ECE 351
Verilog and FPGA Design

Lecture 9: RTL expression operations (wrap-up)
Procedural blocks

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Portland State UNIVERSITY

RTL Expression Operators (wrap-up)

Source material drawn from:
- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Portland State UNIVERSITY

Review: SystemVerilog Operators

Category	Examples	Bit Length
Bitwise	~A, A & B, A | B, A ^ B, A ^ ~B	L(*A*) MAX(L(*A*),L(*B*))
Logical	!A, A && B, A || B	1 bit
Reduction	&A, ~&A, |A, ~ |A, ^ ~ A, ~ ^ A	1 bit
Relational	A = = B, A != B, A > B, A < B A >= B, A <= B A = = = B, A != = B	1 bit
Arithmetic	A + B, A – B, A * B, A / B A % B	MAX(L(*A*), L(*B*))
Shift	A << B, A >> B	L(*A*)
Concatenate	{A,…,B}	L(*A*) + ··· + L(*B*)
Replication	{B{A}}	B * L(*A*)
Condition	A ? B : C	MAX(L(*B*),L(*C*))

Operator Precedence

Operator	Precedence
() [] :: .	highest
+ – ! ~ & ~& | ~| ^ ~^ ^~ ++ –– (unary ops)	
**	
* / %	
+ – (binary operators)	
<< >> <<< >>>	
< <= > >= inside dist	
== != === !== ==? !=?	
& (binary operator)	
^ ~^ ^~ (binary operators)	
| (binary operator)	
&&	
||	
?: (conditional operator)	
-> <->	
= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>= := :/ <= (assignment operators)	
{} {{}}	lowest

Bitwise Operators

Operator	Operation	**Examples** `ain = 3'b101, bin = 3'b110, cin = 3'b01x`
~	invert each bit	`~ain is 3'b010`
&	and each bit	`ain & bin is 3'b100, bin & cin is 3'b010`
|	or each bit	`ain | bin is 3'b111`
^	xor each bit	`ain ^ bin is 3'b011`
~^ or ^~	xnor each bit	`ain ^~ bin = 3'b100`

- Operates on each bit of the operand
- Result is the size of the largest operand
- Left-extended if sizes are different
- Bitwise operators are X-optimistic

Bitwise Operators (cont'd)

Table 5-5: Bitwise AND truth table

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 5-6: Bitwise OR truth table

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Table 5-7: Bitwise XOR truth table

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Table 5-8: Bitwise exclusive NOR truth table

^~ ~^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Reduction operators

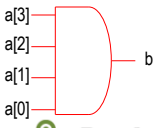
Table 5-9: Reduction operators for RTL modeling

Operator	Example Usage	Description
&	& m	AND all bits of m
~&	~& m	NAND all bits of m
|	| m	OR all bits of m
~|	~| m	NOR all bits of m
^	^ m	Exclusive-OR all bits of m
~^ ^~	~^ m	Exclusive-NOR all bits of m

- Models gates yielding a single output bit
- Reduction operators are X-Optimistic

```
logic[3:0] a;
Logic b;
assign b = &a;
```

⇔



Reduction operators (cont'd)


```
// User-defined type definitions
package definitions_pkg;
  typedef struct {
    logic [3:0] data;
    logic       parity_bit;
  } data_t;
endpackage: definitions_pkg

// Parity checker using even parity (the combined data value
// plus parity bit should have an even number of bits set to 1
module parity_checker
 import definitions_pkg::*;
(input  data_t data_in,  // 5-bit structure input
 input  clk,             // clock input
 input  rstN,            // active-low asynchronous reset
 output logic  error     // set if parity error detected
);

  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN) error <= 0;                 // active-low reset
    else       error <= ^{data_in.parity_bit, data_in.data};
      // reduction-XOR returns 1 if an odd number of bits are
      // set in the combined data and parity_bit
endmodule: parity_checker
```

Logical Operators

Operator	Example Usage	Description
&&	m && n	Logical AND: Is m true AND is n true?
||	m || n	Logical OR: Is m true OR is n true?
!	! m	Logical negate: Is m not true?

- Always evaluate to a 1-bit value 0 (false), 1 (true) or x (ambiguous)
- If a result is not equal 0 it is logical 1 (true), if 0 (false) if x (ambiguous)
- Take variables or expressions as operands
- Perform their operations by first doing a logic OR reduction of each operand (1-bit result) and then evaluates the result to determine if expression is true or false.
- For negate operation the 1-bit result is inverted and then evaluated as true or false

Logical operators (cont'd)

Operand 1	Operand 2	&&	||
4'b0000	4'b0000	1'b0	1'b0
4'b0000	4'b1000	1'b0	1'b1
4'b0000	4'b00zx	1'b0	1'bx
4'b0000	4'b01zx	1'b0	1'b1
4'b1000	4'b0000	1'b0	1'b1
4'b1000	4'b1000	1'b1	1'b1
4'b1000	4'b00zx	1'bx	1'b1
4'b1000	4'b01zx	1'b1	1'b1

Operand 1	!
4'b0000	1'b1
4'b1000	1'b0
4'b00zx	1'bx
4'b01zx	1'b0

- Differences between negate (!) and invert (~):
  - Negate – performs a true/false evaluation of its operand and returns a 1-bit value (true, false, unknown)
  - Invert – logical inversion of each bit of an operand (one's complement)

Sutherland: Tables 5-12, 5-13

Your turn: Logical operators (cont'd)

```
logic       enable;   // 1-bit control signal
logic [1:0] select;   // 2-bit control signal

assign enable = 1'b1;
assign select = 2'b01;

if (!enable) ... //
if (~enable) ... //

if (!select) ... //
if (~select) ... //
```

Differences between negate (!) and invert (~):
- Negate – performs a true/false evaluation of its operand and returns a 1-bit value (true, false, unknown)
- Invert – logical inversion of each bit of an operand (one's complement)

Comparison operators

Operator	Example Usage	Description
==	m == n	Equality: Is m equal to n?
!=	m != n	Not Equality: Is m not equal to n?
<	m < n	Less-than: Is m less than n?
<=	m <= n	Less-than or equal: Is m less than or equal to n?
>	m > n	Greater-than: Is m greater than n?
>=	m >= n	Greater-than or equal: Is m greater than or equal to n?

- Work like they do in C
- If relational operators are used in an expression the expression returns a logical value of 1'b1 if true and 1'b0 if false
- Can do signed compare (only if both operands are signed) or unsigned (either operand is unsigned)
- Are X-pessimistic (if either operator has even a single X or Z bit the result is X even though actual logic gate behavior would be more optimistic…YMMV

Case equality operators

Operator	Example Usage	Description
===	`m === n`	Case equality: Is `m` identical to `n`?
!==	`m !== n`	Not case equality: Is `m` not identical to `n`?
==?	`m ==? n`	Wildcard case equality: Is `m` identical to `n`, after masking?
!=?	`m !=? n`	Wildcard not case equality: Is `m` not identical to `n`, after masking?

- === !== are similar to == and != except all bits are compared for all 4 values (0, 1, Z, X) but may not be supported by your specific synthesis tool
- ==? and !=? Compare the bits of two values w/ ability to mask out (keep from being compared) specific bits (specified w/ X, Z, or ? for the masked bits. Typically synthesizable

Sutherland: Table 5-16

Case equality operators (cont'd)

```
// Set high_addr flag if all bits of upper byte of address
// are set
//
module high_address_check
(input  logic        clk,       // clock input
 input  logic        rstN,      // active-low async reset
 input  logic [31:0] address,   // 32-bit input
 output logic        high_addr  // set high-byte all ones
);

  always_ff @(posedge clk or negedge rstN) // async reset
    if (!rstN)                             // active-low reset
      high_addr <= '0;
    else
      high_addr <= (address ==? 32'hFF??????); // mask low bits
endmodule: high_address_check
```


Sutherland: Example 5-9

Set membership (inside)

Operator	Example Usage	Description
inside {}	`m inside {0, 1, n}`	Set membership: Does m match 0, 1 or n?

- Compares an expression to a set of other expressions enclosed in {..,..,..} and returns true (1'b1) if the first expression matches any of the expressions in the set, false (1'b0) if there are not matches
- Can also be modeled using logical OR operators but not as concise
- Can specify a range of values ([m : n]) instead of specific values
- Can allow bits in the value list to be masked using X, Z, or ? (like ==?)
- Can include variables, expressions, arrays in value list
- Can be used in both continuous assignment statements and in procedural blocks
- Can be synthesized but check the HDL coding guide for your synthesis tool

Sutherland: Table 5-17

Set membership (cont'd)

```
always_ff @(posedge clk)
  if (address inside {0, 32, 64, 128, 256, 512, 1024})
    boundary <= '1;
  else
    boundary <= '0;
```

```
always_ff @(posedge clk)
 boundary <= address inside {0, 32, 64, 128, 256, 512, 1024};
```

```
always_comb begin
  small_value = data inside {[0:255]};
end   // true if data matches a value between 0 and 255
```

```
 always_comb begin
   boundary_flag = (address inside {(0),            // quad 1
                                    (((2**N)/4)*1), // quad 2
                                    (((2**N)/4)*2), // quad 3
                                    (((2**N)/4)*3)  // quad 4
                                   } );
 end
```


Sutherland: Section 5-9 examples

Shift Operators

Operator	Example Usage	Description
>>	m >> n	Bitwise right shift: shifts m right n times
<<	m << n	Bitwise left shift: shifts m left n times
>>>	m >>> n	Arithmetic right shift: shifts m right n times, preserving the value of the sign bit of a signed expression
<<<	m <<< n	Arithmetic left shift: shifts m left n times (same result as bitwise left shift)

- Shifts a vector left or right some number of bits
- Arithmetic shift right fills with sign bit
- All others fill with zeros
- Shifted bits are lost
- Are synthesizable whether shift amount is a constant or a variable

Sutherland: Table 5-18

Arithmetic Shift Operators: <<<, >>>

- The >>> arithmetic right-shift performs sign-extension
  - If shifted expression is signed, >>> fills the vacated left-most bits with the value of the sign bit
  - If shifted expression is unsigned, >>> fills the vacated left-most bits with zero (just like >>)
- The <<< arithmetic left-shift is synonymous with << logical left-shift
  - Fills vacated right-most bits with zero
  - Provided for consistency

Variable Shifts

- Constant operand with variable shift:

```
wire [7:0] Out;
wire [2:0] In;
assign Out = 1'b1 << In; // 3-to-8 decoder
```

  - In other words, Out = 2^{In}

Streaming Operators (Pack and Unpack)

- Takes expression, structure, or array and packs or unpacks it into bits

Operator	Example Usage	Description
{>> {}}	{>>m{n}}	Right stream (extract) m-size blocks from n, working from the right-most block towards the left-most block
{<< {}}	{<<m{n}}	Left stream (extract) m-size blocks from n, working from the left-most block towards the right-most block

- Operators pull-out or push-in groups of bits from or to a vector in a serial stream
- Packing occurs when the streaming operator is used on the RHS of an assignment (pulls blocks as a serial stream from the RHS and packs the stream into a vector on the LHS
- Unpacking occurs when a streaming operator is used on the LHS of an assignment (pulls bits from RHS and assigned to the LFS
- ex: reversing the bits in a vector:

```
logic [7:0] a, b;
assign a = 8'b11000101
always_comb begin
     b = { << {a} };  // sets b to 8'b10100011 (bit reverse of a
```


Sutherland: Section 5.11

Streaming Operators (cont'd)

- Can use stream operators to copy:
  - Unpacked arrays to packed arrays
  - Packed arrays to unpacked arrays
  - Unpacked arrays to different size unpacked arrays
- Takes expression, structure, or array and packs or unpacks it into bits
  - >> streams data from left to right
  - << streams data from right to left

```
initial begin
  int h;
  bit [7:0] b, g[4], j[4] = '{8'ha, 8'hb, 8'hc, 8'hd};
  bit [7:0] q, r, s, t;

  h = { >> {j}};          // 0a0b0c0d - pack array into int
  h = { << {j}};          // b030d050 reverse bits
  h = { << byte {j}};     // 0d0c0b0a reverse bytes
  g = { << byte {j}};     // 0d, 0c, 0b, 0a unpack into array
  b = { << {8'b0011_0101}};   // 1010_1100 reverse bits
  b = { << 4 {8'b0011_0101}}; // 0101_0011 reverse nibble
  {>> {q, r, s, t}} = j; // Scatter j into bytes
```

Sutherland: Table 5-18

Arithmetic Operators

Operator	Example Usage	Description
+	m + n	Add: add the value of m to the value of n
-	m - n	Subtract: subtract the value of n from the value of m
-	-m	Unary minus of the value of m (two's complement of m)
*	m * n	Multiply: multiply the value of m by the value of n
/	m / n	Divide: divide the value of m by the value of n
%	m % n	Modulus: remainder of of m divided by n
**	m ** n	Power: value of m raised to the power of the value of n

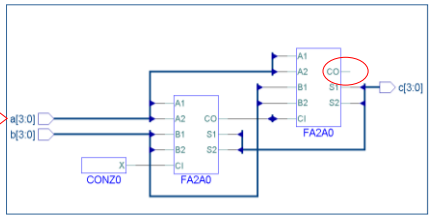
- Treats vectors as a whole value
- If any operand is z or x, then the results are unknown
  - Example: `ain + 2'b0z = unknown`
- All are synthesizable but technology and tools may have restrictions (particularly on *, /, %, **)

Sutherland: Table 5-20

Watch your bit widths…

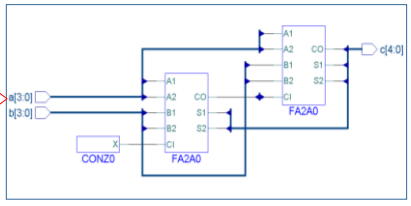
```
module lost_carry (a,b,c);
   input [3:0] a, b;
   output [3:0] c;
   assign c = a + b;
endmodule
```

Synthesizes to




```
module carry_out (a,b,c);
   input [3:0] a, b;
   output [4:0] c;
   assign c = a + b;
endmodule
```

Synthesizes to



If results and operands are same size, then carry is lost

Portland State UNIVERSITY

Other SystemVerilog operators

Table 5-21: Increment and decrement operators for RTL modeling

Operator	Example Usage	Description
++	++n n++	pre-increment the value of n by 1, or post-increment n by 1
--	--n n--	pre-decrement the value of n by 1, or post-decrement n by 1

Table 5-22: Assignment operators for RTL modeling

Operator	Example Usage	Description
+=	n += m	add m to n and assign result to n
-=	n -= m	subtract m from n and assign result to n
*=	n *= m	multiply n by m and assign result to n
/=	n /= m	divide n by m and assign result to n
%=	n %= m	divide n by m and assign remainder to n
&=	n &= m	bitwise AND m with n and assign result to n
|=	n |= m	bitwise OR m with n and assign result to n
^=	n ^= m	bitwise Exclusive OR m with n and assign result to n
<<=	n <<= m	bitwise left-shift n by the number of times indicated by m and assign result to n
>>=	n >>= m	bitwise right-shift n by the number of times indicated by m and assign result to n
<<<=	n <<<= m	arithmetic left-shift n by the number of times indicated by m and assign result to n
>>>=	n >>>= m	arithmetic right-shift n by the number of times indicated by m and assign result to n

Portland State UNIVERSITY

Procedural Blocks

Source material drawn from:
- Roy's ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Procedural Blocks

- `initial` block - Used to initialize behavioral statements for simulation
  - Executes once at simulation time 0
- `always` block(s) - Used to describe the circuit functionality using behavioral statements
  - Executes once every time the block is triggered
- Processes run in parallel and start at simulation time 0
  - Blocks can be scheduled/executed in any order
  - Statements inside a process execute sequentially
- `always` and `initial` blocks cannot be nested
- SystemVerilog supports 4 types of always blocks

SystemVerilog always blocks

- General purpose always block:
  - `always @(<sensitivity list>)`
  - Can model many types of functionality both synthesizable and not synthesizable (ex: clock oscillators, verification code)
  - Downside is that simulators and synthesis tools cannot always determine when the intended usage is
    - Result: synthesis places coding restrictions so it can accurately translate RTL -> netlist
- Specialized always blocks:
  - `always_comb` – for combinational logic
  - `always_latch` – for latch-based (level) sequential logic
  - `always_ff @(<sensitivity list>)` – for edge-triggered sequential logic
  - Also place coding restrictions for synthesis but provide insight into intended usage

Sensitivity lists

- Always blocks tell simulation that functionality being modeled should "always" be evaluated (an infinite loop)
- Simulation and synthesis need more information to:
  - Accurately model hardware behavior
  - Know when to execute the statements in the block
  - For RTL modeling when is:
    - On a clock edge – sequential logic
    - Any of the signals used in the block change value – combinational or latched logic
- The sensitivity list provides the when
  - The list of signals that trigger the execution of the block when they change
  - General purpose `always` and `always_ff` require an explicit sensitivity list
  - `always_comb` and `always_latch` infer an implicit sensitivity list

Explicit sensitivity lists

- Explicit sensitivity list introduced w/ @
- Contains a list of one or more net or variable names separated by either a comma (,) or the keyword or
- Can also specify the edge of a scalar (1-bit) signal using the keywords posedge or negedge
  - posedge (positive edge): 0->1, 0->z, 0->x, z->1, x->1, z->x, x->z
  - negedge (negative edge): 1->0, 1->z, 1->x, z->0, x->0, z->x, x->z
- Examples:
  - always @(a, b, c) or always @(a or b or c) …
  - always @(posedge clk or negedge rstN)…
  - always_ff @(posedge clk)
    ```
    q <= d;        // sequential flip-flop
    ```
  - always_ff @(posedge clk or negedge rstN)
    ```
    if (!rstN) q<= '0;
    else q <= d;
    ```


Implicit sensitivity lists

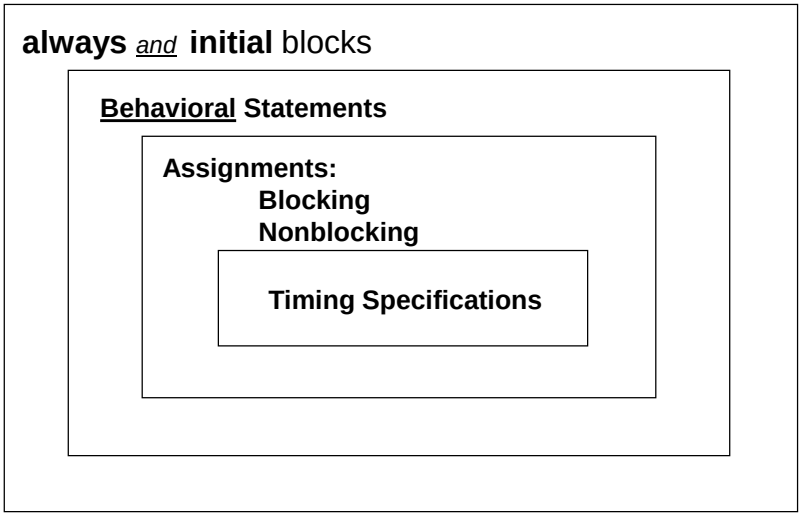
- Combinational logic:
  - Form of combinational logic that cannot store its current state
  - Combinational logic statements need to be re-evaluated (the when) whenever any input to that logic changes values
  - general purpose always needs an explicit sensitivity list that includes all of the signals on the RHS of all the expressions
    - Verilog 2001 supported @* which did this
  - always_comb automatically infers a proper combinational logic sensitivity list
  - Examples:
    - ```
      always @(a, b)
            sum = a + b;
      ```
    - ```
      always_comb
            sum = a + b;
      ```
    - ```
      always @*
            sum = a + b;
      ```
      ```

Implicit sensitivity lists (cont'd)

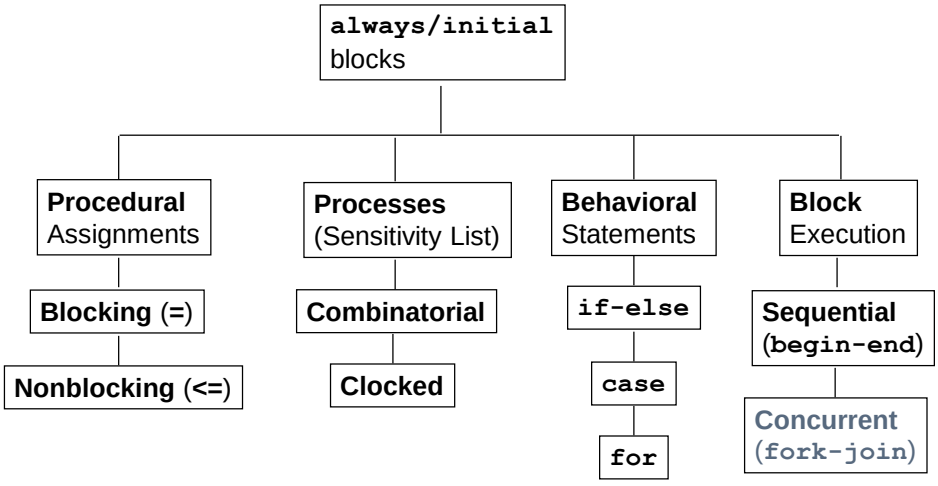
- Latched logic:
  - Form of combinational logic block that can store its current state
  - Latched logic statements need to be re-evaluated (the when) whenever any input to that logic changes values
  - general purpose always needs an explicit sensitivity list that includes all of the signals on the RHS of all the expressions
    - Verilog 2001 supported @* which did this
  - `always_latch` automatically infers a proper combinational logic sensitivity list
  - Examples:

```
• always @(enable, data)
        if (enable) out <= data;
• always_latch
        if (enable) out <= data;
• always @*
        if (enable) out <= data;
```


Elements of a Procedural Block



always and initial blocks



Statement groups

- All procedural block can contain either a single statement or a single group of statements
- Single statements can be nested within another statement

```
always @(posedge clk)
   if (enable)    // single outer statement
      for (int i; i <= 15; i++)     // nested statement
         out[i] = a[i] ^ b[(N - 1) – i];  // another nested stmt
```

- A group of statements is contained between the keywords `begin` and `end`
  - Can include any number of statements…including 1

```
always_comb begin //begin-end is the single group
    sum = a + b;
    dif = a – b;
end
```

Statement groups (cont'd)

- A `begin..end` group can be named using the syntax:

  `begin: <name>…end: <same name>`

- A named statement group can contain local variable declarations which can be used within the statement group but do not exist outside the group

```
always_ff @(posedge clk) begin: two_steps
       logic [7:0] tmp;      // local temporary variable
       tmp <= a + b;
       out <= c - tmp;
end: two_steps
```


Initial Block

- Consists of behavioral statements
- If there are more than one behavioral statement inside an `initial` block, the statements need to be grouped using `begin…end`
- An `initial` block starts at time 0, executes only once during simulation, and then does not execute again
- Assignment statements within an `initial` block execute sequentially - therefore statement order matters
- If there are multiple `initial` blocks, each block executes concurrently at time 0
- Used for initialization, monitoring, waveforms and other processes that must be executed only once during simulation

Initial Block Example

```
module system;
logic a, b, c, d;

// single statement
initial
       a = 1'b0;

/* multiple statements
   need to be grouped */
initial
    begin
            b = 1'b1;
       #5   c = 1'b0;
       #10 d = 1'b0;
    end

initial
    #20  $finish;
endmodule
```

Time	Statement Executed
0	a = 1'b0; b = 1'b1;
5	c = 1'b0;
15	d = 1'b0;
20	$finish

Why?

Always Block

- Used to model a process that is repeated continuously in a digital circuit…but you knew that
- If there are more than one behavioral statement inside an `always` block, the statements need to be grouped using `begin…end`
- An `always` block starts at time 0 and executes the behavioral statements continuously in a looping fashion
- Assignment statements inside an `always` block execute sequentially
  - Therefore, like an `initial` block, statement order matters*

* *sort of…*

Always Block - Example


```
module clock_gen (output logic clk);

parameter period=50, duty_cycle=50;

initial
   clk = 1'b0;

always
   #(duty_cycle*period/100)  clk = ~clk;

initial
   #100  $finish;

endmodule
```

Time	Statement Executed
0	clk = 1'b0
25	clk = ~clk // clk=1
50	clk = ~clk // clk=0
75	clk = ~clk // clk=1
100	$finish

Procedural Assignments

- Update values of variable types including user-defined types based on variables
  - Assigning to a net type (wire, tri, etc.) is illegal inside a procedural block
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value
- Only the LHS of a procedural assignment must be a variable. The RHS of assignments can use variables, nets, parameters or literal values

```
wire [15:0] a, b;     // net types
logic [15:0] sum;     // variable types

always_comb begin: adder
    sum = a + b;   // sum must be a variable type
end: adder
```

Two types of procedural assignments

- Blocking Assignment (=) : executed in the order they are specified in a procedural block
- Non-blocking Assignment (<=) : allow scheduling of assignments without blocking execution of the statements that follow in a procedural block

Blocking vs. Nonblocking Assignments (cont'd)

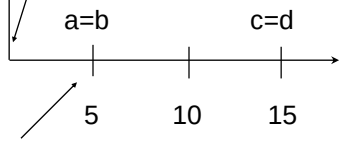
- Blocking ("="): `y = a & b;`
  - Completes the assignment before moving on to the next statement
  - "Blocks" other assignments until the current assignment has completed
  - Results highly dependent on order of assignments
- Nonblocking ("<="): `y <= a & b;`
  - Does not "block" execution of other assignments
  - Evaluates the RHS at the beginning of a simulation "tick"
  - Schedules the LHS update for the end of the "tick"
  - Results less dependent on order of assignments

Blocking vs. Nonblocking Assignments (cont'd)

Blocking (=)

```
initial
    begin
       a = #5 b;
       c = #10 d;
    end
```

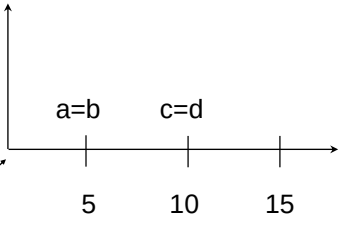
Value of **b** at t=0 is read, update of **a** is scheduled for 5 time units, execution of the initial block stops until then



Value of **d** read here, c update scheduled for 10 time units later

Nonblocking (<=)

```
initial
    begin
       a <= #5 b;
       c <= #10 d;
    end
```



Values of **b** and **d** at t=0 are read, updates of **a** and **c** then occur when scheduled

Why t=0?

Non-Blocking Assignments

- All right-hand sides across the entire design are evaluated before any left-hand sides are updated
- The order of right-hand sides evaluated and left-hand sides updated can be arbitrary
- Non-blocking assigns allow us to:
  - Handle concurrent specification in major systems
  - Reduce the complexity of our descriptions
  - Attach lots of concurrent actions to a single event – usually a clock

```
always @(posedge clock) begin
      q0 <= in | q1;
      q1 <= in & q0;
end
```

Blocking vs. Nonblocking Example


```
module nonblocking_example;

logic a = 0, b = 1;

// Two concurrent always blocks with blocking assignments
// always blocks run in any order…inconsistent results
always_ff @(posedge clock)
       a = b;
always_ff @(posedge clock)
       b = a;

// Two concurrent always blocks with nonblocking assignments
// works as expected because <= uses "old" values of b and a
always_ff @(posedge clock)
       a <= b;

always_ff @(posedge clock)
       b <= a;
endmodule: nonblocking_example
```


Which assignment should I use?

- Recommended:
  - Use non-blocking assignments for modeling clocked processes in sequential logic.
  - Use blocking assignments for modeling combinational logic
  - More on this later

Next Time

- Topics:
  - Decision statements
  - Looping statements
- You should:
  - Read Sutherland Ch 6
- Homework, projects and quizzes
  - Homework #2 will be posted today. Due by 10:00 PM on Mon, 04-May
    - No assignments will be accepted after Noon on Tue, 05-May so we can go over solution in class before the exam
  - Expect a quiz on Thursday
  - Midterm e-Exam will be 2:00 PM – 3:45 PM on Thu, 07-May