Initial isn’t Physical

\[
\text{initial } C = A \& B;
\]

Then what? What happens when A or B changes?

\[
\text{initial OUT } \leq 1\text{’b}0;
\]

That’s not how a flipflop works!
“always” is for circuits

always @(A or B)
C = A & B;

This is the way a gate works.

always @(posedge CLK or posedge CLR) begin
....
Initial Block

- An *initial* block starts at time 0 and runs exactly once.
- Initial blocks can be used to give a variable an initial value.
- They can also be used for setting sequences of values.
- If there is more than one active statement in an initial block, keywords “begin” and “end” are needed.
Begin…End

• Begin…end pairs are analogous to curly braces \{\} in C.

• A single statement does not need to be enclosed in them, two or more do.

```
initial begin
  a = 1'b1;
  c = 1'b0;
end
```

```
initial rst = 1'b0;
initial clk = 1'b0;
```
Begin...End Pitfall

//No begin...end needed
always @(posedge CLK)
    if (!RST) MYREG <= 0;

/*Decide to add a check to see if statement was executed*/
always @(posedge CLK) begin
    if (!RST) MYREG <= 0;
    $display ("Reset Done \
");
end

Syntactically, still OK. Logically?
Logically NOT OK

- Second statement (display) will execute whether or not previous RST clause runs.
- Display is not part of “if” clause.
- No syntax error, but a logical error.
Use begin…end Around ‘if’ Clause

always @(posedge CLK) begin
  if (!RST) begin
    MYREG <= 0;
    $display ("Reset Done");
  end
end

Now the display is only done if RST is 0.
Concurrency

- Verilog is used for hardware modeling.
- In a circuit, many signals are changing simultaneously.
- A computer can typically only do one thing at a time.
- Circuits are simulated by scheduling multiple events to occur at the same simulation time, but assignments are really made sequentially.
Concurrency

always @(posedge CLK)
CLK2 <= ~CLK2;

always @(posedge CLK)
OUT <= A & B;

Both events are concurrent. There is no way to predict which will be scheduled first. Order of statements in code is not significant. Note these devices only make sense with “always” blocks. “Initial” would be meaningless.
Race Condition

`timescale 1 ns / 1 ns
module race (out1, out2, clk, rst);
output out1, out2;
input clk, rst;
reg out1, out2;
always @(posedge clk or negedge rst) 
if (!rst) out1 = 0;else out1 = out2;
always @(posedge clk or negedge rst)
if (!rst) out2 = 1;
else out2 = out1;
endmodule
Another Race

Value of B depends on order of evaluation. This order is non-deterministic.

always @(posedge CLK) A = A + 1;
always @(posedge CLK) B = A;
Order of Execution

always @(posedge CLK or negedge RST) begin
    if (!RST) CLK2 = 1'b0;
    else CLK2 = ~CLK2;
end
always @(posedge CLK) begin
    if (CNT == TC && CLK2) MATCH = 1'b1;
    else MATCH = 1'b0;
end
always @(posedge CLK2 or negedge RST) begin
    if (!RST)
        if (!RST)
            CNT = 4'b0;
        else
            CNT = CNT + 1;
    end
Output relations depend or order of execution. MATCH may go high when CNT is 15.
And it could equally well go high only after CNT has wrapped back to 0.
Order of Execution

• Easy to control in preceding example: just move the ‘always’ blocks around.

• Not easy when the blocks are in different files and subject to different enabling events.

• Proper use of blocking/non-blocking assignment operators mitigates the problem.
Scheduling

always @(posedge CLK)
CLK2 <= ~CLK2;

always @(posedge CLK)
CNT <= CNT + 1;

always @(posedge CLK2)
begin
if (CNT == 4’d15)
  TC <= 1’b1;
else TC <= 1’b0;
end
When Does TC Go High?
When CNT = 15?
When CNT = 0?
module order_of_execution();

reg CLK, CLK2, RST, TC;
reg [3:0] CNT;

initial begin
    CLK = 1'b0;
    forever #1 CLK = ~CLK;
end

initial begin
    RST = 1'b1;
    #2 RST = 1'b0;
    #2 RST = 1'b1;
end

always @(posedge CLK or negedge RST) begin
    if (!RST) begin
        CNT <= 4'b0;
        CLK2 <= 1'b0;
    end
    else begin
        CNT <= CNT + 1;
        CLK2 <= ~CLK2;
    end
end

always @(posedge CLK2) begin
    if (CNT == 4'd15) TC <= 1'b1;
    else TC <= 1'b0;
end
endmodule

TC goes high at CNT = 15

CNT and CLK2 are a Δ time behind CLK. They do not go into the same queue as CLK.

Because CLK2 comes after CNT has been updated, TC goes high when CNT still is 15. This behavior is not guaranteed. It may be scheduled the other way.
```verilog
`timescale 1 ns / 1 ns
module order_of_execution();

reg CLK, CLK2, RST, TC;
reg [3:0] CNT;

initial begin
  CLK = 1'b0;
  forever #1 CLK = ~CLK;
end

initial begin
  RST = 1'b1;
  #2 RST = 1'b0;
  #2 RST = 1'b1;
end

always @(posedge CLK or negedge RST) begin
  if (!RST) CNT <= 4'b0;
  else CNT <= CNT + 1;
end

always @(posedge CLK) CLK2 = CLK;
always @(posedge CLK2) begin
  if (CNT == 4'd15) TC <= 1'b1;
  else TC <= 1'b0;
end
endmodule
```

**TC goes high at CNT = 0**

CNT is a delta time after CLK

CLK2 is the same as CLK

Synchronous design will cause TC to go high on the edge following the edge that cause CNT to go to 15, which is when CNT goes back to 0.
Why Does It Do That?

- CLK and CLK2 happen at the same delta time.
- CNT is incremented a delta cycle after CLK2 changes state.
- So when CNT goes to 15, the evaluation that would cause TC to go to logic 1 has already passed for that cycle.
'timescale 1 ns / 1 ps
module order_of_execution();

reg CLK, CLK2, RST, TC;
reg [3:0] CNT;

initial begin
  CLK = 1'b0;
  forever #1 CLK = ~CLK;
end

initial begin
  RST = 1'b1;
  #2 RST = 1'b0;
  #2 RST = 1'b1;
end
always @(posedge CLK or negedge RST) begin
  if (!RST) CNT <= 4'b0;
  else CNT <= CNT + 1;
end
always @(CLK) #0.1 CLK2 = CLK; //Delay added here
always @ (posedge CLK2) begin
  if (CNT == 4'd15) TC <= 1'b1;
  else TC <= 1'b0;
end
endmodule

Add delay to derived clock and TC goes high at CNT = 15.
Delta Time

• Both CLK2 and CNT will be updated after the rising edge of CLK.
• There is no way to predict which will be scheduled first: both are a delta time after the rising edge of CLK.
• In rigorous synchronous design, it will not matter. Using derived clocks, it might.
• Moral: don’t use logic signals as clocks.
Delta Time and Real Devices

• Delta time is a software construct used in simulators for scheduling.
• Delay is real, silicon delay plus wiring/capacitance delay.
• Use of derived clocks can and does lead to real circuit failures: metastability errors, order of execution problems.
• It’s not just a software problem.
Is This Good Design?

always @(posedge CLK)
CLK2 <= ~CLK2;
Initialization in Simulation

• In a real circuit, initialization might not be necessary.
  – After power on, the flop-flop will stabilize to something, 1 or 0.

• In simulation, it will start out as X and stay X. X is a stable value, like 0 or 1.

• The inverse of X is X, so an uninitialized variable with feedback may stay X forever.
Initialization Is Needed

always @(posedge CLK or negedge RST)
begin
if (!RST) CLK2 <= 1'b0;
else CLK2 <= ~CLK2;
end
Initialization

Initialization must be written in such a manner as to model hardware. A software construct that seems to imply the right function but does not correlate to something physical will not do.

/*NO NO NO*/
initial begin
  OUT <= 1'b0;
forever OUT <= IN;
end

/*The right way*/
always @(posedge CLK or negedge RST) begin
  if (!RST) OUT <= 1'b0;
  else OUT <= IN;
always and initial Blocks

- Similar in syntax, but initial blocks only run one time. always blocks run continuously and are updated whenever something in their sensitivity list changes.
- All blocks run in parallel.
- Only always blocks may be used in circuit descriptions. initial blocks are only for use in stimulus modules.
‘always’ Blocks

• Simple assign statements are, well, simple:
  – assign sum = in1 + in2;
• Operational code is almost always done in ‘always’ blocks.
  – always @(in1 or in2) sum <= in1 + in2;
• Not much advantage so far.
always @(state or start or init_count) begin
  case (state)
    idle: begin
      if (start) begin
        next_state <= init_count;
        load_value <= 16'b0;
      ....
      else begin
        next_state <= idle;
        load_value <= 16’bx;
Combinational or Sequential

- Blocks of operational code can be combinational or sequential.
- A sequential block will produce a gate-level design containing registers.
- A combinational block will infer only gates.
- But are not registers made of gates?
Register Inference

• Synthesizable blocks containing edge statements will ‘infer’ flipflops.
• There is no need to build flipflops from primitive gates.
• The synthesizer will pick the proper type of register from its library of components, which is far bigger than the set of Verilog primitives.
Edge Statements

- Verilog has two types of edge statements: one for rising edges, one for falling.
- `posedge` means rising (positive) edge.
- `negedge` means falling (negative) edge.
- Use of either or both in a sensitivity list will infer a sequential device (flipflop).
- Sensitivity lists must be either sequential or combinational, never both.
### Combinational/Sequential

A combinational sensitivity list may not contain any edge constructs:

- always \(@\(A \text{ or } B\)\)

Resulting circuitry will not have any registers.

A sequential sensitivity list only contains edge specifiers:

- always \(@\)[](posedge \text{CLK})

All assignments in the block will be to flipflops.
Sensitivity Lists

- Always blocks have a “sensitivity list.”
- The simulator will only schedule events that are triggered by changes in signals in the sensitivity list.
- Synthesizers ignore combinational sensitivity lists.
- This leads careless designers to have mismatches between simulated behavioral code and synthesized designs.
Combinational Sensitivity List

always @(a or b or cin)

\{cout, sum\} = a + b + cin;

This is functionally equivalent to:

assign \{cout, sum\} = a + b + cin;
Incomplete Sensitivity List

always @(a or cin)
        {cout, sum} = a + b + cin;

This will ignore any changes on b:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>cout</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Sensitivity List and Synthesis

always @(A or CIN)

\{cout, sum\} = a + b + cin;

always @(A or B or CIN)

\{cout, sum\} = a + b + cin;

will produce exactly the same hardware:
Sequential Sensitivity List

- Sequential sensitivity lists always infer registers.
- They do this by employing ‘edge’ constructs.
- Edges can be positive or negative, referring to rising or falling signals.
- Good design technique will employ only clock or reset edges, never logic signals.
Sensitivity Lists

• A sensitivity list must be either combinational or sequential.
• It can never be a mixture of the two.
• This means that if a block is sensitive to an edge (e.g. posedge CLK) any other signals in that list must also be edges.
Mixed Sensitivity Lists

• The simulator will accept defective, mixed combinational/sequential sensitivity lists.
  always @(posedge CLK or COUNT)
• It would not be meaningful for hardware description.
• Never write a circuit description using such a defective sensitivity list.
• HDL design IS NOT PROGRAMMING!!!
D flipflop Inference

```verilog
always @(posedge CLOCK)
  Q <= D;
```

- On the rising edge of CLOCK, transfer the value of D to Q.
- At all other times, do nothing.
- This is behavioral code. It does not take into account device physics, i.e. setup and hold times.
Flipflop With Reset

always @(posedge CLOCK or negedge RESET) begin
    if (!RESET)
        Q <= 1'b0;
    else
        Q <= D;
end

A sequential element (that is, a flipflop) can be sensitive to more than one type of event.
Sequential Sensitivity Lists

• Should the sensitivity list for the D flipflop include D?
• How would you modify that flipflop model to add an asynchronous “SET” in addition to the “RESET?”
Three Edge Flipflop

always @(posedge CLK or negedge RST or negedge SET) begin
  if (!SET) Q <= 1'b1;
  else if (!RST) Q <= 1'b0;
  else Q <= D;
end

This is behavioral code to model an edge-triggered flipflop with preset and reset. Actual hardware models are done differently (using UDP’s, discussed last week of class).

This model gives priority to SET and does not say what to do if both set and reset are active simultaneously.
Concurrent Statements

• As previously noted, there is not always a guaranteed order of evaluation:
  assign a = b;
  assign c = d;
  $display(a, c);

• Order of evaluation can produce funny results.
Not all Statements Are Concurrent

So why does

always @(posedge CLK or negedge RESET) begin
  if (!RESET) …
  else…

always create a register that has an asynchronous clear that overrides the clock?
“If” Implies Priority

• Blocks are always parallel. Order of blocks in code does not matter. They all run at the same time.
• Within a block, order (priority) may be enforced.
• “If” clauses do imply order of evaluation.
Regs vs. Wires

• The quick and dirty rule of thumb:
  – If a signal is assigned a value inside an ‘always’ block or an ‘initial’ block, it is a reg.
  – If it is assigned a value anywhere else (where else might that be?) it is a wire.

• A rigorous examination of variable types will be forthcoming at a more convenient time.
# Equality Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>logical equality</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>logical inequality</td>
</tr>
<tr>
<td><code>===</code></td>
<td>case equality</td>
</tr>
<tr>
<td><code>!==</code></td>
<td>case inequality</td>
</tr>
</tbody>
</table>

---

50
Equality Operators

- The logical equality operators \((==, !=)\) generates \(x\) if any of the operands is \(x\) or \(z\).
- The case equality operators \((===, !==)\) compare all bits of both operands including \(x\) and \(z\).
Operators Incorporating “==”

• == is an assignment operator. It assign the value of the RHS of the equation to the LHS.
Example: assign X = Y;
• === is the equality operator.
• ==== is the identity operator, aka “case equality operator.”

<table>
<thead>
<tr>
<th>==</th>
<th>0 1 x z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 x x</td>
</tr>
<tr>
<td>1</td>
<td>0 1 x x</td>
</tr>
<tr>
<td>x</td>
<td>x x x x</td>
</tr>
<tr>
<td>z</td>
<td>x x x x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>===</th>
<th>0 1 x z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>x</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>z</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>
Equality Operators

Examples:

assume x = 4; y = 8; w = 3'b11z;
v = 3'b01x; j = 8; k = 3'b01x;
m = 3'b11z;

(x != y) evaluates to Logical 1
(y == x) evaluates to Logical 0
(v == k) evaluates to x
(v == = k) evaluates to 1
(w != = k) evaluates to 1
Resolving Multi-bit Operands

• Equality operators **will** resolve multi-bit operands containing x or z values when possible.

• Example: \( X = 5'b1010x \), \( Y = 5'b01001 \)
  
  \[
  (X == Y) \Rightarrow 0 \\
  (X === Y) \Rightarrow 0
  \]

• Text is misleading or wrong on this operator.
Equality Operators and Synthesis

- Case equality/inequality operators are UNSUPPORTED for synthesis.
- What would it mean to create a circuit that compared a value to an unknown?
- Circuit description Verilog must only use equality (==) and inequality (!=) operators, never case (=== and !==) operators. Case operators are only for use in test fixtures.
Assignment Operators

- Verilog includes two assignment operators:
  - blocking assignment: =
  - non-blocking assignment: <=

- While the difference may seem subtle, it is critical for generating the desired simulation response and the desired post-synthesis circuit.
Blocking Assignments

Blocking operators force sequential evaluations of statements:

```verilog
always @(posedge CLOCK) begin
    delay1 = input;
    delay2 = delay1;
end
```

Second statement is not evaluated until first assignment is completed.
Blocking Assignment

DELAY2 is not evaluated until after DELAY1 has been assigned.
Non-blocking Assignments

Non-blocking assignments are scheduled at the same time:

```vhdl
always @(posedge CLOCK) begin
    DELAY1 <= INPUT;
    DELAY2 <= DELAY1;
end
```

DELAY1 and DELAY2 are evaluated before the clock event and assigned at the clock edge.
Non-Blocking Assignment
Blocking and Non-blocking

• Only one type of assignment operator may be used within any one ‘always’ block.
• Both may be used within a module.
• The non-blocking operator was invented to overcome difficulties imposed by the blocking operator. It is extraordinarily rare to use the blocking operator in a sequential block.
Blocking Assignment Operator

• Previous example showed how use of blocking assignment operator can produce defective logic.
• Solution is NOT to throw out the blocking assignment operator.
• Inappropriate use of non-blocking operator can also produce defective logic.
Defective Coding Style

module ao4 (y, a, b, c, d);
  output y;
  input a, b, c, d;
  reg y, tmp1, tmp2;
  always @(a or b or c or d) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y <= tmp1 | tmp2;
  end
endmodule
What’s the Problem?

• It’s a combinational block.
• It will, when triggered, evaluate, evaluate, evaluate, then assign, assign, assign.
• Result is that old, stale values of tmp1 and tmp2 will be used to form output y.
• Problem occurs when a variable is on both left and right sides of assignments.
Poor Solution

• Put internal variables (tmp1 and tmp2) into the sensitivity list.
• Will cause multiple evaluations of the block, poor simulation performance.
• Will solve the problem, though.
The Right Way

• Use blocking assignment operators.
• Evaluate-assign, evaluate-assign, etc.
• Rule of thumb:
  – For sequential blocks, always use non-blocking operators.
  – For combinational blocks, always use blocking operators.
Mixing Assignment Operators

• NC Verilog simulator will allow mixing of assignment operators.
• It should not.
• Scheduling of assignments is unclear.
• Synthesis tools will not allow it: compile error.
• Never mix assignment operators in any always or initial block.