If(a == 1)
Z = A + B;
Else
Z = A + C;

if(a == 1)
aux = B;
else
aux = C;
Z = A + aux;

assign B = 3;
assign Z = A * B;

assign Z = A + 2 * A;

assign aux = {1'b0, A[n-1:1]} + A[n-1:0];
assign Z = {aux, A[0]};

Simulation (1)
- Event Driven Simulation
- Order of execution in time tick is not fixed
- Simulator dependent (ouch!)
- Deadlocks can happen in perfectly good design
- Simulation and Synthesis can differ functionally
Execution Order

- Time Slice:
  - Q1 — (in any order):
    - Evaluate RHS of all non-blocking assignments
    - Evaluate RHS and change LHS of all blocking assignments
    - Evaluate RHS and change LHS of all continuous assignments
    - Evaluate inputs and change outputs of all primitives
    - Evaluate and print output from $display and $write
  - Q2 — (in any order):
    - Change LHS of all non-blocking assignments
  - Q3 — (in any order):
    - Call PLI with reason_synchronize
  - Q4:
    - Call PLI with reason_rosynchronize

Blocking vs Non-Blocking (1)
- Result: $a = b = c$

```
always @(b) begin
  a = b;
  c = a;
end
```

Result $a = b$
and $c = old a$

Blocking vs Non-Blocking (2)
- Use Non-Blocking for FlipFlop Inference:
  - posedge/negedge require nonblocking
  - Use #1 to visual causality!

```
always @(posedge clock) begin
  b <= #1 a; /* b and c will be flip flips */
  c <= #1 b;
end
```

```
always @(posedge clock) begin
  b = a; /* Only c will be a flip flop */
  c <= #1 b;
end
```

```
always @(posedge clock) begin
  b = a; /* Only c will be a flip flop */
  c <= #1 b;
end
```

Blocking vs Non-Blocking (3)
- If you use Blocking for FlipFlop Inference: You will not get what you want

```
always @(posedge clock) begin
  b = a; /* Only c will be a flip flop */
  c = b; /* b will go away after synthesis */
end
```

```
always @(posedge clock) begin
  c = b; /* c and b will be flip flops */
  b = a;
end
```

Blocking vs Non-Blocking (4)
Race Conditions

```
file xyz.v:
module xyz (a, b, clock);
  input b, clock;
  output a;
  reg a;
  always @(posedge clock)
    a = b;
endmodule
```

```
file abc.v:
module abc (b, c, clock);
  input c, clock;
  output b;
  reg b;
  always @(posedge clock)
    b = c;
endmodule
```

Combination Lock (1)
- Used to allow entry to a locked room:

2-bit serial combination. Example 01, 11:
1. Set switches to 01, press ENTER
2. Set switches to 11, press ENTER
3. OPEN is asserted (OPEN=1).
If wrong code, ERROR is asserted (after second combo word entry).
Press Reset at anytime to try again.
Combination Lock (2)

<table>
<thead>
<tr>
<th>Input Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Clear any entered numbers</td>
</tr>
<tr>
<td>Enter</td>
<td>Read the switches (enter a number in the combination)</td>
</tr>
<tr>
<td>Comb (1:0)</td>
<td>Two binary switches</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Lock opens</td>
</tr>
<tr>
<td>Error</td>
<td>Incorrect combination</td>
</tr>
</tbody>
</table>

State Machine Transition Diagram

Partitioning

FSMs in Verilog (1)
- Two always blocks
  - One is CurrentState register
  - Other is combinational
    - Generates NextState
    - Generates Outputs
- USE MOORE MACHINES
  - Avoid combinational loops

FSMs in Verilog (2)
```verilog
module MyFSM(In, Out, Clock, Reset);
  input In, Clock, Reset;
  output Out;
  parameter IDLE = 1'b0,
              RUNNING = 1'b1;
  reg CurrentState, NextState, Out;
  always @(posedge Clock)
  if (Reset) CurrentState <= IDLE;
  else CurrentState <= NextState;
  ```
FSMs in Verilog (3)

```verilog
always @ (CurrentState or In) begin
    NextState = CurrentState;
    Out = 1'b0;
    // A case block goes here
end
endmodule
```

FSMs in Verilog (4)

```verilog
case (CurrentState)
    IDLE: begin
        if (In) NextState = RUNNING;
        Out = 1'b0;
    end
    RUNNING: begin
        if (In) NextState = IDLE;
        Out = 1'b1;
    end
    default: begin
        NextState = 1'bX;
        Out = 1'bX;
    end
endcase
```