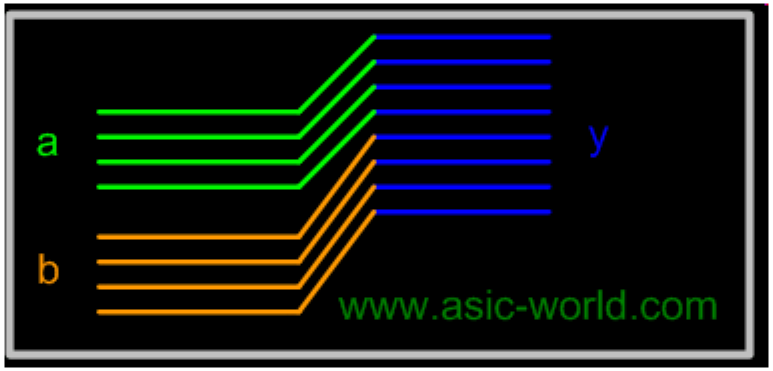
Implements if-then-else statement
(cond) **?** (result if cond true) **:** (result if cond false)

```
module sample (a, b, c, d);
input [3:0] a, b;
output [3:0] c, d;
wire z,y,carry;

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 c=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;
```

Concatenation example



```
module bus_con (a,b);
                input [3:0] a, b;
                output [7:0] y;
                wire [7:0] y;

                assign y = {a,b};

endmodule
```



Blocking vs Non-Blocking assignment

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

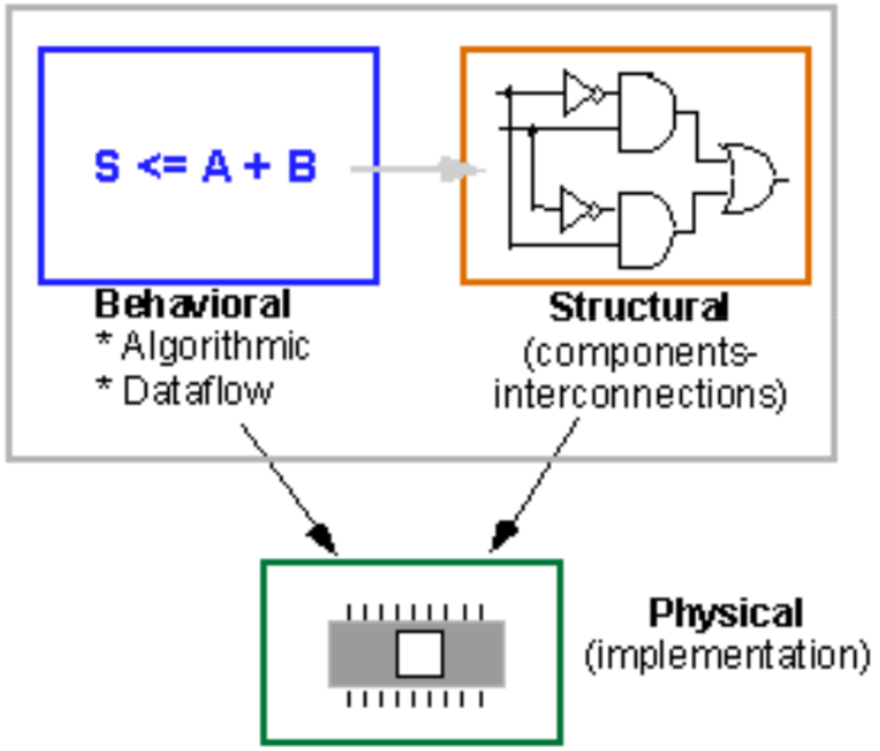
Do not mix both assignments in one procedure

Blocking vs Non-Blocking assignment

```
B = A
C = B+1
// sets C to A+1

B <= A
C <= B+1
// sets C the original value of B+1
```

Behavior Modeling

Digital system Abstractions

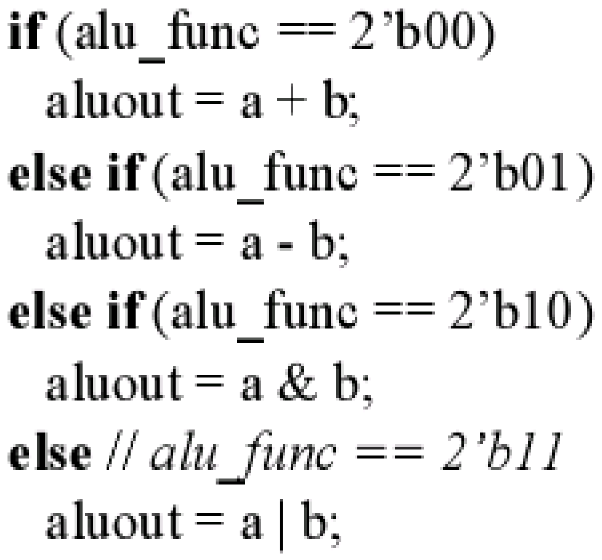


Levels of abstraction: Behavioral, Structural, Physical

if Statements

Syntax

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

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

 else
 begin
   ...statements...
 end
```

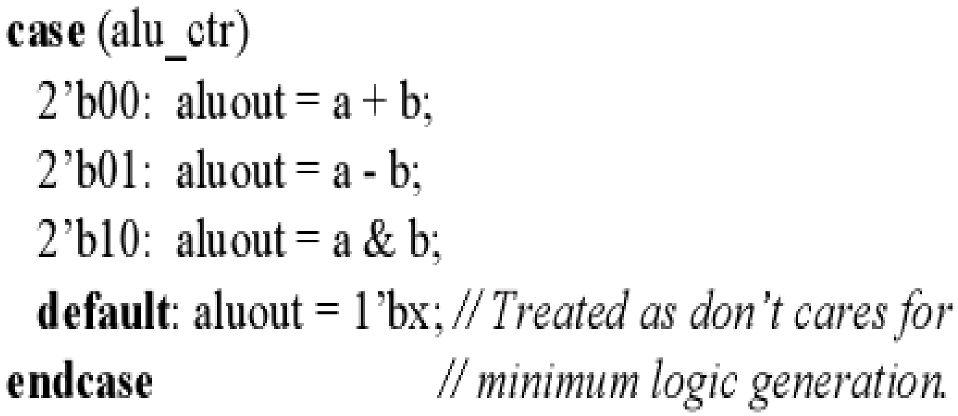


```
if (alu_func == 2'b00)
   aluout = a + b;
else if (alu_func == 2'b01)
   aluout = a - b;
else if (alu_func == 2'b10)
   aluout = a & b;
else // alu_func == 2'b11
   aluout = a | b;
```

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
```

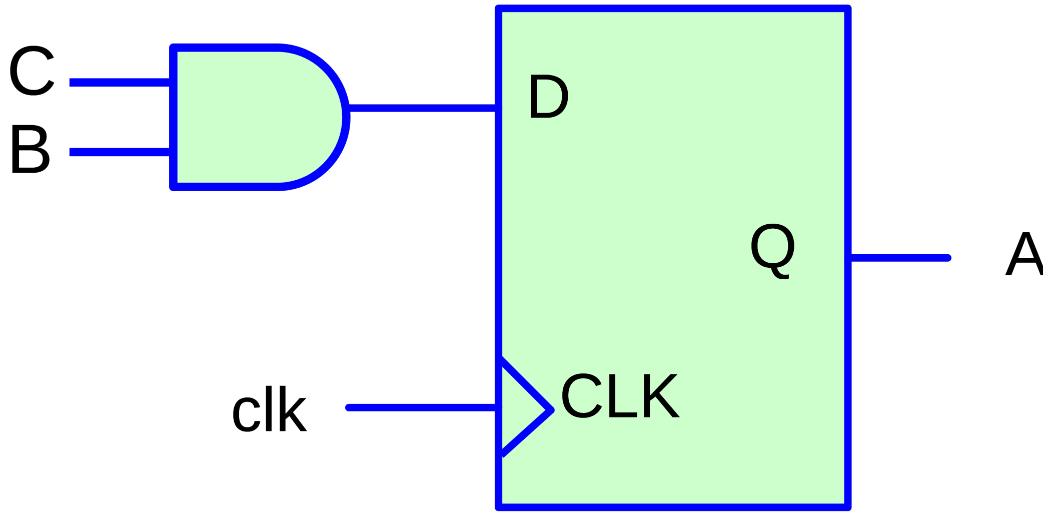
for loops cont’d

When you will code for a chip. The for loops will be unrolled and then used. Hence, the code can not be synthesized if the loop condition is a variable. It has to be a constant to be able to unroll.

Component Inference

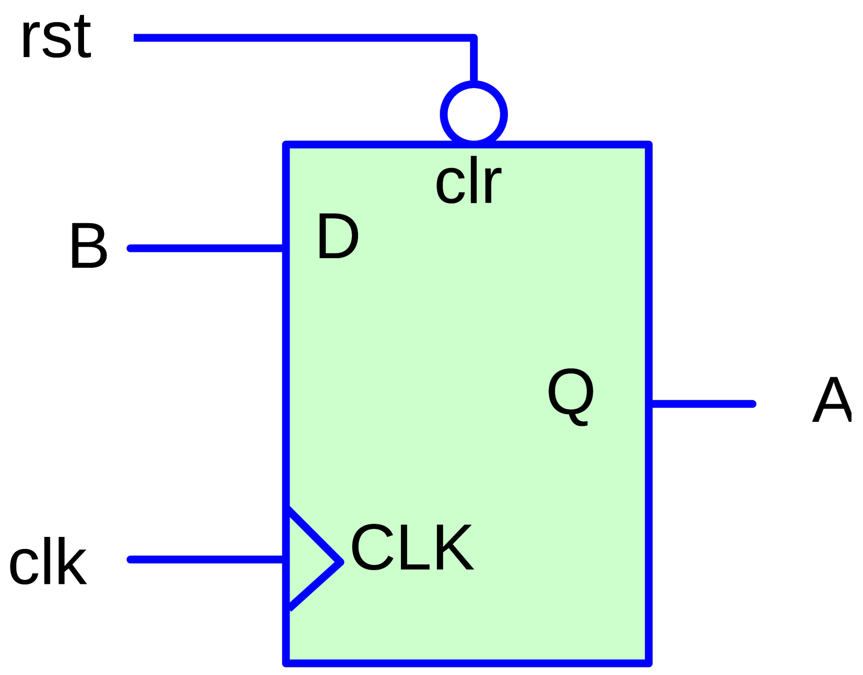
Flip-Flops

```
module ff(b, c, clk, a);

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



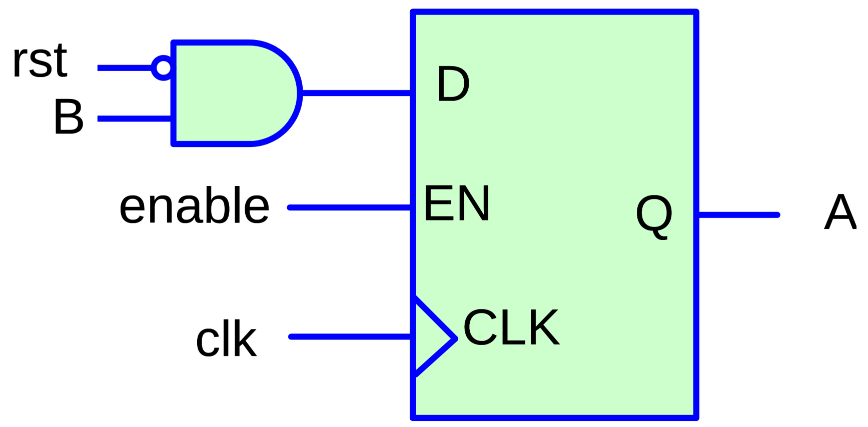
D Flip-Flop with Asynchronous Reset

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



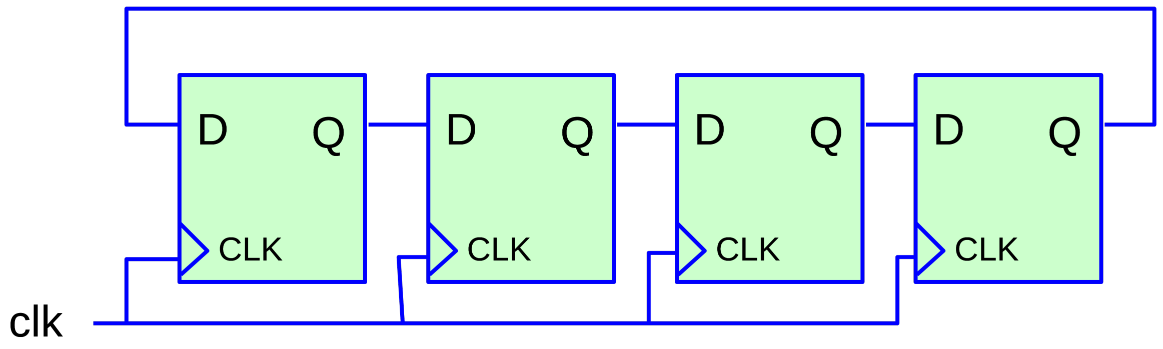
D Flip-flop with Synchronous reset and Enable

```
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[3:1]<=Q[2:0];
    Q[0]<=Q[3];
   end
```



Multiplexers

Method 1

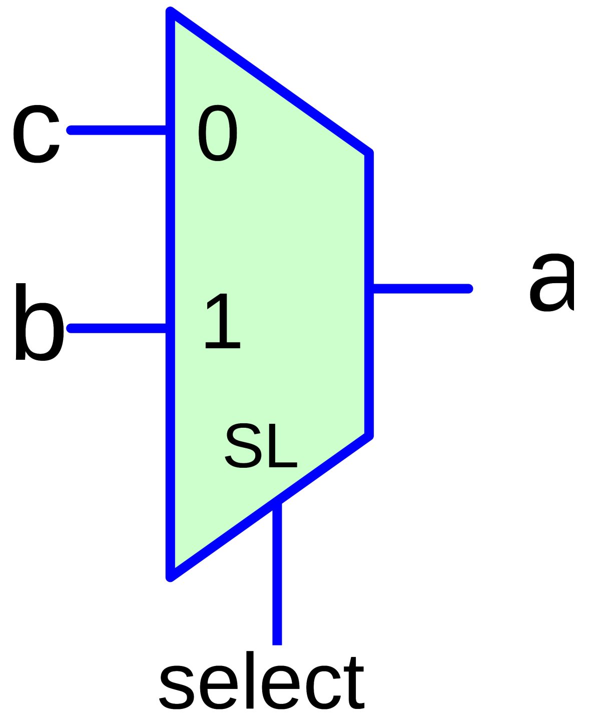
```
assign a = (select ? b : c);
```

Method 2

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

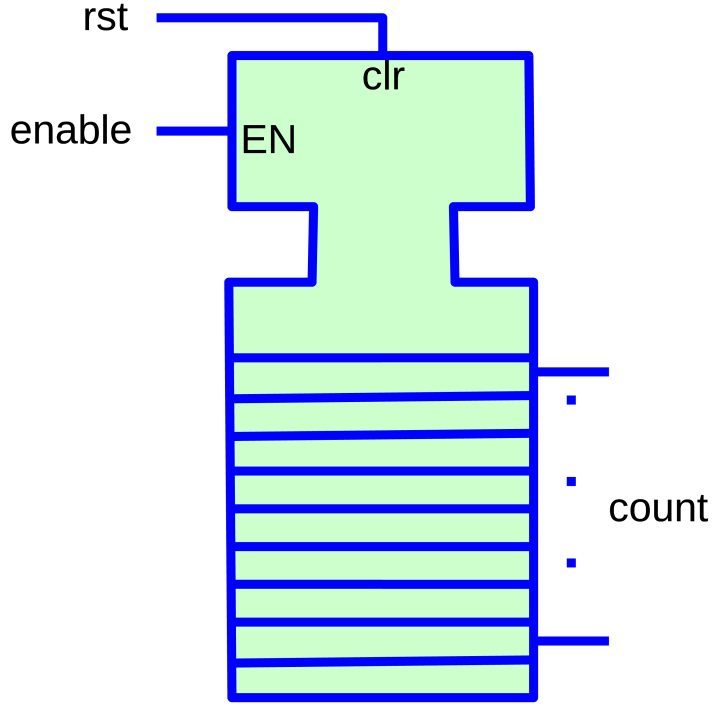
Method 2b

```
always@(select or b or c) begin
  case(select)
    1'b1: a=b;
    1'b0: a=c;
  endcase
end
```



Counters

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



Verilog Coding Guidelines

Rule #1

If the procedure has several paths, every path must evaluate all outputs

- **<u>Method1</u>:**

Set all outputs to some value at the start of the procedure.
Later on different values can overwrite those values.

```
always @(...
begin
  x=0;y=0;z=0;
  if (a) x=2; elseif (b) y=3; else z=4;
End
```

- **<u>Method2</u>:**

Be sure every branch of every if and case generate every output

```
always @(...
begin
  if (a) begin       x=2; y=0; z=0; end
  elseif (b) begin x=0; y=3; z=0; end
  else begin       x=0; y=0; z=4; end
end
```

Rule #2

All inputs used in the procedure must appear in the trigger list

- **Right-hand side variables:**

Except variables both calculated and used in the procedure.

```
always @(a or b or c or ~~x or y~~)
  begin
    x=a; y=b; z=c;
    w=x+y;
  end
```

Rule #3

All possible inputs used to control statements must be covered

- End all case statements with the default case whether you need it or not.

```
case(state)
  ...
  default: next_state = reset;
endcase
```

- Do not forget the self loops in your state graph

```
if(a|b&c) next_state=S1;
elseif(c&d) next_state=S2;
else next_state=reset;
```

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);

- 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
1. Set the clk signal
1. Send test vectors
1. 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
  $monitor(“Time: %d Output z: %b”, $time, z);
  #1 x=0;
  #400 x=1;
  $display(“Output z: %b”, z);
  #100 x=0;

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

One last slide

Coding guidelines

- Don't mix level and edge sensitive elements in the same always block
- Avoid mixing positive and negative edge-triggered flip-flops
- Use continuous assign statements for simple combinational logic
- Use nonblocking for sequential and blocking for combinational logic
- Don't mix blocking and nonblocking assignments in the same always block
- Be careful with multiple assignments to the same variable
- Define all if-else or case statements explicitly