Verilog HDL

Mark Redekopp

Purpose

- HDL's were originally used to model and simulate hardware before building it
- In the past 30 years, synthesis tools were developed that can essentially build the hardware from the same description
- A progression continues allowing more abstract descriptions to be used for both simulating a hardware/software design and synthesizing actual hardware

Differences from Software

- Software programming languages specify a sequence of operations to be executed with implied ordering
  - Operations executed in sequential order (line 1, line 2, line 3)
- Hardware Description Languages (HDLs) like Verilog and VHDL do NOT describe a ***temporal sequence of operations like software*** but a **spatial/physical hardware design**
  - **HDL processes (code blocks) execute in parallel (not sequentially)**

Software

Perform x+y first and store the result in v. Only then assign d-v in the tmp.

```
1  v = x+y;
2  tmp = d-v;
```

Hardware

This description models a mux and an adder running in parallel. One operation does not run *BEFORE* another

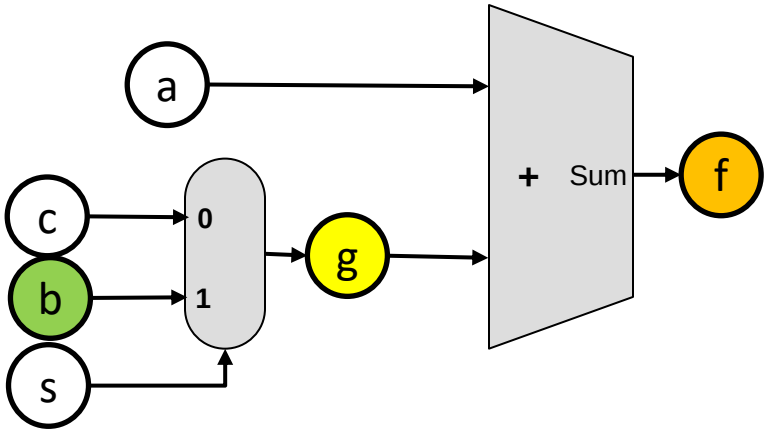
```
1  assign f = a + b;
1  assign g = (s==1) ? a : b;
```

Event Driven Paradigm: If a, b, or s changes, f and/or g will be re-evaluated

HW Description Differences

- "Execution" of our described hardware model is "event-based"
  - This means that code executes (really hardware updating) when certain events occur (like an input changing)
- Given the description below
  - Execution does NOT occur in sequence/order
  - But if 'b' changes (which is an input to 'g') then the assign statement for 'g' executes which then triggers the assign statement for 'f' to execute.

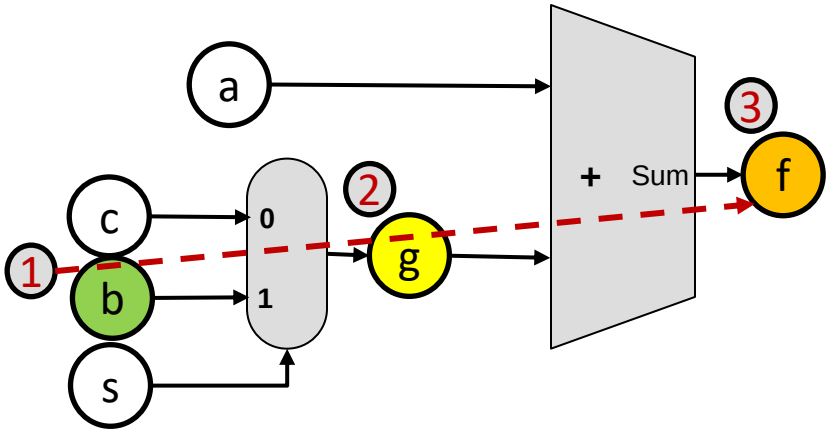
```
assign f = a + g;
assign g = (s==1) ? b : c;
```



HW Description Differences

- "Execution" of our described hardware model is "event-based", in that code executes when certain events occur (like an input changing)
- Given the description below
  - Execution does NOT occur in sequence/order
  - But if 'b' changes (which is an input to 'g') then the assign statement for 'g' executes which then triggers the assign statement for 'f' to execute.

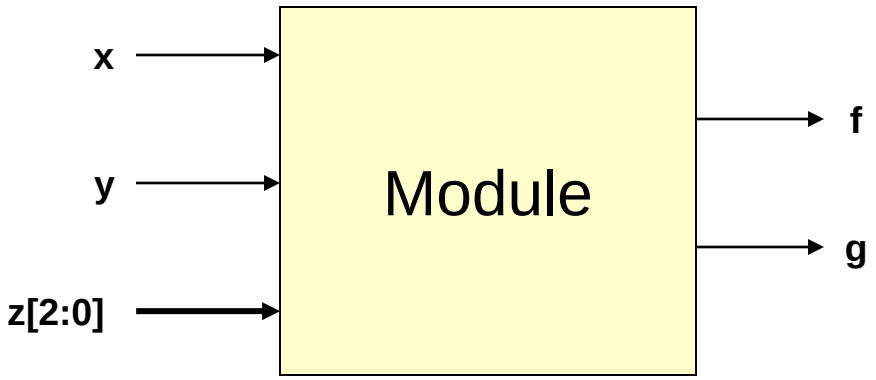
```
assign f = a + g;
assign g = (s==1) ? b : c;
```



VERILOG BASICS

Modules

- Each Verilog designs starts as a block diagram (called a “module” in Verilog)
- Start with input and output signals, then describe how to produce outputs from inputs




```
module m1(x,y,z,f,g);

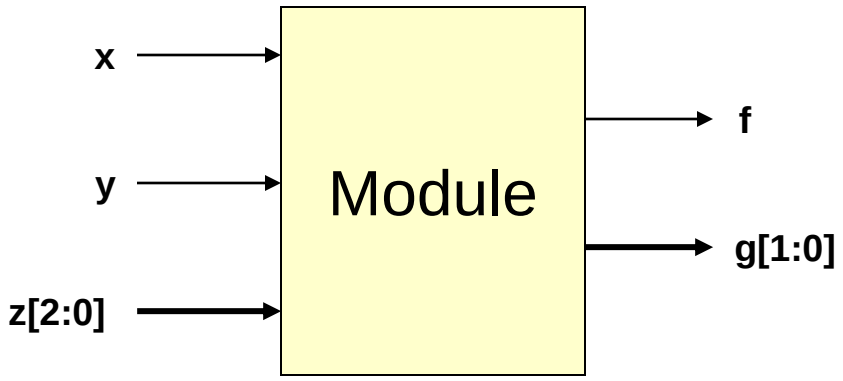
// circuit
// description

endmodule
```

Software analogy: Modules are like functions, but also like classes in that they are objects that you can instantiate multiple times.

Ports

- Input and output signals of a module are called **ports** (similar to parameters/arguments of a software function)
- Unlike software, ports need to be declared as **input** or **output**
- Vectors declared using `[MSB:LSB]` notation



These are the ports

```
module m1(x,y,z,f,g);
input  x,y;
input  [2:0] z;
output f;
output [1:0] g;
endmodule
```

Signal Types

- Signals represent the inputs, outputs, and internal values
- Signals need to be typed
  - Similar to variables in software (e.g. int, char)
- 2 basic types
  - **Wire**: Represents a node whose value can be derived from its input drivers' values
    - Only for modeling combinational logic or structural descriptions (wiring outputs of a module instance to another)
    - Should be used for any signals produced by **assign** statements
  - **Reg**(ister): Used for signals that are described behaviorally
    - **Important**: ***reg type signals do NOT imply they are produced by a hardware register*** but instead refers to the simulator's need to use storage to track this signals value
    - Should be used for anything produced by an **always** or **initial** block (be it combinational or sequential)

```
module m1(x,y,z,f,g);

 input  x,y;
 input  [2:0] z
 output f;
 output reg [1:0] g;


 wire   n1, n2;
 reg    n3, n4;
 ...
endmodule
```

Inputs are always type 'wire'. Outputs are assumed 'wire' but can be redefined as 'reg'

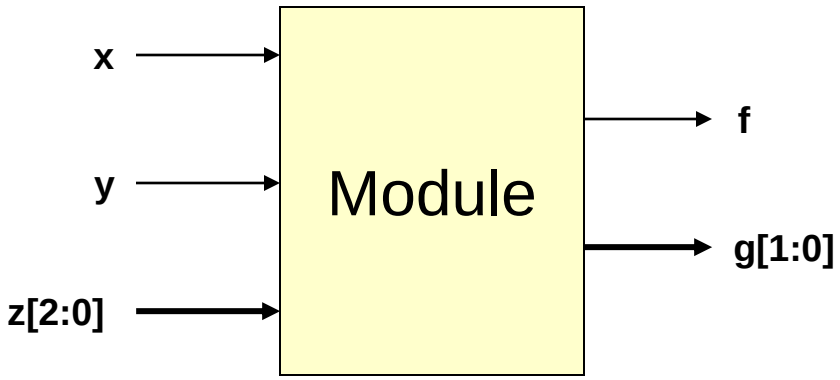
Ports Revisited

- In Verilog 2001, the direction and type of ports may be provided together in the module declaration

```
module m1(x,y,z,f,g);
input  x,y;
input  [2:0] z;
output f;
output [1:0] g;

reg [1:0] g;
...
```

Pre-Verilog2001




```
module m1(
  input        x,
  input        y,
  input  [2:0] z;
  output reg   f;
  output [1:0] g);
...
```

Verilog2001

Constants

- Multiple bit constants can be written in the form:
  - [size] `base value
    - *size* is number of bits in constant
    - *base* is o or O for octal, b or B for binary, d or D for decimal, h or H for hexadecimal
    - *value* is sequence of digits valid for specified *base*
      - Values a through f (for hexadecimal base) are case-insensitive
- Examples:
  - `4'b0000   // 4-bits binary`
  - `6'b101101 // 6-bits binary`
  - `8'hfC     // 8-bits in hex`

  - ***Decimal is default base***
  - `17      // 17 decimal converted to appropriate`
    `// number of unsigned bits (i.e. 32'b010001`

Structural vs. Behavioral Modeling

Structural

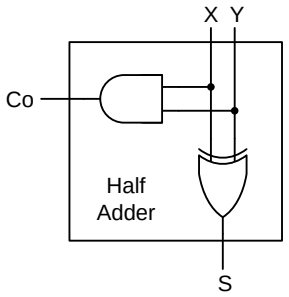
- Starting with primitive gates, build up a hierarchy of components and specify how they should be connected

Behavioral

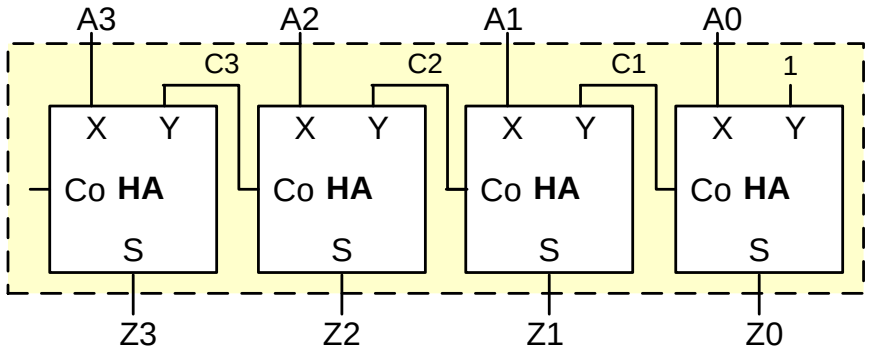
- Describe behavior and let synthesis tools select internal components and connections

Structural Modeling

- Starting with primitive gates, build up a hierarchy of components and specify how they should be connected



Structural specification of a half adder



Use half adders to structurally describe an incrementer

```
module ha(x,y,s,co);
  input    x,y;
  output   s,co;

  xor i1(s,x,y);
  and i2(co,x,y);
endmodule

module incrementer(a,z);
  input    [3:0] a;
  output   [3:0] z;
  wire     [3:1] c;

  ha ha0(a[0],1,z[0],c[1]);
  ha ha1(a[1],c[1],z[1],c[2]);
  ha ha2(a[2],c[2],z[2],c[3]);
  ha ha3(a[3],c[3],z[3], );

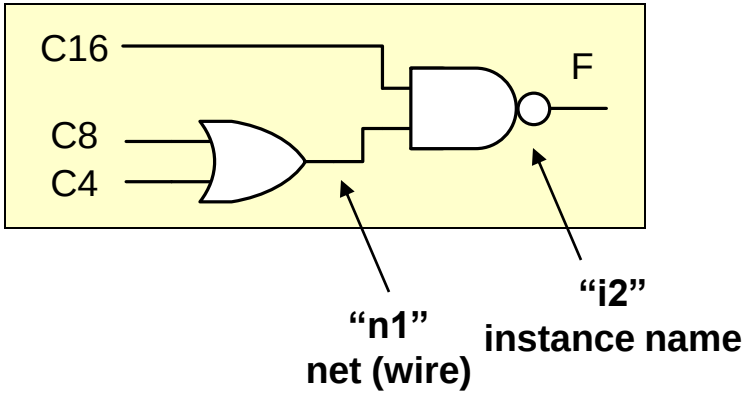
endmodule
```

Structural Modeling of Logic Gates

- Modules and primitive gates can be instantiated using the following format:

  `module_name instance_name(output, input1, input2,…)`

- Input and outputs must be wire types
- Built-in supported Gates: *and, or, not, nand, nor, xor, xnor*




```
module m1(c16,c8,c4,f);
  input    c16,c8,c4;
  output   f;
  wire     n1;

  or   i1(n1,c8,c4);
  nand i2(f,c16,n1);

endmodule
```

Verilog Description

Instantiating User-Defined Modules

- Format: `module_name instance_name(port1, port2, port3, …)`
- **Positional mapping**
  - Signals of instantiation ports are associated using the order of module's port declaration (i.e. order is everything)
- **Named mapping**
  - Signals of instantiation ports are explicitly associated with module's ports (i.e. order is unimportant)
  - `module_name instance_name(.module_port_name(signal_name),…);`

```
module ha(x,y,s,co);
  ...
endmodule

module incrementer(a,z);
  ha ha0(a[0],1,z[0],c[1]);
  ...
endmodule
```

Positional mapping

```
module ha(x,y,s,co);
  ...
endmodule

module incrementer(a,z);
  ha ha0(.x(a[0]),
         .s(z[0]),
         .y(1),
         .co(c[1]) );
  ...
endmodule
```

Named Mapping

Internal Signals

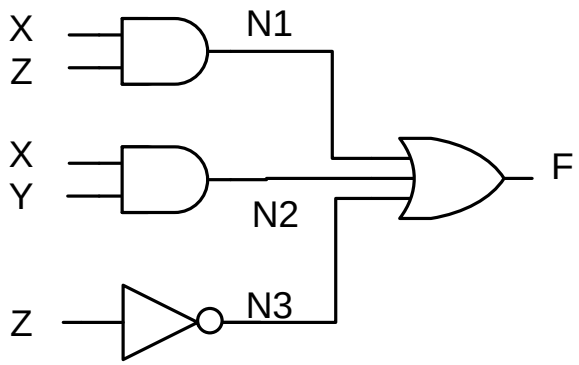
- Define signals (wire or reg) for each internal signal/wire

```
module m2(x,y,z,f);

input  x,y,z;
output f;
wire   n1,n2,n3;

and u1(n1,x,z);  // instance names need
and u2(n2,x,y);  // not be declared
not u3(n3,z);
or  u4(f,n1,n2,n3);

endmodule
```



Behavioral Modeling

- Describe behavior and let synthesis tools select internal components and connections
- Advantages:
  - Easier to specify
  - Synthesis tool can pick appropriate implementation (for speed / area / etc.)

```
module incrementer(a,z);
  input    [3:0] a;
  output   [3:0] z;

  assign z = a + 1'b1;

endmodule
```

Use higher level operations and let synthesis tools infer the necessary logic

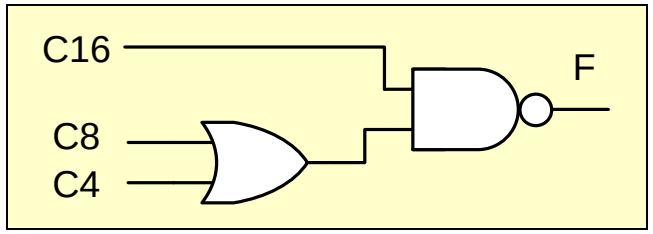
Could instantiate an appropriate implementation (e.g. ripple-carry adder, a fast carry-lookahead adder, etc.) or optimize the design as needed

Operators

- Operator types
  - Non-blocking / Blocking assignment (<=, =)
  - Arithmetic (+, -, *, /, %)
  - Relational (<, <=, >, >=)
  - 2-State Equality (0,1 only) (==, !=)
  - 4-State Equality (0,1,Z,X) (=== , !==)
  - Logical (&&, ||, !)
  - Bitwise (~, &, |, ^, ~^)
  - Reduction (&, ~&, |, ~|, ^, ~^)
  - Shift (<<, >>)
  - Conditional (? :)
  - Concatenation and replication { }

`assign` Statement

- Used for combinational logic expressions
- **Must output to a `wire` signal type!**
- Can be used anywhere in the body of a module's code
- All `assign` statements run in parallel
- Change of any signal on RHS (right-hand side) triggers re-evaluation of LHS (output)
- Format:
  - **assign *output* = *expr;***
    - '&' means AND
    - '|' means OR
    - '~' means NOT
    - '^' means XOR
    - Can use arithmetic operators too (and synthesis will infer adder, multiplier, etc.)




```
module m1(c16,c8,c4,f);
  input    c16,c8,c4;
  output   f;
  wire     n1;

  or   i1(n1,c8,c4);
  nand i2(f,c16,n1);

endmodule
```

```
module m1(c16,c8,c4,f);
  input    c16,c8,c4;
  output   f;

  assign f = ~(c16 & (c8 | c4));

endmodule
```

Multi-bit (Vector) Signals

- Reference individual bits or groups of bits by placing the desired index in brackets (e.g. x[3] or x[2:1])
- Form vector from individual signals by placing signals in brackets (i.e. { }) and separate with commas

```
module m1(x,f);

  input [2:0] x;
  output      f;

  // f = minterm 5
  assign f = x[2] & ~x[1] & x[0];

endmodule
```

```
module incrementer(a,x,y,z);

  input [2:0] a;
  output x,y,z;


  assign {x,y,z} = a + 1;

endmodule
```

More Assign Statement

- Can be used with other operators besides simple logic functions
  - Arithmetic
    (+, -, *, /,%=modulo/remainder)
  - Shifting (<<, >>)
  - Relational
    (<, <=, >, >=, !=, ==)
    - Produces a single bit output
      ('1' = true / '0' false)
  - Conditional operator (? :)
    - Syntax:
      `condition ? statement_if_true : statement_if_false;`

```
module m1(x,y,sub,s,cout,d,z,f,g);
  input    [3:0] x,y;
  input          sub;
  output   [3:0] s,d;
  output   [3:0] z;
  output         cout,f,g;

  assign {cout,s} = {0,x} + {0,y};
  assign d = x - y;
  assign f = (x == 4'h5);
  assign g = (y < 0);
  assign z = (sub==1) ? x-y : x+y;

endmodule
```

Sample "Assign" statements

Always Block (Combinational)

- Primary unit of parallelism in code
  - `always` and `assign` statements run in parallel
  - Statements w/in always blocks are executed sequentially
- Format
  - ```
    always @(sensitivity list)
    begin
      statements
    end
    ```
- Always blocks are "executed" when there is a change in a signal in the sensitivity list
- When modeling combinational logic, sensitivity lists should include ALL inputs (i.e. all signals in the RHS's)
  - Verilog2001 allows @* as a shorthand all RHS signals
- Generation of a signal must be done within a single always block (not spread across multiple always blocks)
  - Signals generated in an always block must be declared type **reg**

```
module addsub(a,b,sub,s);

  input [3:0] a,b;
  input sub;
  output reg [3:0] s;
  reg [3:0] bmux;

  always @(b,sub)
  begin
    if(sub == 1)
       bmux <= ~b;
    else
       bmux <= b;
  end

  always @*
  begin
    s = a + bmux + sub;
  end
endmodule
```
```

Always Block (Sequential)

- Flip-flops (sequential logic) are modeled using an always block sensitive to the edge (posedge or negedge) of the clock
  - block will only be executed on the positive edge of the clock
- Generally, use the non-blocking assignment operator (<=) in clocked "always" blocks

```
module accumulator(x,z,clk,rst);

  input [3:0] x;
  input clk,rst;
  output [3:0] z;
  reg [3:0] z;

  always @(posedge clk)
  begin
    if(rst == 1)
      z <= 4'b0000;
    else
      z <= z + x;
  end

endmodule
```

Procedural Statements

- Must appear inside an *always* or *initial* block
- Procedural statements include
  - if…else if…else…
  - case statement
  - for loop (usually unnecessary for describing logic)
  - while loop (usually unnecessary for describing logic)

If…Else If…Else Statements

- Syntax

```
if(expr)
   statement;
else if(expr)
   statement;
else
   statement;
```

- If multiple statements exist in the body of *if…else if…else* then enclose in ***begin…end*** construct

```
// 4-to-1 mux description
always @*
begin
  if(sel == 2'b00)
    y <= i0;
  else if(sel == 2'b01)
    y <= i1;
  else if(sel == 2'b10)
    y <= i2;
  else
    y <= i3;
end
...
```

```
...
if( rst == 1 )
 begin
  q1 <= 0;
  q2 <= 0;
 end
else
 begin
  q1 <= d;
  q2 <= q1;
 end
```

Case Statements

- Syntax

```
case(expr)
option1:
   begin
      statements;
   end
option2: statement;
[default: statement;]
endcase
```

```
// 4-to-1 mux description

always @(i0,i1,i2,i3,sel)
begin
  case(sel)
    2'b00: y <= i0;
    2'b01: y <= i1;
    2'b10: y <= i2;
    default: y <= i3;
  endcase
end
```

- Default statement is optional
- If multiple statements as the body of an option then enclose in *begin…end* construct

More About Always Blocks

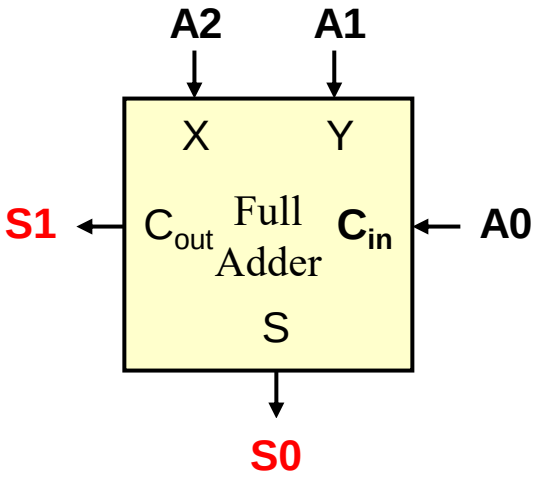
- When writing an always block your goal is to provide enough of a description to allow the tools to fill in a truth table for one more output signals
  - Cases can be overwritten with the last assignment taking precedence
  - Any use of loops or if statements is essentially "0-time" operation to describe a truth table

```
reg       f;
reg [2:0] a;
always @*
begin
  f <= 0;  // all rows = 0
  // now overwrite desired
  // locations with 1s
  if(a < 2) f <= 1;
  else if(a == 3'b101) f <= 1;
  else if(a % 3 == 1) f <= 1;
end
```

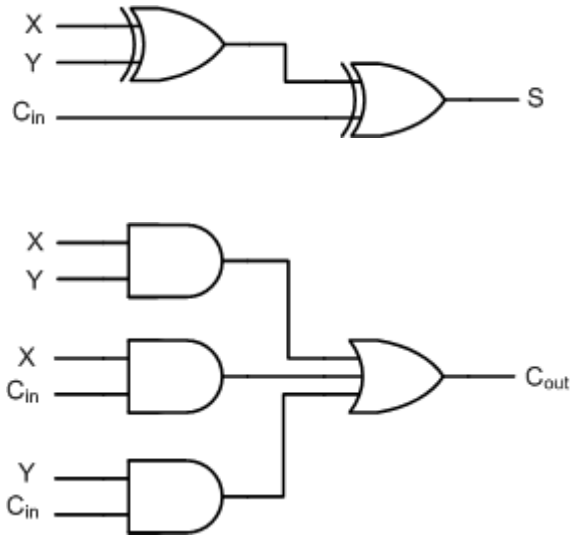
A2	A1	A0	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Understanding Combinational Always Blocks: Full Adder

- Consider a full adder (adds 3 bits to produce a 2-bit sum)
  - Block diagram, truth table, and logic implementation shown below
- 4 different description approaches are shown in the subsequent slides
  - Approach 1 is likely the easiest/preferred approach
  - Approach 2-4 are there to illustrate how various Verilog constructs operate and are interpreted



A2	A1	A0	S1	S0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Approach 1 and 2

- Approach 1 (Preferred): Just use Verilog's built in operators
  - Synthesis tool will determine logic
- Approach 2: Determine the logic yourself (via some minimization approach) and describe the logic equations.

```
integer i;
wire [2:0] a;
wire [1:0] sum;

assign sum = a[2] + a[1] + a[0];
```

Approach 1

```
integer i;
wire [2:0] a;
wire [1:0] sum;

assign sum[0] = a[2] ^ a[1] ^ a[0];
assign sum[1] = (a[2] & a[1]) |
                (a[2] & a[0]) |
                (a[1] & a[0]);
```

Approach 2

Approach 3

- Describe the truth table uses cases or if statement
  - Synthesis tool will determine logic

```
 integer i;
 reg [2:0] a;
 reg [1:0] sum, cnt;
always @*
begin
  if(a == 3'b000)
    sum <= 2'b00;
  else if(a == 3'b001 || a == 3'b010 || a == 3'b100)
    sum <= 2'b01;
  else if(a == 3'b011 || a == 3'b101 || a == 3'b110)
    sum <= 2'b10;
  else
    sum <= 2'b11;
end
```

Approach 3

A2	A1	A0	S1	S0
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Approach 4

- Here a for loop executes in "0-time" to determine the correct output
  - ***The key is that loops and if statements execute to determine/complete a truth table which is then synthesized to logic***
  - There is NO sequential logic inferred in this design, even though the description uses sequence
  - **Note**: You almost never need for loops to describe hardware (Other than maybe memory or register arrays)

```
 integer i;
 reg [2:0] a;
 reg [1:0] sum, cnt;
always @*
begin
  cnt = 0;
  for(i=0; i < 3; i=i+1)
  begin
    cnt = cnt + a[i];
  end
  sum <= cnt;
end
```

Approach 4

A2	A1	A0	S1	S0
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

UNDERSTANDING SIMULATION AND TIME

Understanding Simulation Timing

- When expressing parallelism, an understanding of how time works is crucial
- Even though ‘always’ and ‘assign’ statements specify operations to be run in parallel, simulator tools run on traditional computers that can only execute sequential operations
- To maintain the appearance of parallelism, the simulator keeps track of events in a sorted event queue and updates signal values at appropriate times, triggering more statements to be executed

Explicit Time Delays

- In testbenches, explicit delays can be specified using '# delay'
  - When this is done, the RHS of the expression is evaluated at time *t* but the LHS is not updated until *t+delay*

```
module m1_tb;
 reg  a,b,c;
 wire  w,x,y,z;
 initial begin
   a = 1;
   #5;  // delay 5 ns (ns = default)

   a = 0;
   b = 0;
   #2;  // delay 2 more ns

   a = 1;
 end
endmodule
```

Simulator Event Queue

Time	Event
0 ns	a = 1
5 ns	a = 0
5 ns	b = 0
7 ns	a = 1

Explicit Time Delays

- Assignments to the same signal without an intervening delay will cause only the last assignment to be seen

```
module m1_tb;
  reg  a,b,c;
  wire  w,x,y,z;

  initial begin
    a = 1;
    #5  // delay 5 ns (ns = default)
    a = 0;
    a = 1;
  end
endmodule
```

Simulator Event Queue

Time	Event
0 ns	a = 1
5 ns	a = ~~0~~→1

Explicit Propagation Delay

- When modeling logic, explicit propagation delays can be inserted
  - Normally behavioral descriptions should avoid this since the delays will be determined by the synthesis tools
- Verilog supports different propagation delay paradigms
- One paradigm is to specify the delay with the RHS of an assignment in an always block.
- When this is done, the RHS of the expression is evaluated at time *t* but the LHS is not updated until *t+delay*
- This is called "transport" delay since we are specifying the time to transport the value from inputs to output

```
module m1(a,b,c,w,x,y,z);
 input    a,b,c;
 output   w,x,y,z;

 always @(a,b,c)
 begin

   w <= #4 a ^ b;
   x <= #5 b | c;
 end

endmodule
```

Time	Event
0 ns	a,b,c = 0,0,1
4 ns	w = 0
5 ns	x = 1

Simulator Event Queue

Implicit Time Delays

- Normal behavioral descriptions don't model propagation delay until the code is synthesized
- To operate correctly the simulators event queue must have some notion of what happens first, second, third, etc.
- Delta (δ) time is used
  - Delta times are purely for ordering events and all occur in "0 simulated time"
  - The first event(s) occur at time 0 ns
  - Next event(s) occur at time 0 + δ
  - Next event(s) occur at time 0 + 2δ

```
always @(a,b,c,w,x,y)
begin
   w <= a ^ b;
   x <= b | c;
   y <= w & x;
   z <= ~y;
end
```

Equivalent Implementations

```
assign w = a ^ b;
assign x = b | c;
assign y = w & x;
assign z = ~y;
```

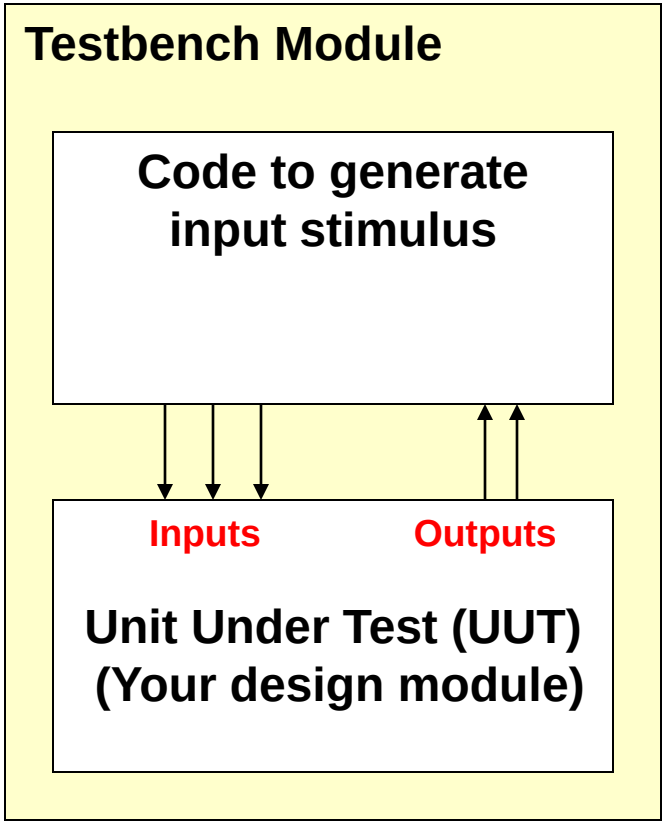
Time	Event	Triggers
0 ns	a,b,c = 0,0,1	w and x assigns
0 + δ	w=0, x=1	y assign
0 + 2δ	y = 0	z assign
0 + 3δ	z = 1	Anything sensitive to z

Simulator Event Queue

TESTBENCHES

Testbenches

- Generate input stimulus (values) to your design over time
- Simulator will run the inputs through the circuit you described and find what the output from your circuit would be
- Designer checks whether the output is as expected, given the input sequence
- Testbenches consist of code to generate the inputs as well as instantiating the design/unit under test and possibly automatically checking the results



Testbench Modules

- Declared as a module just like the design circuit
- No inputs or outputs

```
module my_tb;
  // testbench code
endmodule
```

Testbench Signals

- Declare signals in the testbench for the inputs and outputs of the design under test
  - inputs to your design should be declared type ‘reg’ in the testbench (since you are driving them and their value should be retained until you change them)
  - outputs from your design should be declared type ‘wire’ since your design is driving them

```
module m1(x,y,z,f,g);

  input  x,y,z;
  output f,g;

...
```

Unit Under Test

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

endmodule
```

Testbench

UUT Instantiation

- Instantiate your design module as a ***component*** (just like you instantiate a gate in you design)
- Pass the input and output signals to the ports of the design
- For designs with more than 4 or 5 ports, use named mapping rather than positional mapping

```
module m1(x,y,z,f,g);

  input  x,y,z;
  output f,g;
  ...
endmodule
```

Unit Under Test

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);
  /* m1 uut(.x(x), .y(y),
            .z(z), .f(f),
            .g(g));
  */
endmodule
```

Testbench

Generating Input Stimulus (Values)

- Now use Verilog code to generate the input values over a period of time

```
module m1(x,y,z,f,g);

  input  x,y,z;
  output f,g;
  ...
endmodule
```

Unit Under Test

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);
  /* m1 uut(.x(x), .y(y),
            .z(z), .f(f),
            .g(g));
  */
endmodule
```

Testbench

Initial Block Statement

- Tells the simulator to run this code just once (vs. always block that runs on changes in sensitivity list signals)
- Inside the "initial" block we can write code to generate values on the inputs to our design
- Use "begin…end" to bracket the code (similar to { .. } in C or Java)

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);

  initial
  begin

    // input stimulus
    // code

  end

endmodule
```

Testbench

Assignment Statement

- Use '=' to assign a signal a value
  - Can assign constants
    - x = 0; y = 1;
  - Can assign logical relationships
    - x = ~x // x = not x
    - x = y & z // x = y and z

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);

  initial
  begin

    x = 0;

  end

endmodule
```

Testbench

Aggregate Assignment Statement

- Can assign multiple signals at once
- Place signals in brackets (i.e. { }) and separate with commas
- Multiple bit constants can be written in the form:
- ***num_bits*** *'{b,o,d,h} value*
  - 4'b0000 // 4-bits **b**inary
  - 6'b101101 // 6-bits **b**inary
  - 8'hFF // 8-bits in **h**ex
  - Decimal is default
  - 17 // 17 decimal

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);

  initial
  begin

    {x,y,z} = 3'b000;

  end

endmodule
```

Testbench

Time

- We must explicitly indicate when and how much time should pass between assignments
- Statement ('#' indicates a time delay):
  - # 10; // wait 10 ns;
  - # 50; // wait 50 ns;
- Default timescale is nanoseconds (ns)

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 dut(x,y,z,f,g);

  initial
  begin

    {x,y,z} = 3'b000;
    #10;
    {x,y,z} = 3'b001;
    #25;

  end

endmodule
```

Testbench

Integer Signal Type

- To model a collection of bits representing a number, declare signals as type 'integer'
- Assigning an integer to a bit or group of bits will cause them to get the binary equivalent
- Assigning an integer value too large for the number of bits will cause just the LSB's of the number to be assigned
  - Assigning $8_{10}=1000_2$ to a 3-bit value will cause the 3-bit value to be 000 (i.e. the 3 LSB's of 1000)

```
module my_tb;

  reg     w,x,y,z;
  integer num;

  initial
    begin
      num = 15;
      {w,x,y,z} = num;
      // assigns
      // w,x,y,z = 1111
      #10;
      num = num+1;
      // num = 16
      {w,x,y,z} = num;
      // w,x,y,z = 0000
  end
endmodule
```

Testbench

For loop

- Integers can also be used as program control variables
- Verilog supports 'for' loops to repeatedly execute a statement
- Format:
  - for(initial_condition; end_condition; increment statement)

```
module my_tb;

  reg     a,b;
  integer i;

  initial
    begin
      for(i=0;i<4;i=i+1)
      begin
        {a,b} = i;
      end
    end
endmodule
```

You can't do "i++" as in C/C++ or Java

a,b = 00, then 01, then 10, then 11

Here, 'i' acts as a counter for a loop. Each time through the loop, i is incremented and then the decimal value is converted to binary and assigned to a and b

For loop

- **Question**: How much time passes between assignments to {a,b}
- **Answer**: 0 time…in fact if you look at a waveform, {a,b} will just be equal to 1,1…you'll never see any other combinations
- We must explicitly insert time delays!

```
module my_tb;

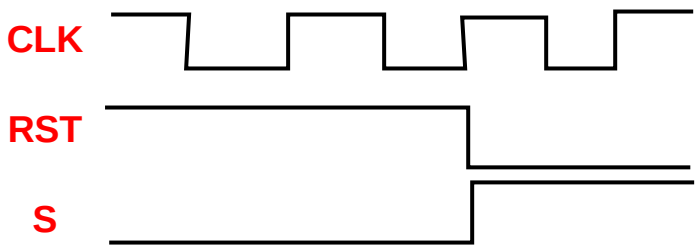
  reg     a,b;
  integer i;

  initial
    begin
      for(i=0;i<4;i=i+1)
      begin
        {a,b} = i;
        #10;
      end
    end
endmodule
```

Now, 10 nanoseconds will pass before we start the next iteration of the loop

Generating Sequential Stimulus

- Clock Generation
  - Initialize in an initial block
  - Continue toggling via an always process
- Reset generation
  - Activate in initial block
  - Deactivate after some period of time
  - Can wait for each clock edge via @(posedge clk)



Generated stimulus

```
module my_tb;
  reg     clk, rst, s;

  always #5 clk = ~clk;

  initial begin
   clk = 1; rst = 1; s=0;
   // wait 2 clocks
   @(posedge clk);
   @(posedge clk);
   rst = 0;
   s=1;
   @(posedge clk);
   s=0;
  end
endmodule
```

BLOCKING VS. NON-BLOCKING ASSIGNMENT

(Non) Blocking Assignment Overview

- There are two different assignment operators in Verilog (and most HDLs)
- Non-blocking (<=): ***Schedule an update for the value*** of a variable for the next possible simulation (aka delta) time
  - Similar to "propagation" delay
  - More common in hardware descriptions
- Blocking (=): ***Update the value*** of a signal/variable **immediately** (in current and simulated time)
  - Similar to assignment in software programming languages (variable immediately updates)
  - More common for simulation/testbenches

(Non-) Blocking Assignment Example 1

a is 0
Schedule a to be 1
Schedule a to be 1

```
reg [31:0] a;
initial a = 0;
always(@posedge clk)
begin
  a <= a+1;
  a <= a+1;
end
```

		Simulated Time	Scheduled Event
a	0	t=0	
a	0	t=0	@t=0+δ, a=1
a	0	t=0	@t=0+δ, a=1
a	1	t=0 +δ	

a is 0
Update a to be 1
Update a to be 2

```
reg [31:0] a;
initial a = 0;
always(@posedge clk)
begin
  a = a+1;
  a = a+1;
end
```

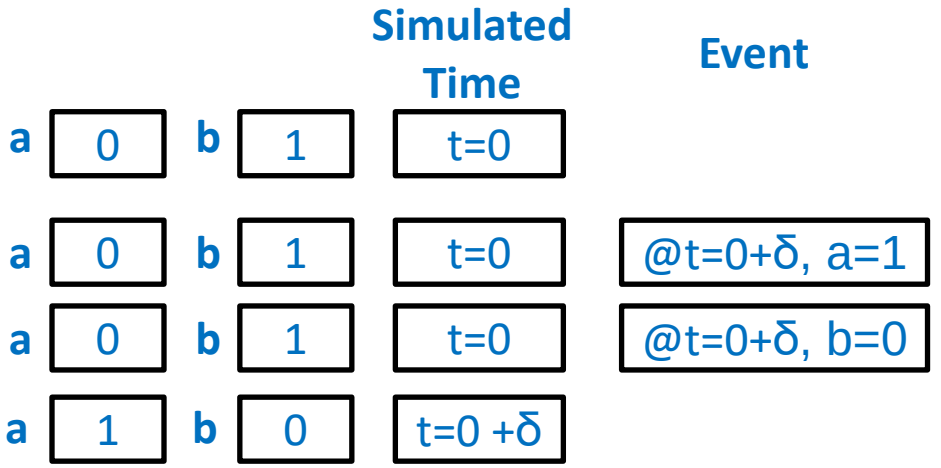
		Simulated Time	Scheduled Event
a	0	t=0	
a	1	t=0	
a	2	t=0	

(Non-) Blocking Assignment Example 2

Schedule a to be 1
Schedule b to be 0

```
reg a, b;

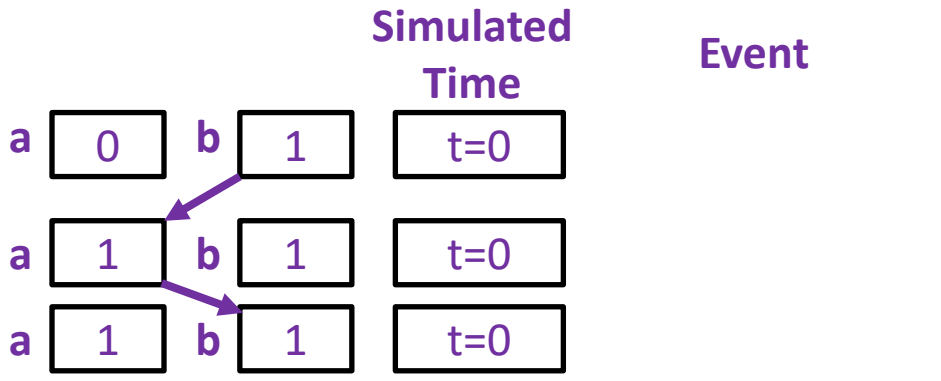
always(@posedge clk)
begin
  a <= b;
  b <= a;
end
```



Update a to be 1
Update b to be 1

```
reg a, b;

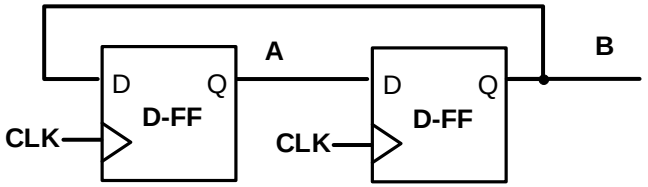
always(@posedge clk)
begin
  a = b;
  b = a;
end
```



Synthesis: Blocking vs. Non-Blocking

- **Non-blocking**: Each updated signal will result in a separate register (flip-flop)

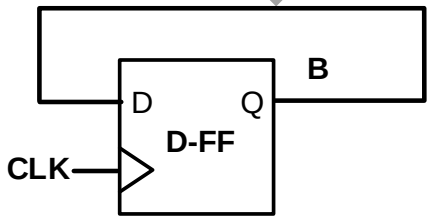
```
reg a,b;
always @(posedge clk)
begin
  a <= b;
  b <= a;
end
```



Synthesis Result with Non-Blocking Assignment

- **Blocking**: Due to the semantics of the blocking assignment, the code to the right results in a single register feedback the same value.

```
reg a,b;
always @(posedge clk)
begin
  a = b;
  b = a;
end
```



Synthesis Result with Blocking Assignment

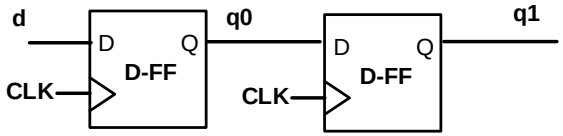
USC Viterbi School of Engineering

Non-Blocking Assignment and Registered Outputs

- General rule: Use non-blocking assignments when describing a registered (clocked) output
  - Your description should work equally well regardless of the order in which the simulator executes always blocks
  - With blocking assignments, different ordering may lead to different simulation results

```
reg d,q0,q1;
// block 1
always @(posedge clk)
begin
  q0 <= d;
end
// block 2
always @(posedge clk)
begin
  q1 <= q0;
end
```

Same simulation results regardless of execution order




```
reg d,q0,q1;
// block 1
always @(posedge clk)
begin
  q0 = d;
end
// block 2
always @(posedge clk)
begin
  q1 = q0;
end
```

If Block 2 executes first, same result as non-blocking (q1 = q0 & q0 = d)…LUCKY!
If Block 1 executes first, q1 = d…BAD!!

Review: Full-Adder Description

- Recall the description of a full-adder below
- Should we use blocking or non-blocking assignments for cnt?

```
 integer i;
 reg [2:0] a;
 reg [1:0] sum, cnt;
always @*
begin
  cnt = 0;
  for(i=0; i < 3; i=i+1)
  begin
    cnt = cnt + a[i];
  end
  sum <= cnt;
end
```

Approach 4

A2	A1	A0	S1	S0
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Review: Full-Adder Description

- Recall the description of a full-adder below
- Should we use blocking or non-blocking assignments for cnt?

```
 integer i;
 reg [2:0] a;
 reg [1:0] sum, cnt;
always @*
begin
  cnt = 0;
  for(i=0; i < 3; i=i+1)
  begin
    cnt = cnt + a[i];
  end
  sum <= cnt;
end
```

Approach 4

A2	A1	A0	S1	S0
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Missing Cases: Inferring Latches

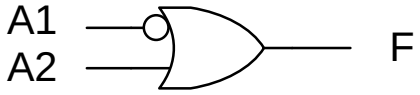
- Take care when describing combinational processes to cover all input cases or provide a default value

A2	A1	A0	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

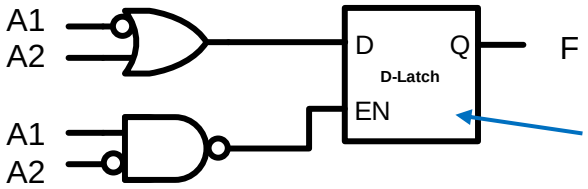
```
reg f;  reg [2:0] a;
always @*
begin
  if(a < 2) f <= 1;
  else if(a >= 4) f <= 1;
  else f <= 0;
end
```

```
reg f;  reg [2:0] a;
always @*
begin
  f <= 0;
  if(a < 2) f <= 1;
  else if(a >= 4) f <= 1;
end
```

```
reg f;  reg [2:0] a;
always @*
begin
  if(a < 2) f <= 1;
  else if(a >= 4) f <= 1;
  // missing else to cover
  // cases 2 (010) and 3 (011)
end
```



Correct Description: Synthesis/Simulation View



Incorrect Description: Synthesis/Simulation View

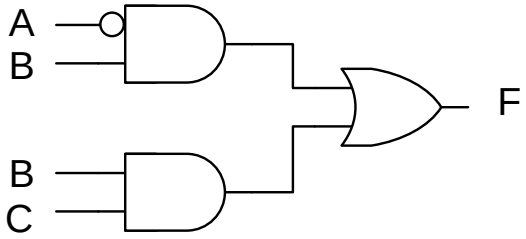
When A2,A1=01 our description indicates that f should retain its value (thus the latch)

Missing Cases: Sensitivity Lists

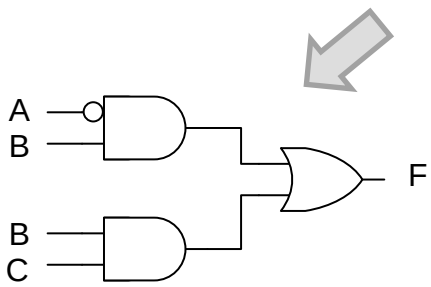
- Missing a signal in the sensitivity list of a combinational process will lead to strange simulation behavior (but generally correct synthesis)

```
reg f; wire a,b,c;
always @(a or b or c) // or @*
begin
  f <= 0;
  if( (a==0 && b==1) ||
      (b==1 && c==1) )
    f <= 1;
end
```

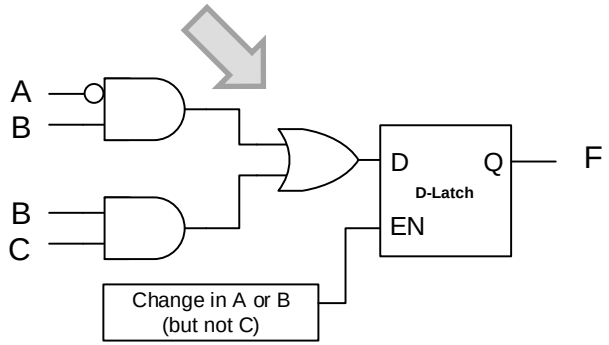
```
reg f; wire a,b,c;
always @(a or b)
Begin
  f <= 0;
  if( (a==0 && b==1) ||
      (b==1 && c==1) )
    f <= 1;
end
```



Correct Description: Synthesis/Simulation View



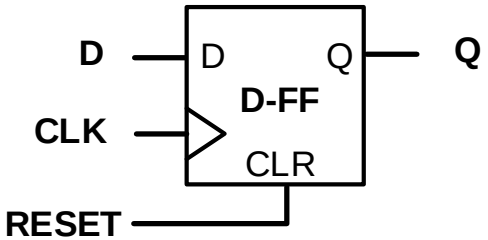
Synthesis View



Simulation View

Sensitivity List: Synchronous vs. Asynchronous Reset

- Asynchronous Reset:
  - Q is initialized when the reset signal is asserted (regardless of the clock state)
- Synchronous Reset
  - Q is initialized only on a clock edge AND the reset signal is asserted




```
always @(posedge clk, posedge reset)
begin
  if(reset == 1)
    q <= 0;
  else
    q <= d;
end
```

Asynchronous Reset

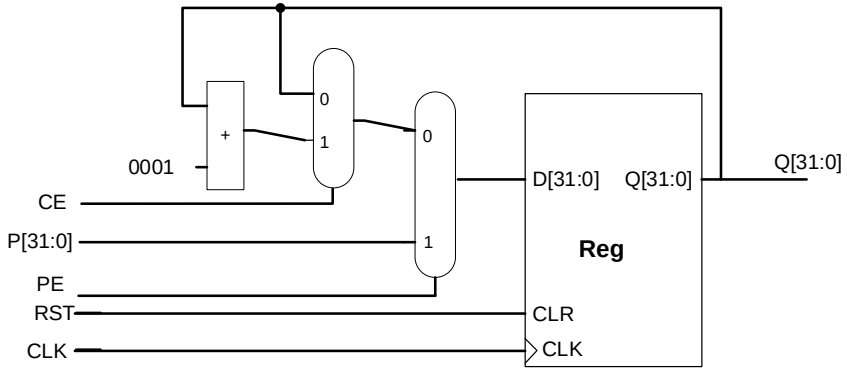
```
always @(posedge clk)
begin
  if(reset == 1)
    q <= 0;
  else
    q <= d;
end
```

Synchronous Reset

DESIGN APPROACHES

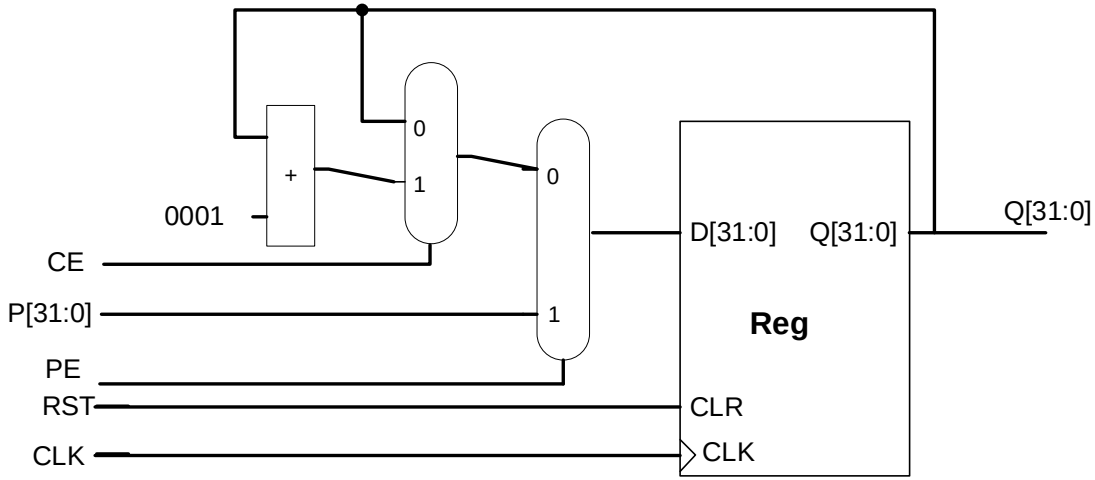
General Tips

- Don't start coding…start drawing
  - Sketch out the physical, component-level block diagram of the design
  - Identify what happens per clock (i.e. clock boundaries or where registers are needed)
- Partition your design
  - Identify repeated components and extract modules/hierarchy
  - See slides below to help organize how you will describe your logic
- Setup a testbench and test setup
  - Take time to create a useful waveform setup (.do) files



Partitioning 1

- Given the counter design below a few methods of partitioning are possible
- **Option 1**: Separate combinational and sequential process
- **Option 2**: Combined process



Correct Description: Synthesis/Simulation View

```
reg [31:0] q; wire [31:0] p;
wire ce, pe, rst, clk;
reg [31:0] d;
always @*
begin
  if(pe == 1) d <= p;
  else if(ce == 1) d <= q+1;
  else d <= q;
end
always @(posedge clk)
begin
  if(clr == 1) q <= 0;
  else q <= d;
end
```

Separate Processes

```
reg [31:0] q; wire [31:0] p;
wire ce, pe, rst, clk;

always @(posedge clk)
begin
  if(clr == 1) q <= 0;
  else if(pe == 1) q <= p;
  else if(ce == 1) q <= q+1;
end
```

Combined Processes

Traffic Light State Machine

```
module trafficlight(s1, s2, clk, rst, msg, ssg, mtg, msr,
     ssr, mtr);
    input s1, s2, clk, rst;
    output msg, ssg, mtg, msr, ssr, mtr;

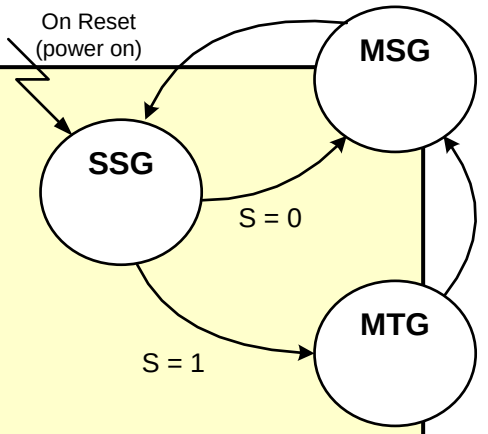
    reg   msg, ssg, mtg, msr, ssr, mtr;

    reg [1:0] state;
    reg [1:0] state_d;
    wire      s;
    parameter MT = 2'b11;
    parameter MS = 2'b10;
    parameter SS = 2'b00;

    assign s = s1 | s2;
    always @(state, s)
     begin
        if(state == MS)
          state_d = SS;
        else if(state == SS)
          if(s == 1)
            state_d = MT;
          else
            state_d = MS;
        else  // state == MT
          state_d = MS;
      end
```

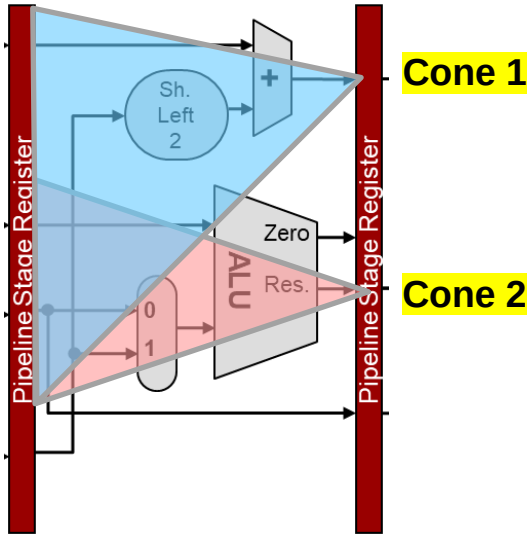
```
    always @(posedge clk)
    begin
         if(rst == 1)
           state <= SS;
         else
           state <= state_d;
    end

    always @(state)
    begin
         mtg = 0; msg = 0; ssg = 0;
         mtr = 0; msr = 0; ssr = 0;
         case(state)
                MT:
                  begin
                  mtg = 1; ssr = 1; msr = 1;
                  end
                MS:
                  begin
                  msg = 1; ssr = 1; mtr = 1;
                 end
                SS:
                  begin
                  ssg = 1; msr = 1; mtr = 1;
                  end
         endcase
     end
endmodule
```



Partitioning 2

- One approach to describing your design is to break it into **cones of logic**
  - A **cone** of logic ends with a registered signal/output and includes all the signals that feed that registered output (usually back to the previous registers or primary inputs)

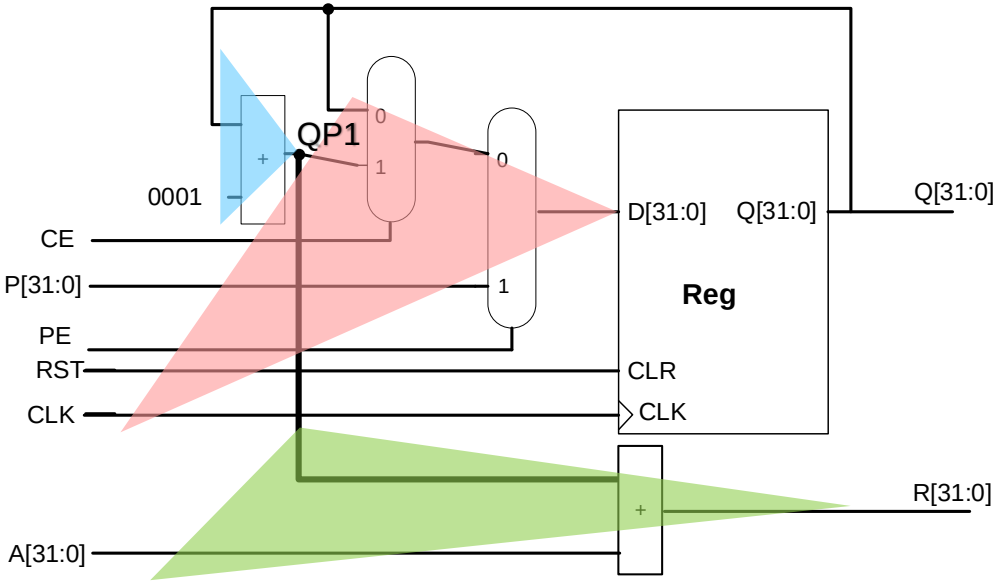



```
reg [31:0] q1, q2, a, b;
reg [2:0]  op;
wire rst, clk;

always @(posedge clk)
begin
  if(rst == 1) q1 <= 0;
  else
    q1 <= a + {b[29:0],2'b00};
end
always @(posedge clk)
begin
  if(rst == 1) q2 <= 0;
  else if(op == 3'b000)
    q2 <= a+b;
  else if(op == 3'b001)
    q2 <= a-b;
  ...
end
```

Partitioning 3

- When an output (**e.g. QP1 below**) is needed in multiple cones it is likely easiest to produce that in a separate process to produce an intermediate output that can be used in multiple other processes




```
reg [31:0] q;
wire [31:0] p, r, a, qp1;
wire rst, clk, ce, pe;

assign qp1 = q + 1;

always @(posedge clk)
begin
  if(rst == 1) q <= 0;
  else if(pe == 1) q <= p;
  else if(ce == 1) q <= qp1;
end

assign r = qp1 + a;
```

Partitioning 4

- For combinational processes (muxes and adders) consider using assign statements (over always blocks)

```
// mux
assign y = (sel == 1) ? a : b;

// adder
assign y = x + 1;
```