SystemVerilog Procedural Blocks

Procedural Blocks

- *always*
- *always_comb*
- *always_latch*
- *always_ff*
- *final*

always block

- *The Verilog **always** procedural block is used for general purpose modeling.*
- *At the RTL level, the **always** procedural block can be used to **model :***
- ***combinational logic**, **latched logic**, and **sequential logic***

```
always @(a, b)
  begin
        sum = a + b;
        diff = a - b;
        prod = a * b;
  end
```

```
always @(posedge clk)
if(reset) q<=d;
else q<=0;

always @(clk or d)
If(clk) q<=d;
```

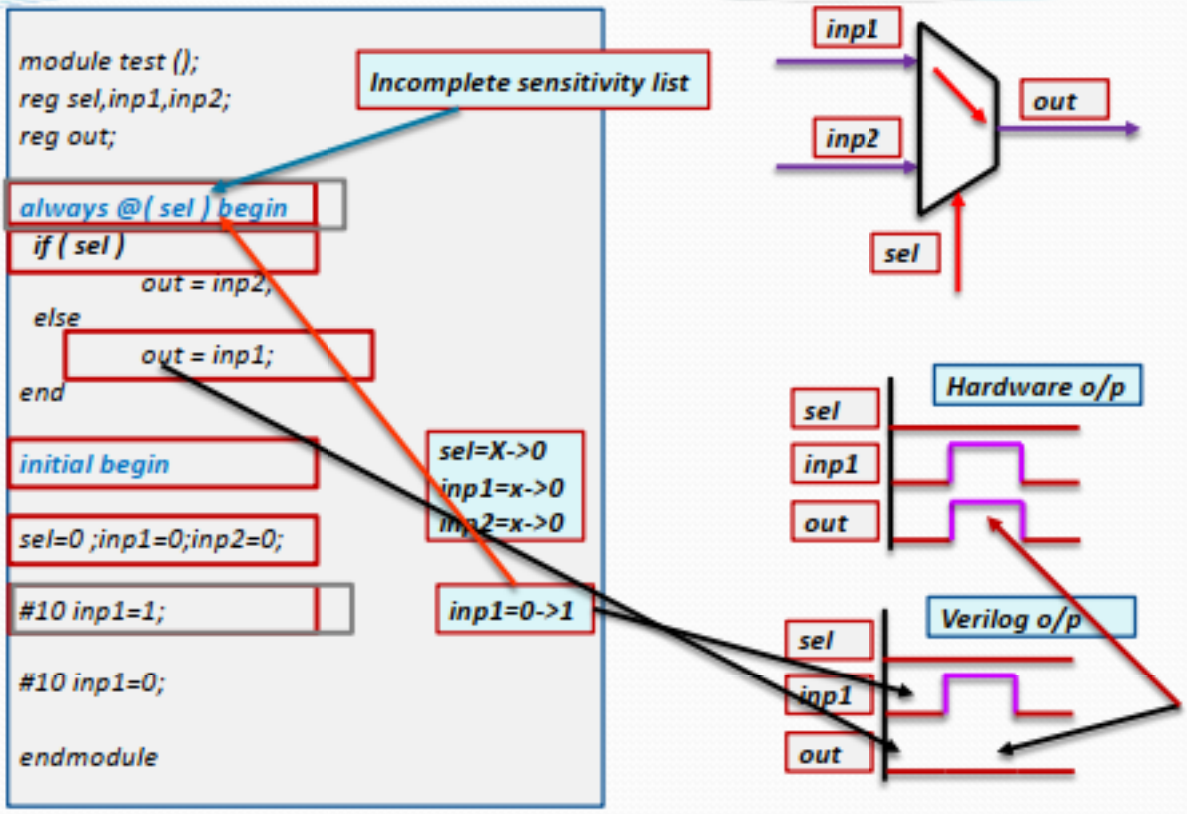
```
always @(a, en)
if (en) y = a;
else  y=0;
```

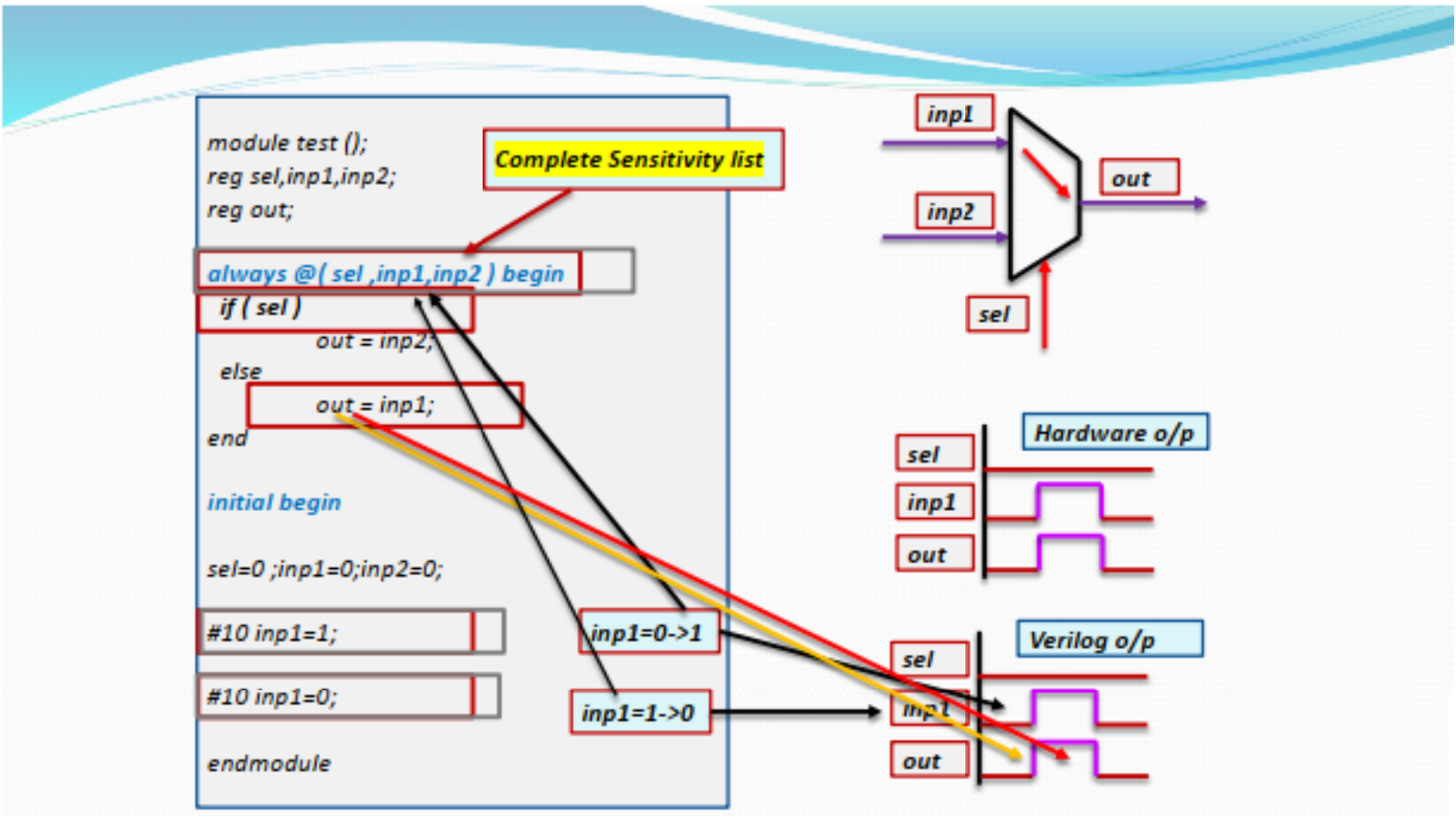

latch would be required to realize the procedural block's functionality in hardware

- *With SystemVerilog, this same example could be written as follows:*

```
always_comb
if (en) y = a;
```


***Designer's intent** was to **model combinational logic.** So, tools can issue a warning that a latch would be required*


module test ();
reg sel,inp1,inp2;
reg out;
always @(sel) begin
if (sel)
out = inp2;
else
out = inp1;
end
initial begin
sel=0 ;inp1=0;inp2=0;
#10 inp1=1;
#10 inp1=0;
endmodule
Incomplete sensitivity list
sel=X->0
inp1=x->0
inp2=x->0
inp1=0->1
inp1
inp2
sel
out
Hardware o/p
sel
inp1
out
Verilog o/p
sel
inp1
out


module test ();
reg sel,inp1,inp2;
reg out;
Complete Sensitivity list
always @(sel ,inp1,inp2) begin
if (sel)
out = inp2;
else
out = inp1;
end
initial begin
sel=0 ;inp1=0;inp2=0;
#10 inp1=1;
#10 inp1=0;
endmodule
inp1=0->1
inp1=1->0
inp1
inp2
sel
out
Hardware o/p
sel
inp1
out
Verilog o/p
sel
inp1
out

always_comb

```
always_comb
if (!mode)
y = a + b;
else
y = a – b;
```

1) The **always_comb** procedural block is used to **indicate the intent to model combinational logic**

2) Infers sensitivity list automatically. (mode,a,b)

3) Automatic evaluation at time zero

4) The **always_comb** procedural block also requires that **variables on the left-hand side of assignments cannot be written to by any other procedural block**

always_comb

- The **always_comb** procedural block also requires that **variables on the left-hand side of assignments cannot be written to by any other procedural block**

```
always @(sel or inp1 or inp2)
begin
   if(sel)
      out = inp1;
   else
      out = inp2;
end

always @(sel)
   out = !inp3;
```

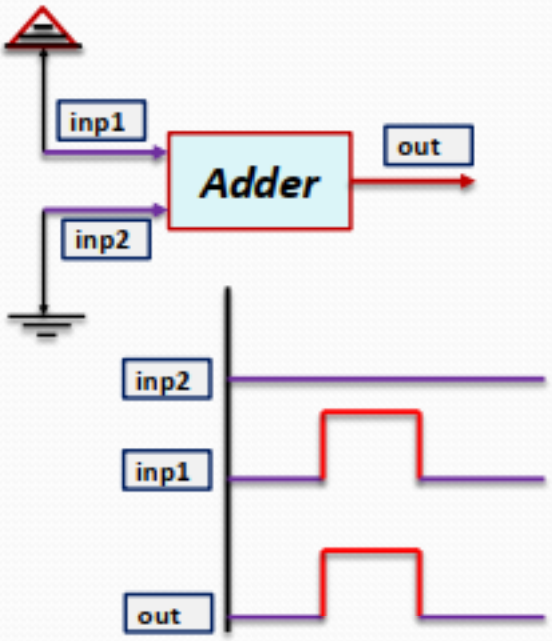
```
always_comb begin
   if(sel)
      out = inp1;
   else
      out = inp2;
end

always @(sel)
   out = !inp3;
```

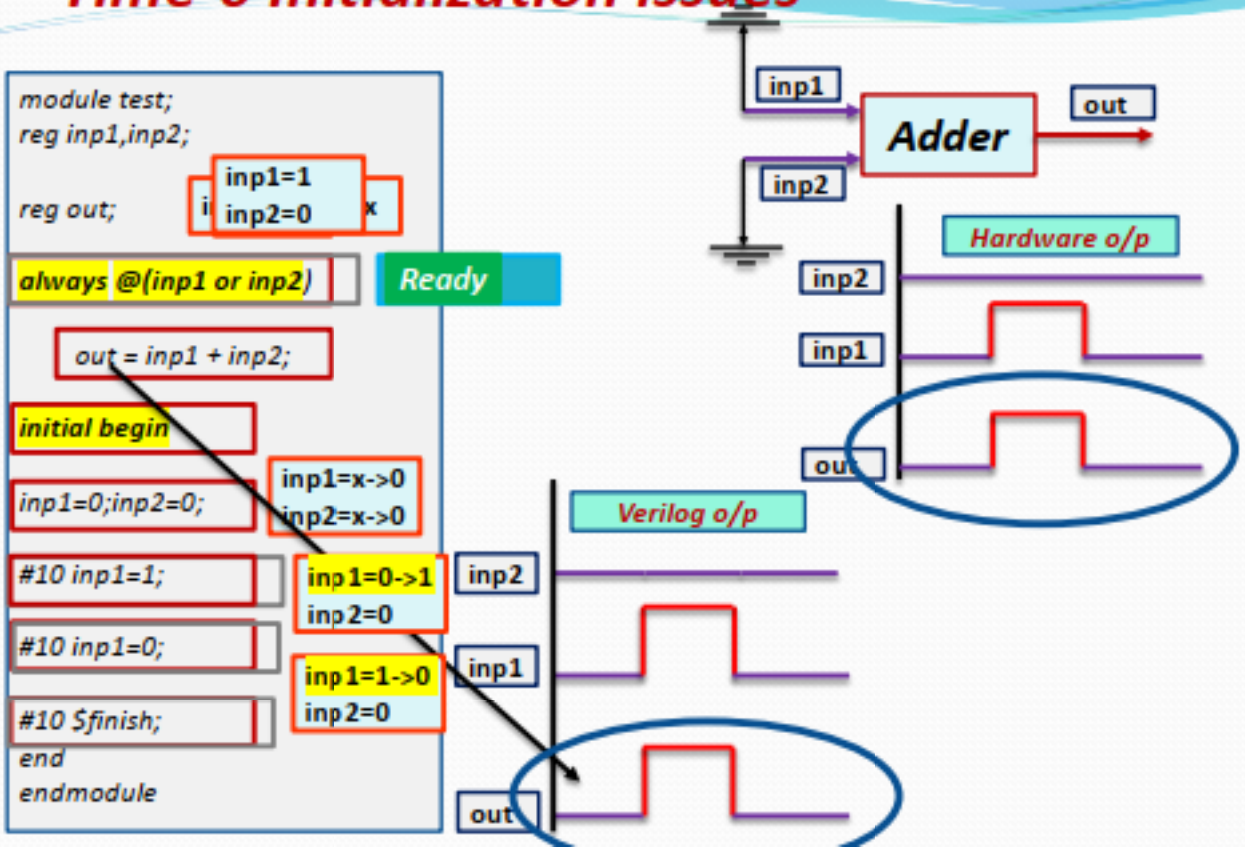
```
always_comb begin
   if(sel)
      out = inp1;
   else
      out = inp2;
end

always_comb
   out = !inp3;
```

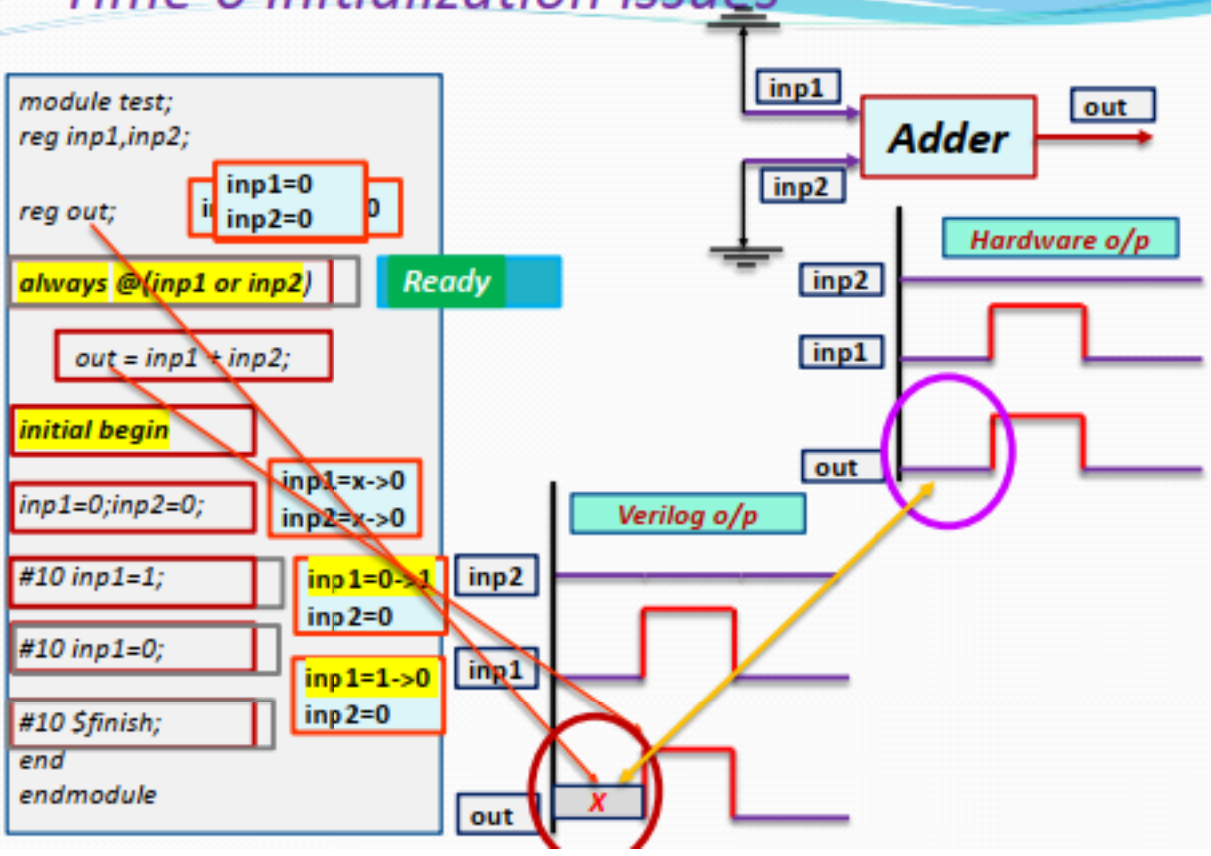
Time-0 Initialization issues



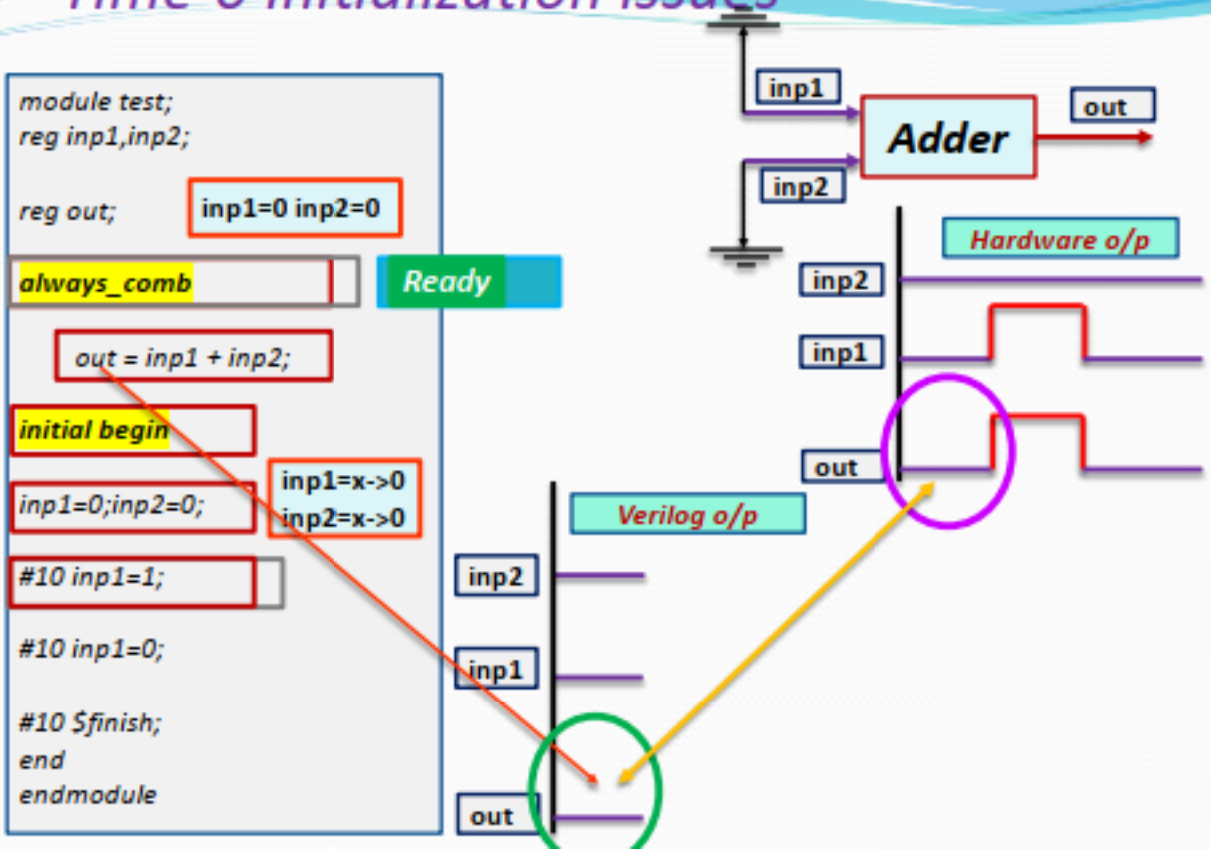
Time-0 Initialization issues



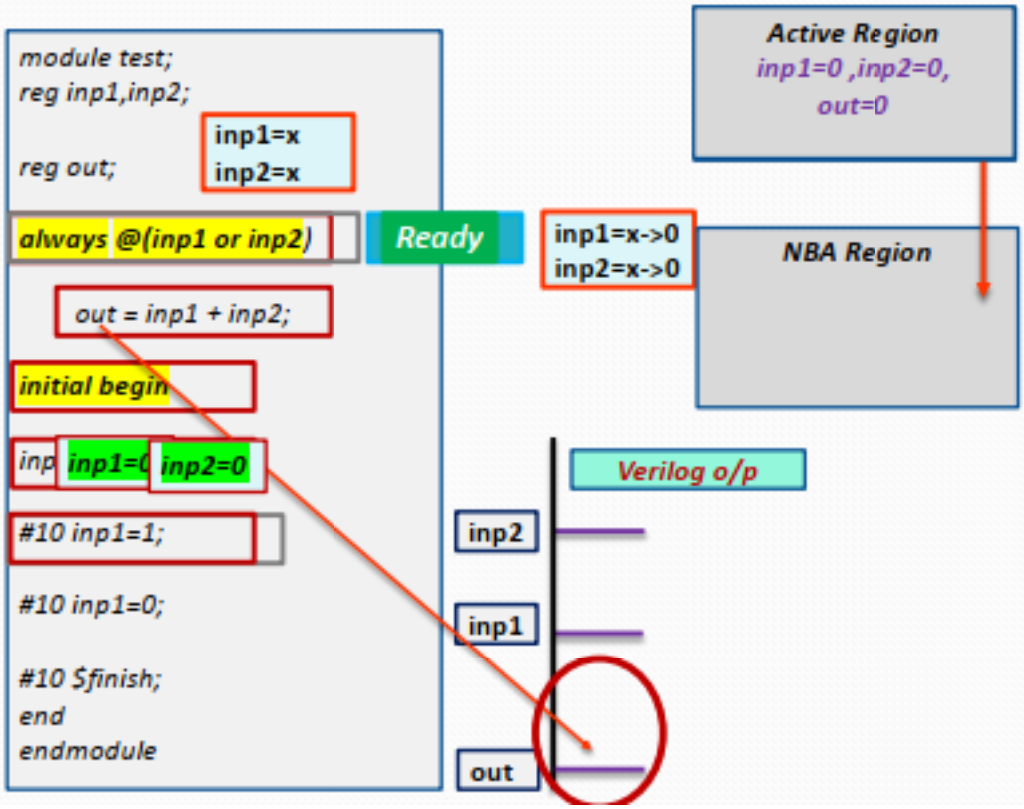
Time-0 Initialization issues



Time-0 Initialization issues



Time-0 Initialization issues



always_latch

- The always_latch procedural block is used to indicate that the **intent of the procedural block is to model latched-based logic**.
- always_latch **infers its sensitivity list**, just like always_comb.

Ex: **always_latch**

```
if (enable) q <= d;
```

- Variables written in an always_latch procedural block cannot be written by any other procedural block.

- The **always_latch** procedural blocks also **automatically execute once at time zero**

always_ff

- *The **always_ff** specialized procedural block indicates that the designer's intent is to model synthesizable sequential logic behavior.*

Ex: always_ff @(**posedge** clock, **negedge** resetN)

```
if (!resetN) q <= 0;
else q <= d;
```

- A ***sensitivity list must be specified*** with an **always_ff** procedural block.
- This allows the engineer to model either synchronous or asynchronous set and/or reset logic, based on the contents of the sensitivity list.

final block

- The **final** procedure is like an **initial** procedure, defining a procedural block of statements, except that **it occurs at the end of simulation time and executes without delays.**
- A **final** procedure is typically used to display statistical information about the simulation
- A **final** procedure executes when simulation ends due to an explicit or implicit call to $finish.

Ex:

```
final begin
$display("Number of cycles executed %d",$time/period);
$display("Test Passed ");
end
```

```
module test;

dut dut_inst(.....);

initial begin
//stimulus generation code
If(error==1)
$finish;
end

initial begin
//stimulus monitoring code
If(dut_out==5)
$finish;
end

initial begin
//protocol monitoring code
If(violations==5)
$finish;
end

final begin
$display("Number of cycles executed %d",$time/period);

If(error !=0)
  $display("Test terminated due to error condition ");

If(violations !=0)
  $display("Test terminated due to Protocol violations");

$display("Test PASSED"
end
```

Thank you

always_comb compared to always @*

- *The SystemVerilog always_comb procedure differs from always @* in the following ways:*

- *always_comb automatically executes once at time zero,*
- *whereas always @* waits until a change occurs on a signal in the inferred sensitivity list.*

- *always_comb is **sensitive to changes within the contents of a function**,*
- *whereas always @* is **only sensitive to changes to the arguments of a function**.*

- *Variables on the left-hand side of assignments within an always_comb procedure, including variables from the contents of a called function, shall not be written to by any other processes,*
- *whereas always @* permits multiple processes to write to the same variable.*

- *Variable that is read and written in always_comb won't be added to sensitivity list but always@* will add to sensitivity list.*

- *Statements in an always_comb **shall not include** those that block, **have blocking timing or event controls, or fork-join statements.***

- *always_comb is sensitive to expressions in immediate assertions within the procedure and within the contents of a function called in the procedure,*
- *whereas always @* is sensitive to expressions in immediate assertions within the procedure only.*

```
always @(*) // equivalent to @(a or b or c or d or f)
y = (a & b) | (c & d) | myfunction(f);

always @* begin // equivalent to @(a or b or c or d or tmp1 or tmp2)
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end

always @* begin // equivalent to @(b)
@(i) kid = b; // i is not added to @*
end

always @* begin // equivalent to @(a or b or c or d)
x = a ^ b;
@* // equivalent to @(c or d)
x = c ^ d;
end
```

```
always @* begin // same as @(a or en)
y = 8'hff;
y[a] = !en;
end

// same as @(state or go or ws)
always @* begin
next = 4'b0;
case (1'b1)
state[IDLE]:   if (go) next[READ] = 1'b1;
               else next[IDLE] = 1'b1;
state[READ]: next[DLY ] = 1'b1;
state[DLY ]:   if (!ws) next[DONE] = 1'b1;
               else next[READ] = 1'b1;
state[DONE]: next[IDLE] = 1'b1;
endcase
end
```