ECE 4514
Digital Design II
Spring 2008

Behavioral Modeling II: Conditionals and Loops

A *Language* Lecture

Patrick Schaumont

Behavioral Modeling so far

- Procedural statements (lecture 2 and on)
  - Always and Initial constructs
  - Sequential block statements
- Procedural Assignments
  - Blocking assignments (lecture 2 and on)
  - Non-blocking assignments (lecture 10)
- Event control statements (lecture 5)
  - @
  - wait
  - *#NNN*

Today

- Conditional statements
  - if-then-else
  - case
- Loop statement
- Parallel block statements, named block statements
- Plus a few useful *structural* techniques
  - Generate construct
- Palnitkar: 7.4 - 7.10
- IEEE 1364: 9.4 - 9.8, 12.4

Today

- Conditional statements
  - if-then-else
  - case
- Loop statement
- Parallel block statements, named block statements
- Plus a few useful *structural* techniques
  - Generate construct
- Palnitkar: 7.4 - 7.10
- IEEE 1364: 9.4 - 9.8, 12.4

Conditional statement

```
reg [7:0] a, b, max;

always @(posedge clk) begin
   max = b;
   if (a > b)
      max = a;
end

always @(posedge clk) begin
   if (a > b)
      max = a;
   else
      max = b;
end
```


True if non-zero
False if zero, X or Z

Conditional statement

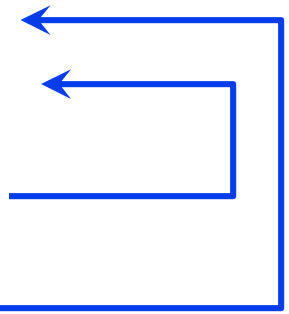
```
reg [7:0] a, b, max;

always @(posedge clk) begin
   if (a > b) begin
      max = a;
      ... other statement ..
   end
end
```

Use begin .. end to include multiple statements

Nested if-then-else

- Else associates with the innermost if

```
reg [7:0] a, b, c, max;

always @(posedge clk) begin
   if (b > a)
     if (b > c)
       max = b;
     else
       max = c;
   else
     max = a;
end
```



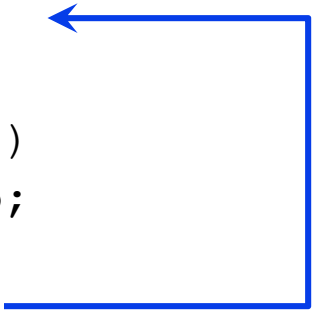
Note that indentation does not imply association.
Verilog is context-free.

How to change this association ?

Nested if-then-else

- Use begin--end to change association

```
reg [7:0] a, b, c, max;

always @(posedge clk) begin
   if (b > a)
     begin
     if (b > c)
       max = b;
     end
   else
     max = a;
end
```



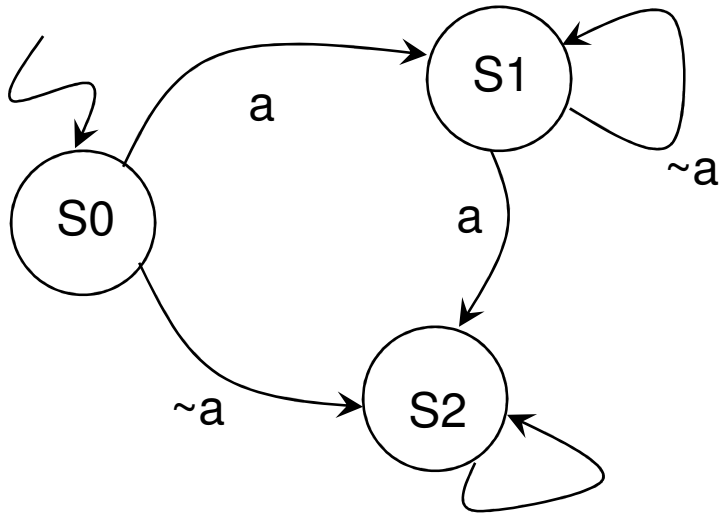
Multiway decisions: if-else-if

- Frequently used as a case-decoding statement

```
reg [7:0] a, b, c, max;

always @(posedge clk) begin
   if (a < 10) begin
     // ... when a < 10
   end
   else if (a < 20) begin
     // ... when a >= 10 and a < 20
   end
   else if (a < 30) begin
     // .. when a >= 20 and a < 30
   end
   else begin
     // .. when a >= 30
   end
end
```

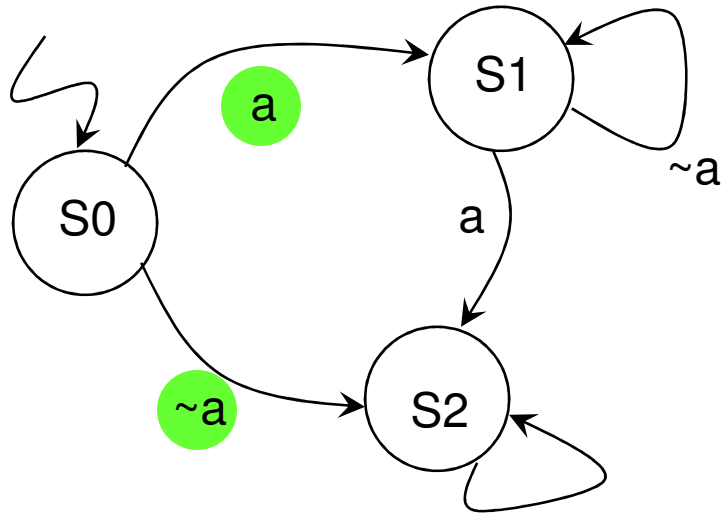
Application: Next-state logic for FSM Design

```
reg a;
reg [1:0] state;
parameter s0 = 2'b00,
          s1 = 2'b01,
          s2 = 2'b10;

always @(posedge clk) begin
   if (state == s0)
     if (a)
        state <= s1;
     else
        state <= s2;
   else if (state == s1)
     if (a)
        state <= s2;
     else
        state <= s1;
   else if (state == s2)
     state <= s2;
   else
     state <= s0;
```



Application: Next-state logic for FSM Design

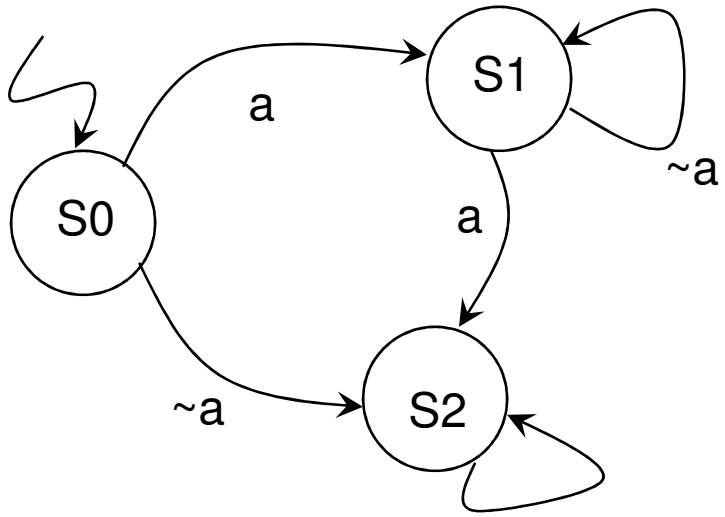
```
reg a;
reg [1:0] state;
parameter s0 = 2'b00,
          s1 = 2'b01,
          s2 = 2'b10;

always @(posedge clk) begin
   if (state == s0)
     if (a)
        state <= s1;
     else
        state <= s2;
   else if (state == s1)
     if (a)
        state <= s2;
     else
        state <= s1;
   else if (state == s2)
     state <= s2;
   else
     state <= s0;
```



By matching up if-else conditions, you can be sure that the set of state transitions is logically complete.

Case statement

```
reg a;
reg [1:0] state;
parameter s0 = 2'b00,
          s1 = 2'b01,
          s2 = 2'b10;

always @(posedge clk)
   case (state)
     s0: if (a)
           state <= s1;
         else
           state <= s2;
     s1: if (a)
           state <= s2;
         else
           state <= s1;
     s2: state <= s2;
     default state <= s0;
   endcase
```



Case statement

```
reg a;
reg [1:0] state;
parameter s0 = 2'b00,
          s1 = 2'b01,
          s2 = 2'b10;

always @(posedge clk)
   case (state)
     s0: if (a)
           state <= s1;
         else
           state <= s2;
     s1: if (a)
           state <= s2;
         else
           state <= s1;
     s2: state <= s2;
     default: state <= s0;
   endcase
```

Two differences with if-then-else

1. if-then-else conditions can be more general. Each if-then-else leg can test a different expression
2. case statement performs *exact* matching*, including x and z
   If-then-else always returns false when matching x or z

* cfr case-equality operator ===

Case statement

```
reg a;
reg [1:0] state;
parameter s0 = 2'b00,
          s1 = 2'b01,
          s2 = 2'b10;

always @(posedge clk)
   case (state)
     s0: if (a)
           state <= s1;
         else
           state <= s2;
     s1: if (a)
           state <= s2;
         else
           state <= s1;
     s2: state <= s2;
     default: state <= s0;
   endcase
```

Default is optional. However, if no case label matches, then no entry of case will execute.

Another example

```
reg a, b;

always @(posedge clk)
   case ({a,b})
     2'b00:   // will match a = 0, b = 0
     2'bx0:   //              a = x, b = 0
     2'b10:   //              a = 1, b = 1
     2'b01,
     2'b0x:   //              a = 0, b = 1 or x
     2'b00:   //              a = 0, b = 0
     default: // all other cases
   endcase
```

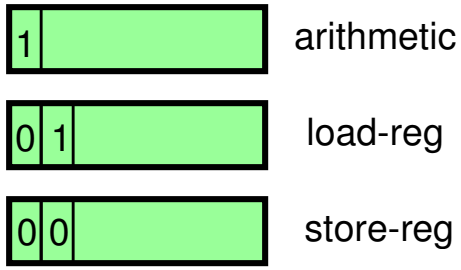
Case labels are tested in sequence
Do you see anything special ?

Case statement with *don't care*

- casez allows to express don't care with 'z' or '?'

```
reg [7:0] instr;

always @(posedge clk)
   casez (instr)
     7'b1???????: // arithmetic
     7'b01??????: // load-reg
     7'b00??????: // store-reg
   endcase
```

Example instruction format



OR:

```
   casez (instr)
     7'b1zzzzzzz: // arithmetic
     7'b01zzzzzz: // load-reg
     7'b00zzzzzz: // store-reg
   endcase
```

Case statement with *don't care*

- casex allows to express don't care with 'x' *or* 'z'
- However, please use casez for all practical applications involving don't cares

Constant-case

- Search for a pattern '10' in a word, starting from the lsb

```
reg [4:0] thebits;

always @(posedge clk)
   case (2'b10)
     thebits[1:0]: // it's on the lsb position
     thebits[2:1]: // it's on the lsb+1 position
     thebits[3:2]: // it's on the lsb+2 position
   endcase
```

Today

- Conditional statements
  - if-then-else
  - case
- Loop statement
- Parallel block statements, named block statements

- Plus a few useful *structural* techniques
  - Generate construct

- Palnitkar: 7.4 - 7.10
- IEEE 1364: 9.4 - 9.8, 12.4

Four Different loop statements

- forever:
  - does what it says
- repeat:
  - repeat a fixed number of times
- while:
  - conditionally repeat a number of times
- for:
  - initialize variable;
    test loop condition;
    run body;
    loop counter update

Forever

- Note that there is no break statement ..

```
always @(posedge clk)
   forever
     // OMG! How do I get out?
```

  - Can be terminated by creating a *named block* and a *disable* statement

Repeat

- Sample expression at start. If non-zero, do this N number of times. If zero, exit immediately

```
module clock_divider(clk, clkdiv);
   input clk;
   output clkdiv;
   reg clkdiv;

   initial clkdiv <= 0;

   always begin
      repeat(5)
         @(posedge clk)
             ;
       clkdiv <= ~clkdiv;
   end
endmodule
```

While

```
module countones(num, in);
  input  [7:0] in;
  output [7:0] num;
  reg    [7:0] num;
  reg    [7:0] wrk;

  initial begin
    num = 0;
    wrk = in;
    while (wrk) begin
      if (wrk[0])
        num = num + 1;
      wrk = wrk >> 1;
    end
  end
endmodule
```

For

```
module countones(num, in);
  input  [7:0] in;
  output [7:0] num;
  reg    [7:0] num, a;
  reg    [7:0] wrk;

  initial begin
    num = 0;
    wrk = in;
    for (a=0; a < 8; a = a + 1)
      if (wrk[a])
        num = num + 1;
  end
endmodule
```

Procedural Timing Controls

- @, wait, #: we discussed these in Lecture 5. What follows are some further details

- Timing of a procedure is affected with event controls or with delay controls
  - @ is an *event control*
  - # is a *delay control*
  - wait is the combination of a while loop and an event control

Event control

- posedge (**P**) and negedge (**N**)

From \ To	0	1	X	Z
0		P	P	P
1	N		N	N
X	N	P		
Z	N	P		

- Events are combined with **or** or **,**

```
@(posedge a or b or c)

@(a, posedge b, negedge c)
```

Implicit Event Expression List

- **Automatically select RHS of all expressions in a block**

```
always @(a, b, c, e)
  begin
  b = a + 1;
  c = b / 2;
  d = b + c + a - e;
  end
```

```
always @*
  begin
  b = a + 1;
  c = b / 2;
  d = b + c + a - e;
  end
```

same as @(a, b, c, e)

Implicit event lists are useful

- What RHS are included here?

```
always @*
  begin
  tmp1 = a & b;
  tmp2 = c & d;
  end
```

```
always @*
  begin
  @(n) a = b;
  end
```

```
always @*
  begin
  y = 8'hff;
  y[a] = !en;
  end
```

Implicit event lists are useful

- What RHS are included here?

```
always @*
  begin
  tmp1 = a & b;
  tmp2 = c & d;
  end
```

@(a, b, c, d)

```
always @*
  begin
  @(n) a = b;
  end
```

@(b)

```
always @*
  begin
  y = 8'hff;
  y[a] = !en;
  end
```

@(a, en)

Implicit event lists are useful

- What RHS are included here?
  - assume CAPITAL variables are constants

```
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
```

Implicit event lists are useful

- What RHS are included here?
  - assume CAPITAL variables are constants

@(go, ws, state)

```
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
```

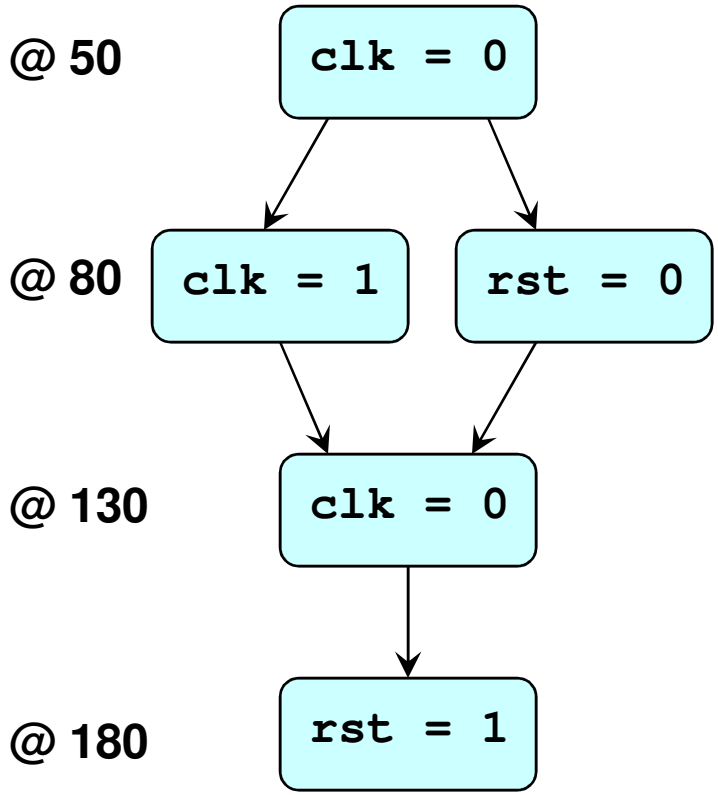
Today

- Conditional statements
  - if-then-else
  - case
- Loop statement
- Parallel block statements, named block statements
- Plus a few useful *structural* techniques
  - Generate construct
- Palnitkar: 7.4 - 7.10
- IEEE 1364: 9.4 - 9.8, 12.4

Parallel Block and Sequential Block

- Sequential Block: `begin .. end`

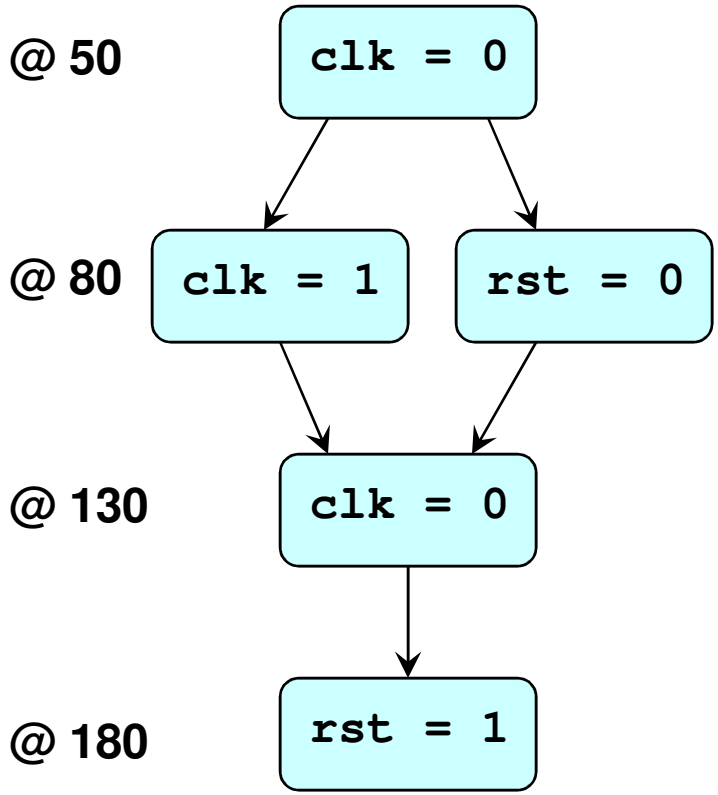
```
initial begin
  #50 clk = 0;
  #30 clk = 1;
      rst = 0;
  #50 clk = 0;
  #50 rst = 1;
end
```



Parallel Block and Sequential Block

- Parallel Block: `fork .. join`

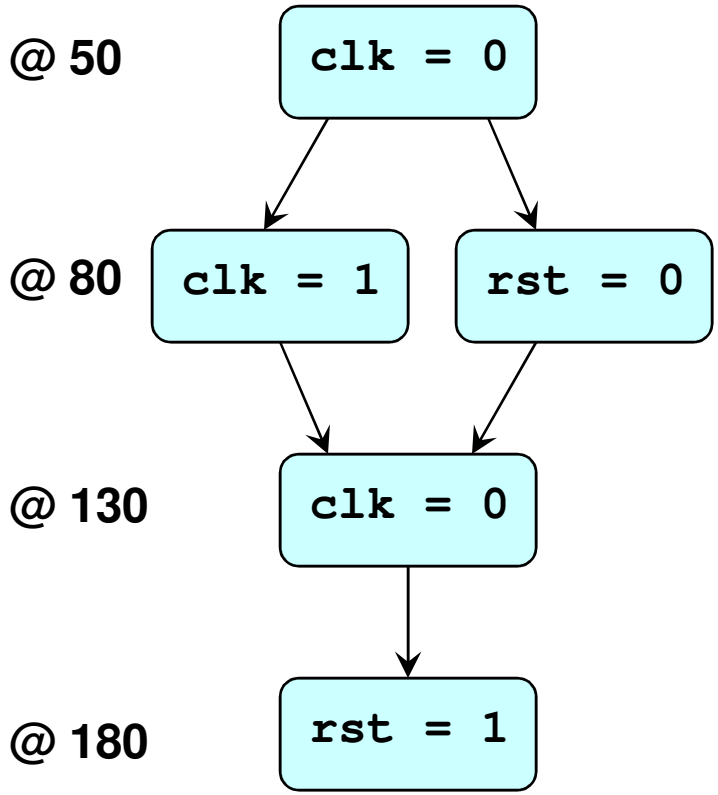
```
initial fork
  #50 clk = 0;
  #80 clk = 1;
  #80 rst = 0;
 #130 clk = 0;
 #180 rst = 1;
join
```



Parallel Block and Sequential Block

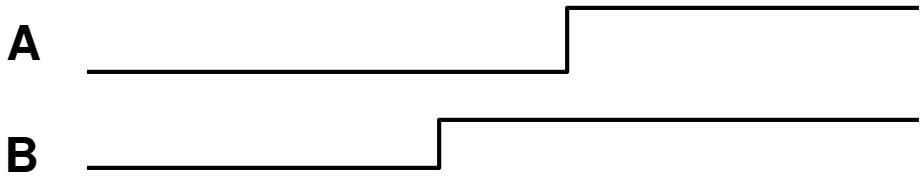
- Mixed parallel/sequential

```
initial fork
  #50 clk = 0;
  #80 begin
      clk = 1;
      rst = 0;
      end
 #130 clk = 0;
 #180 rst = 1;
join
```



Parallel Block and Sequential Block

- Can have a different start/stop behavior




```
initial begin
  @(A) $display("A");
  @(B) $display("B");
end
```

```
initial fork
  @(A) $display("A");
  @(B) $display("B");
join
```

Will these two behave differently for the given stimuli ?

Named Block

- Give a parallel or sequential block a name

```
initial fork : stimuli
  #50 clk = 0;
  #80 begin
      clk = 1;
      rst = 0;
      end
 #130 clk = 0;
 #180 rst = 1;
join
```

- Named blocks can be *disabled* (terminated)

```
initial begin : runthis
    forever
      begin
      #10 a = a + 1;
      if (a > 30)
        disable runthis;
      end
end
```

Today

- Conditional statements
  - if-then-else, nested-if, multiway decisions
  - case
- Loop statement
- Parallel block statements, named block statements
- Plus a few useful *structural* techniques
  - Generate construct
- Palnitkar: 7.4 - 7.10
- IEEE 1364: 9.4 - 9.8, 12.4

Generate

- Whenever you instantiate modules, you interconnect and wire them *statically*.
- The behavior of the modules is also *static*.
- Parameters are useful but simple, e.g. simple expansion of bus interconnection, creation of gate *arrays*, and so on.

- The Generate statement provides more flexibility for design creation. It enables to expand your design
  - using for loops (generate)
  - using if-then-else (generate-conditional)
  - using case (generate-case).

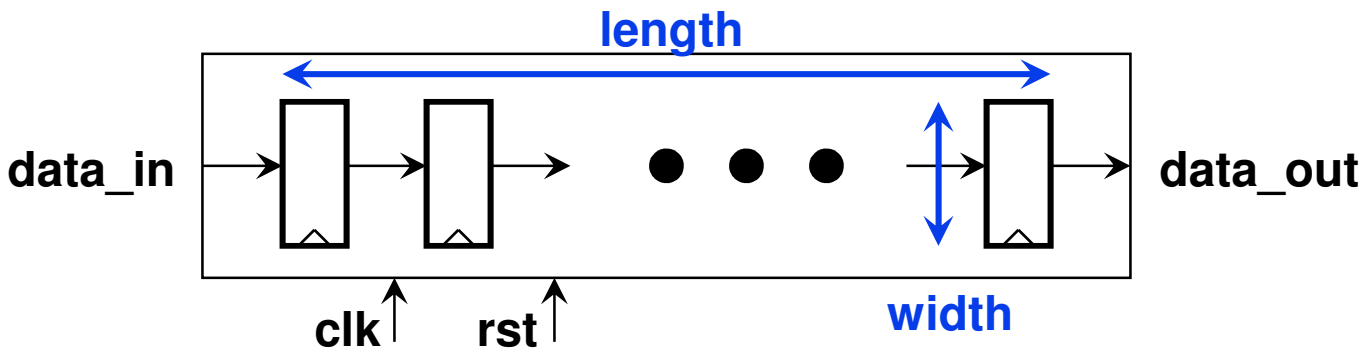
Example

- Assume you have 2 types of adders, A and B.
  - For wordlengths smaller than 12 bit, adder type A is quicker.
  - For wordlengths above 12 bit, adder type B is quicker.
  - Can we create a single adder module that will choose the right module for us?

```
module #(parameter width = 10) myadder;
begin
  output [0:width-1] q;
  input  [0:width-1] a, b;

   if (width > 12)
      instantiate code for adder B
    else
      instantiate code for adder A

end
```

This is *meta* code. It does not make part of simulation model. But it does help to create the simulation model

Example: Generate Pipeline




```
module #(parameter width = 8,
                   length = 8) pipeline(data_out,
                                        data_in, clk, rst);
  output [0:width-1] data_out;
  input  [0:width-1] data_in;
  input  clk, rst;
  reg  [0:width-1] pipestage [0:length-1];
  wire [0:width-1] d_in      [0:length-1];

// generate_pipe goes here ..

endmodule
```

Example: Generate Pipeline

```
module #(parameter width = 8,
                   length = 8) pipeline(data_out,
                                        data_in, clk, rst);
  output [0:width-1] data_out;
  input  [0:width-1] data_in;
  input  clk, rst;
  reg    [0:width-1] pipestage [0:length-1];
  wire   [0:width-1] d_in      [0:length-1];
  assign d_in[0] = data_in;
  assign data_out = pipestage[length-1];
  generate
    genvar k;
    for (k=0; k<=length-1; k = k + 1) begin : W
              if (k > 0) assign d_in[k] = pipestage[k-1];
              always @(posedge clk or negedge rst)
                 if (~rst) pipestage[k] = 0; else
                           pipestage[k] = d_in[k];
            end
  endgenerate
endmodule
```

Example: Generate Pipeline

```
module #(parameter width = 8,
                   length = 8) pipeline(data_out,
                                        data_in, clk, rst);
  output [0:width-1] data_out;
  input  [0:width-1] data_in;
  input  clk, rst;
  reg    [0:width-1] pipestage [0:length-1];
  wire   [0:width-1] d_in      [0:length-1];
  assign d_in[0] = data_in;
  assign data_out = pipestage[length-1];
  generate
    genvar k;
    for (k=0; k<=length-1; k = k + 1) begin : W
      if (k > 0) assign d_in[k] = pipestage[k-1];
      always @(posedge clk or negedge rst)
        if (~rst) pipestage[k] = 0; else
                  pipestage[k] = d_in[k];
    end
  endgenerate
endmodule
```

metacode

generate loop

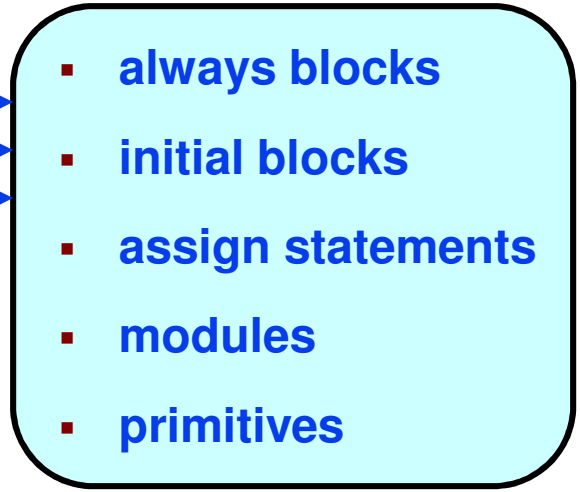
generate conditional

expanded code

Generate code and real code

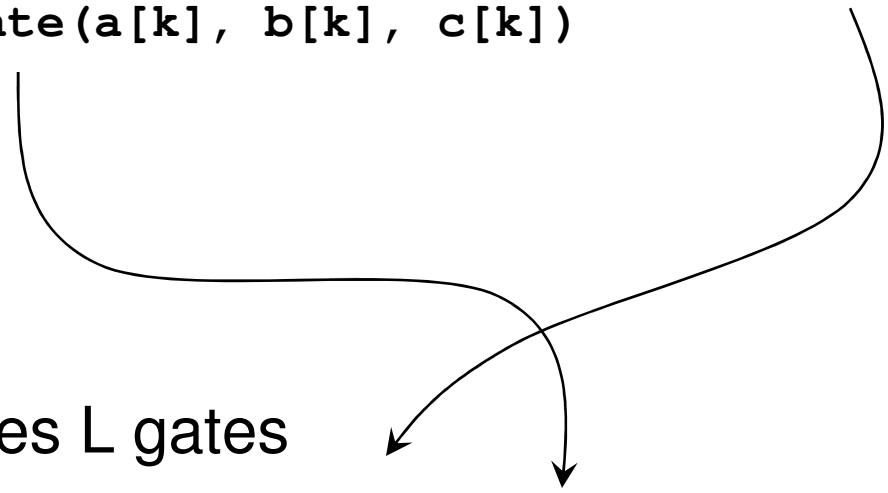
Meta Code:

```
generate for (init; condition; incr)
   begin : name
     // real code
   end
endgenerate

generate if (condition)
    // real code
endgenerate

generate case (N)
    1: // real code
   endcase
endgenerate
```



Generate loop creates hierarchical names

```
generate
  genvar k;
  for (k=0; k<=L-1; k = k + 1) begin : thegate
    xor mygate(a[k], b[k], c[k])
  end
endgenerate
```



- this generates L gates
  - First one is called thegate[0].mygate
  - Second one is called thegate[1].mygate
  - and so on

Summary

- Behavioral modeling can express fairly sophisticated constructs
  - if-then-else, nested-if, multiway-if
  - case, casex, casez, constant-case
  - forever, repeat, while, for
  - implicit event lists
  - parallel and sequential blocks, named blocks
- Many (but not all) of these constructs are synthesizable and create hardware
  - Worthwhile to learn how they work
- Verilog-2001 enables *generate* - parametric generation of code when a module is instantiated