Quick Verilog

John Tramel
CECS
CSULB

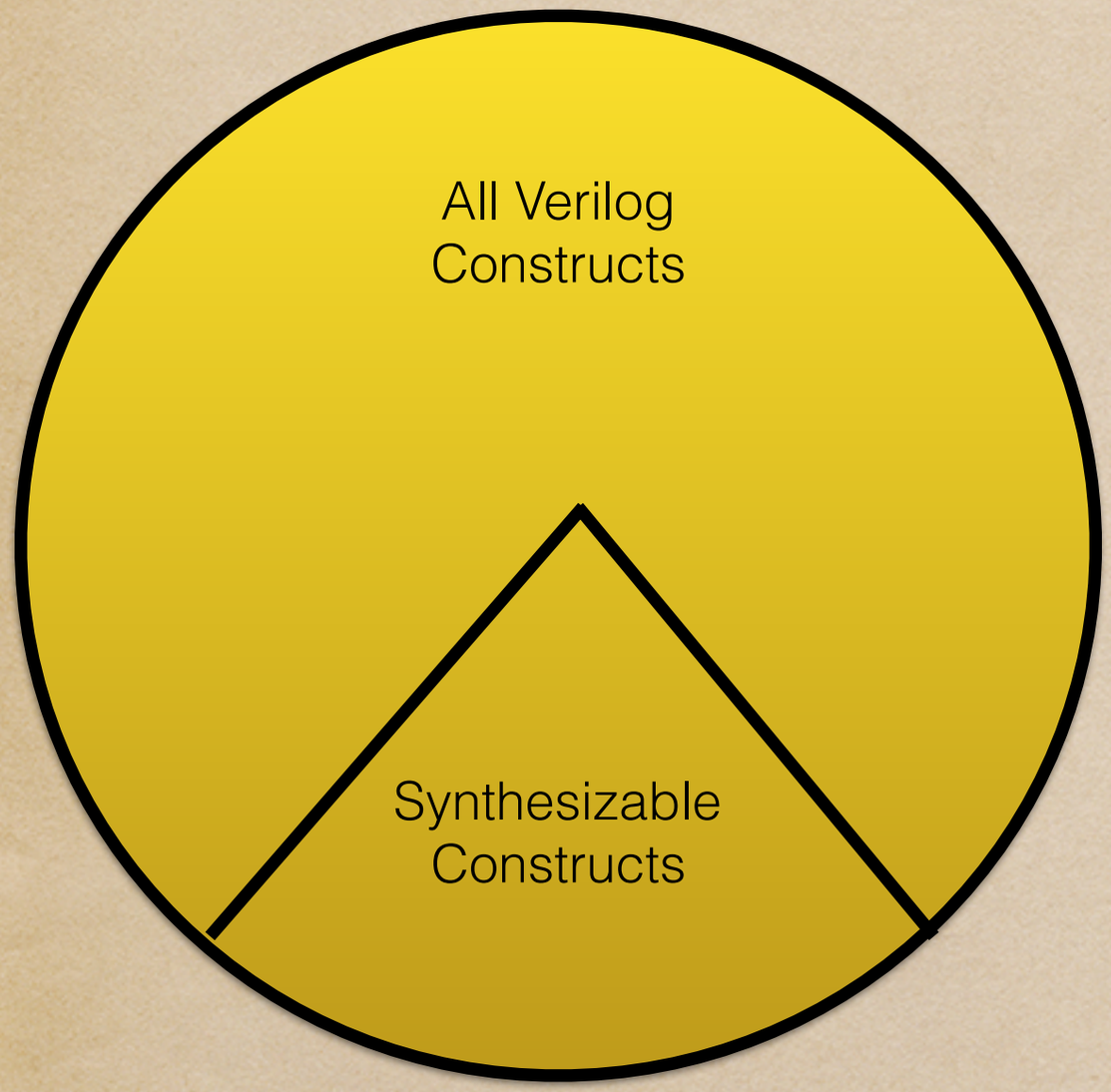
Brief History

- Designers originally defined digital circuits using a symbolic representation (schematics)
- In the 1980's simulators were developed that would check the correct syntax and provided a means to verify the correct functionality - problem is that when creating the schematic every single connection must be defined (very tedious!)
- In the 1990's Hardware Description Languages (HDLs) became popular where the digital circuit could be defined in a textual abstraction. The simulators developed for HDLs then compiled the design and again provided a means to verify the functionality. HDLs allow the designer to concentrate on functionality.
- The IEEE Std. 1364-2001, nicknamed "Verilog 2001", is the latest update to the Verilog Hardware Description Language

Verilog

- Verilog is an HDL that is used to model and simulate digital circuits
- VHDL is another HDL that also models digital logic
- The same language will be used to model the digital circuit and to build the test environment around the circuit (testbench)
- Verilog allows the designers to define both combinational and sequential circuits: several styles will be shown for modeling combinational circuits, one style will be shown for modeling sequential circuits
- Verilog “borrows” syntactic constructions from “C”
- Verilog is a procedural language: this implies the concept of a “procedural flow”

Verilog Applicability



- Verilog HDL is a general-purpose modeling language.
- We can use any construct for the development of a test bench
- A subset of Verilog is used for modeling our chip designs
- This subset is referred to as “synthesizable Verilog”

Verilog Values and Data Types

- Verilog is a four value logic system
  - (0: logic 0, low) (1: logic 1, high) (x: unknown) (z: high impedance)
- Two basic data types are registers and nets
  - register: retains a value, similar to a variable in a High Level Language
    - integer: 32 bit signed register
    - reg: register of user defined dimension
  - net: similar to a physical wire in a circuit - its value is determined by the output of the device driving the net
    - wire: net of user defined dimension

```
wire        x;       // scalar wire named ‘x’
wire [31:0] data;    // 32 bit bus with msg = data[31]
wire [15:0] b1, b2;  // two 16 bit busses
wire [ 4:2] split;   // 3 bit bus
reg         a;       // scalar register
reg  [31:0] a, b, c; // Three 32 bit registers
reg  [0:15] flip;    // 16 bit register w/ msg = flip[0]
```

Numeric Representations

- Default base in Verilog is decimal
- Numbers can be specified as binary, hex, octal and decimal
- Integer values represented by: <size>'<base> <value> (tick next to ENTER key)
  - size: the number of bits represented (optional, for documentation purposes only)
  - base: decimal (d), hex (h), octal (o), binary (b). This specifies base of value
  - value: value of number represented in terms of specified base
- Examples

```
16              // integer with value of 16₁₀
4'hA            // 4 bit integer with value of 10₁₀
8'b1010_0101    // 8 bit integer with value of 165₁₀
9'o567          // 9 bit integer with value 375₁₀
32'hff00        // 32 bit integer with value 65280₁₀
32'bx           // 32 bits all x
'bz             // high z (fill LHS with z's)
```

Verilog Building Block

- The basic building block in Verilog is the module
  - Delimited by keywords **module** and **endmodule**
  - Represents autonomous processes with defined interfaces
  - A Verilog design is composed of one or more modules
- The circuit being modeled and the test bench will be modules
- The structure of a Verilog module:

Header

Functionality

```
module andgate (a, b, y);  // keyword, module name, port list
input  a;
input  b;
output y;                  // define all ports as input, output or inout

wire   y;                  // define data types

assign y = a & b;          // define with continuous assignment

endmodule
```

Verilog - 2001 Building Block

- Combine data type declaration with port declaration

```
module andgate (a, b, y);
input  wire a;   // define functionality of a
input  wire b;   // define functionality of b
output wire y;   // define functionality of y

assign y = a & b; // define with continuous assignment

endmodule
```

- Combine data type declaration with port list

```
module andgate (a, b, y);
input  wire a,
input  wire b,
output wire y);   // keyword, module name, port list

assign y = a & b; // define with continuous assignment

endmodule
```

Defining Combinational Logic using Verilog

- There is more than one technique for modeling combinational logic using Verilog
  - Instantiating Verilog “built-in” primitives
    - and, nand, nor, or, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1
    - Similar to schematics except with text description
  - Continuous Assignment statements - defines a relationship, not included in a procedural block
  - Procedural block
    - always is the procedural block used to model circuits
    - Implies a “flow” through the description with implied trigger points defined in the “sensitivity list”
- Switching characteristics assumed to be ideal (no delay)
  - Verilog supports several techniques to include delay into analysis

Three Scalar Multiplexers

```
module mux1 (a, b, s, y);
input a, b, s;
output y;
wire y;
assign y = s ? b : a;

endmodule
```

Continuous Assignment

```
module mux3 (a, b, s, y);
input a, b, s;
output y;
reg y;
always @(a or b or s)
   if (s) y = b; else
          y = a;

endmodule
```

Procedural (always block)

```
module mux3 (input wire a,
             input wire b,
             input wire s,
             output reg y);

always @(*)
   if (s) y = b; else
          y = a;

endmodule
```

Verilog 2001 Syntax

Decoder

```
module decoder (s, y);
input  [2:0] s;
output [7:0] y;
wire   [7:0] y;

assign y =
   (s == 3'b000) ? 8'h01 :
   (s == 3'b001) ? 8'h02 :
   (s == 3'b010) ? 8'h04 :
   (s == 3'b011) ? 8'h08 :
   (s == 3'b100) ? 8'h10 :
   (s == 3'b101) ? 8'h20 :
   (s == 3'b110) ? 8'h40 :
   (s == 3'b111) ? 8'h80 :
                   8'bx;


endmodule
```

Continuous Assignment

```
module decoder (s, y);
input  [2:0] s;
output [7:0] y;
reg    [7:0] y;

always @(s)
    case (s)
        3'b000: y = 8'h01;
        3'b001: y = 8'h02;
        3'b010: y = 8'h04;
        3'b011: y = 8'h08;
        3'b100: y = 8'h10;
        3'b101: y = 8'h20;
        3'b110: y = 8'h40;
        3'b111: y = 8'h80;
       default: y = 8'hx;
    endcase
endmodule
```

Procedural (always block)

Defining Sequential Circuits using Verilog

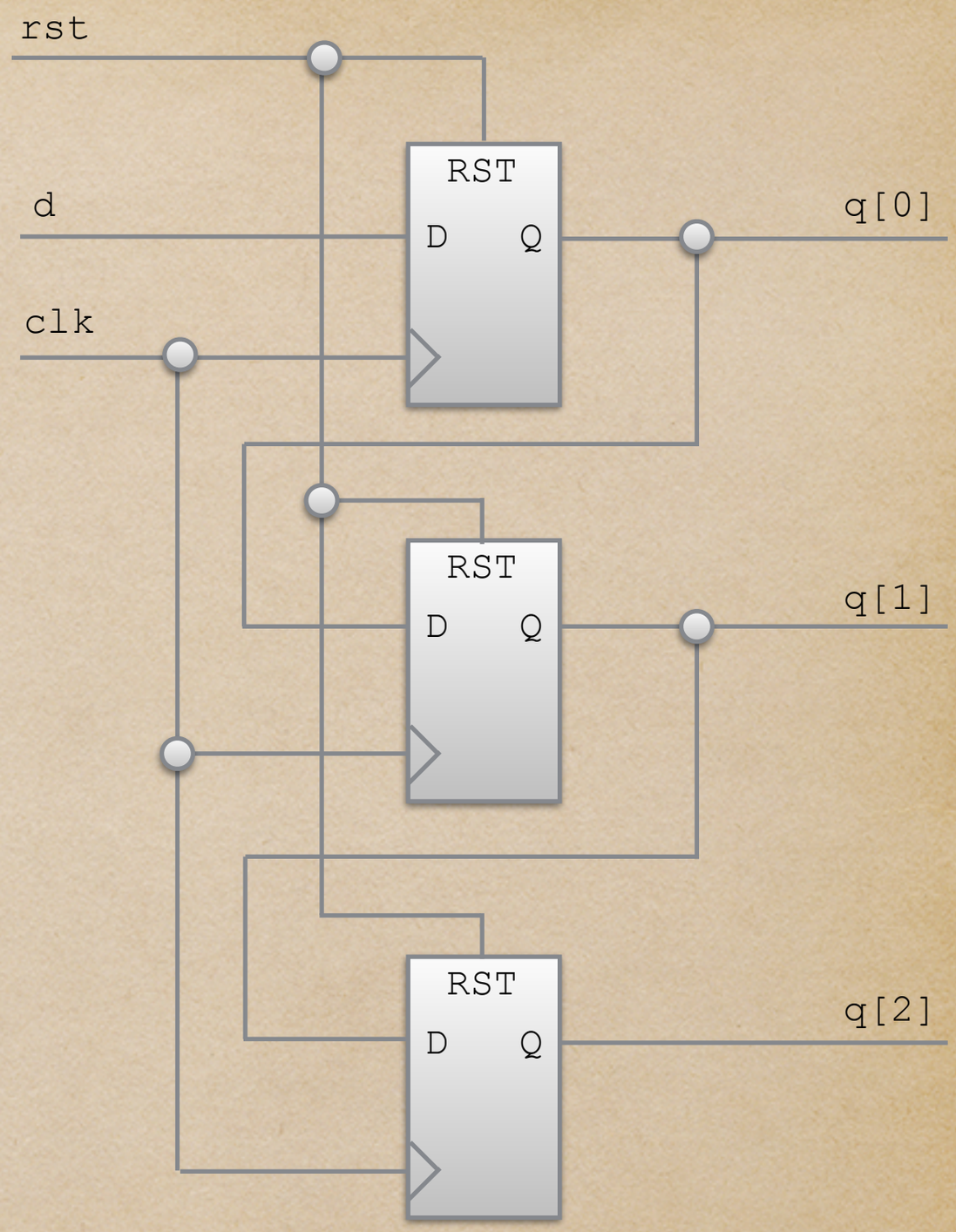
```
module flop (clk, rst, d, q);
input   clk;
input   rst;
input   d;

reg     q;

always @(posedge clk or posedge rst)
   if (rst) q <= 1'b0;
   else     q <= d;

endmodule
```

```
module flop (input wire clk,
             input wire rst,
             input wire d,
             output reg q);

always @(posedge clk, posedge rst)
   if (rst) q <= 1'b0;
   else     q <= d;

endmodule
```

Verilog 2001 Syntax

Note that the Verilog 2001 syntax has a comma separator in the always instead of the keyword “or”

```
module shift3 (clk, rst, d, q);

input         clk, rst, d;
output [2:0] q;

always @(posedge clk, posed rst)
   if (rst)
      q[2:0] <= 3'b0;
   else
      q[2:0] <= {q[1],q[0],d};

endmodule
```



Eight-bit Up Counter

```
module count8 (clk, rst, load, d, q);
input     clk, rst, load;
input  [7:0] d;
output [7:0] q;
reg    [7:0] q;

always @(posedge clk, posedge rst)
   if   (rst)  q <= 8'b0; else
   if   (load) q <= d;
   else        q <= q + 8'b1;

endmodule
```

What is the difference between synchronous inputs and asynchronous inputs and the way they are modeled?

Verilog Constructs

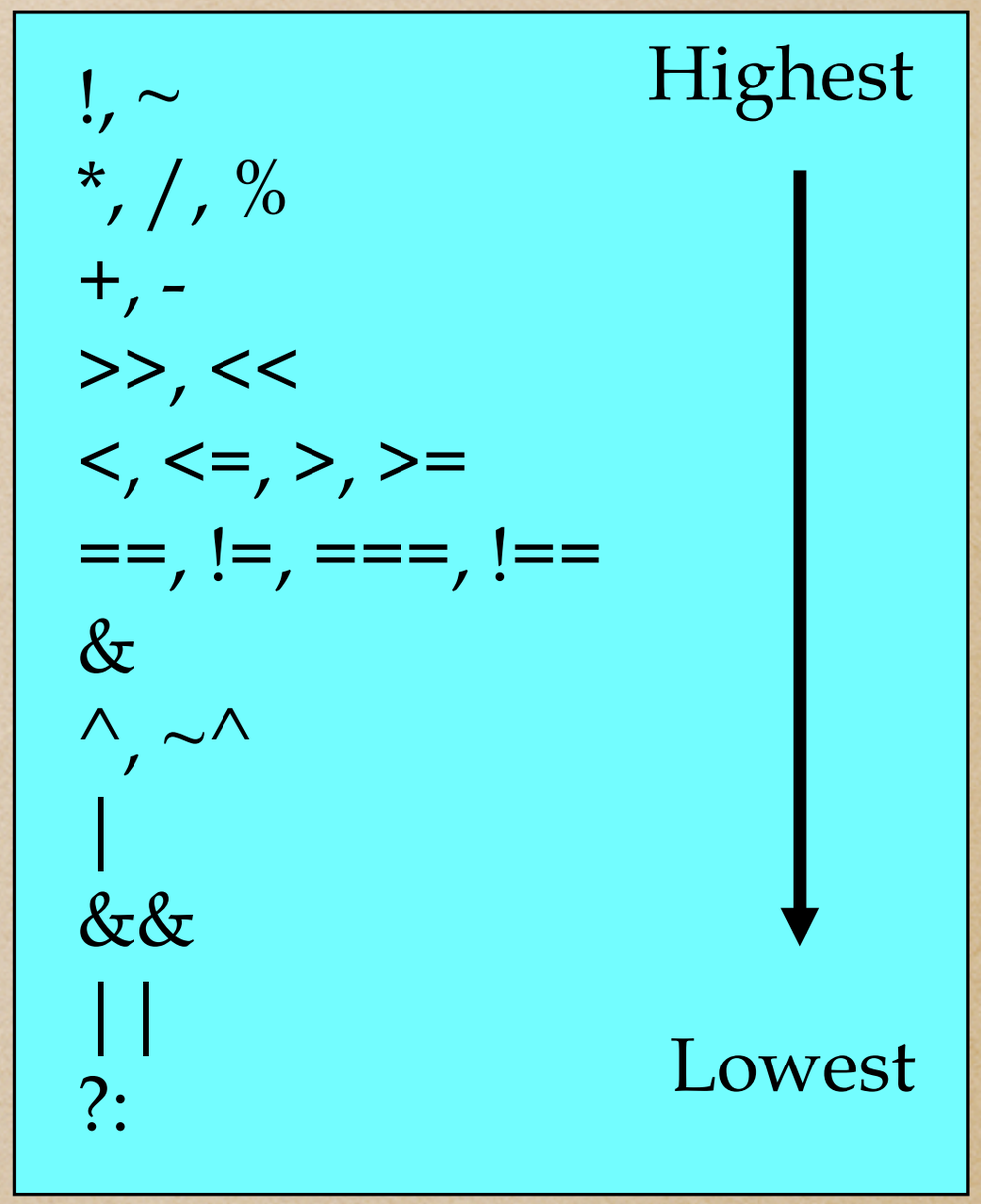
- Verilog is case sensitive
- Keywords should be lower case
- Identifiers are module names, variable names, instance names, etc
  - All identifiers begin with {a..z, A..Z, "_"} and may contain numerics {0 ..9}, alphabetics, and the underscore "_"
  - Identifiers may be up to 1023 characters long
- Verilog is a "free format" language
  - Format code to enhance readability
  - Try to show logical flow by indention
  - White space {tab, space, newlines} is ignored (try to avoid the use of tabs for formatting issues)

Verilog Constructs

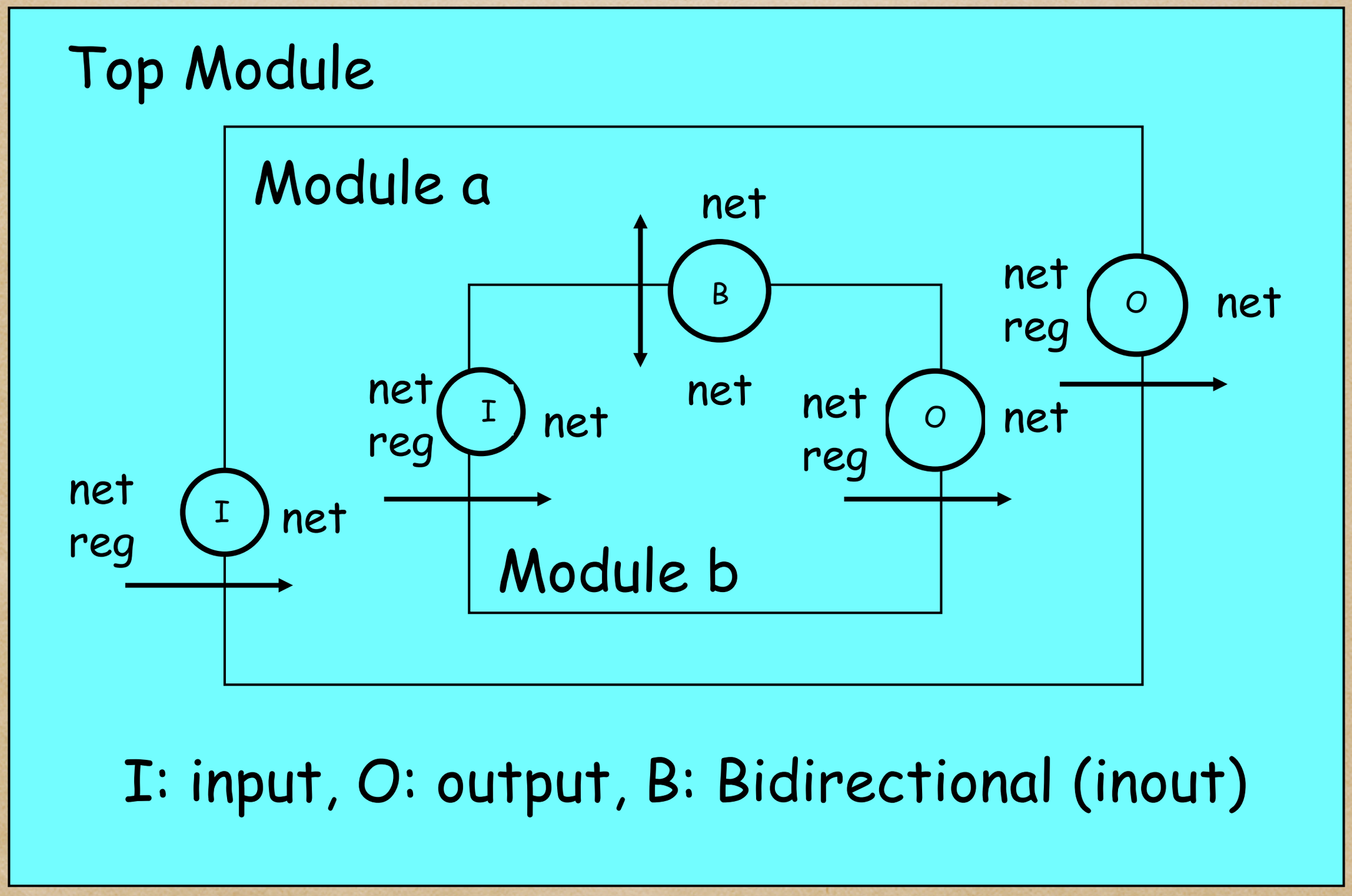
- Parameters may be used as constants
  - May be used to produce scalable modules (i.e. scaling data paths per instantiation)
  - Parameters are defined in the header portion of the module prior to use
  - Parameters are also useful in defining the states in a state machine
- "**$display**" system task used to display the value of signals within the design
- "C" like print formatters may be used which will take precedence over default base
  - {%h,%H,%d,%D,%o,%O,%b,%B,%c,%C,%s,%S,%t,%T}

Verilog Operators and their Hierarchy

{}	Concatenation
+, -, *, /	Arithmetic
%	Modulus
~	bit-wise Negation
&	bit-wise/reduction AND
|	bit-wise/reduction OR
^	bit-wise/reduction XOR
^~, ~^	bit-wise/reduction XNOR
<<	Left shift
>>	Right shift
?:	Conditional
>, >=, <, <=	Relational
!	Logical Negation
&&	Logical AND
||	Logical OR
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality



Module Interfacing Guidelines



I: input, O: output, B: Bidirectional (inout)

Continuous Assignments

- Combinational circuits may be modeled using continuous assignments
- Key features of continuous assignments
  - Not contained within procedural block (always/initial)
  - LHS is always a net data type
  - LHS variable must be declared prior to the assignment
  - Declaration may be combined with the assignment
  - Remember: continuous assignments establish relationships

```
wire y;                 //declare variable
assign y = a|b;         //establish the relationship
wire x = c&d;           //combined declaration/assignment

//complex relationships are easily defined
wire bigger;
wire [31:0] vector1, vector2;
assign bigger = (vector1 > vector2);
```

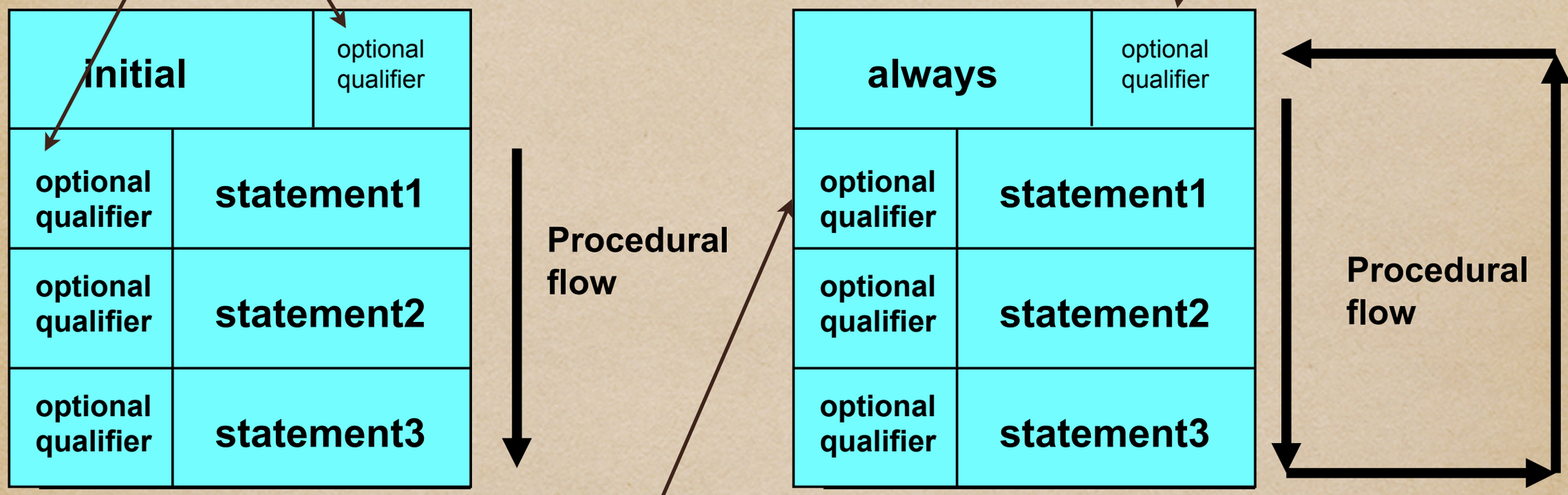
Modeling with Procedural Blocks

- Both combinational and sequential circuits may be modeled with procedural blocks (**always** block)
- Key features of modeling with procedural blocks
  - Always contained within a procedural block (always, initial, task or function)
    - LHS **must be** a register data type and must be declared (reg, integer, time or real)
    - Remember: behavioral constructs have the concept of a procedural flow
    - initial block used only in test benches
    - always block used for design (and test benches)

Behavior of Procedural Blocks

These break up the flow of the test bench to allow you to observe the results of your stimulus

Typically @(posedge clock) or @(*)



initial blocks are ONLY used in test benches, we never model circuits using them

always blocks are used to model circuits (AND they may be used in test benches)

Typically not used

Procedural Qualifiers and Assignments

- Block qualifiers control when a procedural block is activated
- Statement qualifiers control when a statement is executed
  - Time delay
    - Delays are specified by #<number> with the units being defined by the currently active `timescale
      - `timescale 1ns/10ps // first # is unit, second # is precision
      - Include a `timescale at the beginning of each file (outside module)
  - Edge sensitive
    - @(m) // wait till m changes (i.e. any transition of m)
    - @(posedge clk) // wait till clock goes 0 --> 1
    - @(m or n) // **now @(*)**
    - @(posedge clk or negedge rst) // clock plus asynchronous reset
  - Level sensitive
    - wait (signal) - Use in test benches only
- LHS is always a register data type
- Failure to declare LHS as a register will result in:
  - “Illegal LHS assignment” error message

Procedural Assignments

- "Blocking" (=) procedural assignments "block" the procedural flow until the assignment has been made

Use "blocking" procedural assignment for Combinational Blocks
"="

```
reg [15:0] yo, ya, ye;
always @(a or b)
        begin
        #5  yo = a | b;
        #50 ya = a & b;
        #75 ye = a ^ b;
        end
```

When a or b switches then wait 5 time units and assign yo, then wait 50 time units and assign ya, and then wait 75 time units and assign ye. If a or b changes in the middle the next assignment will use a different value

- "Non-blocking" (<=) procedural assignments allow assignments to be made without interrupting the flow

Use "non-blocking" procedural assignment for Sequential Blocks
"<="

```
reg [15:0] ys, yp, yq;
always @(posedge clk)
        begin
        ys <= a + b;
        yp <= a * b;
        yq <= a / b;
        end
```

When the active edge of the clock occurs (^) then the right hand side of each expression is evaluated and then all of the left hand sides are updated. Consider this like temporary variables holding intermediate results

Overview of Flow Control

- if - else
- while loop
- for loop
- case, casex, casez
- repeat *
- forever *

* Test bench use only

if - else Flow Control

- Basic Form - statements may be compound and delimited by begin - end pair

```
if (condition)
        statement1;
else
        statement2;
```

- Nested if/else statements - remember the “else” is always associated with the most recent “if” or you can force the association with begin - end pair

```
if (condition1)
        begin
        statement1T;
        •
        •
        •
        if (condition2)
                statement2T;
        end
else
        if (condition3)
                statement1F3T;
        else
                statement1F3F;
```

While Loop Control

- Simple form

```
while (condition)
      statement;
```

Be careful not to define an endless while loop with no elapsed time. Simulation will hang.

- Example

```
count = 0;
while (sum < 256)
      begin
      sum = sum + inc;
      count = count + 1;
      end
```

Remember that the register sum must be at least 9 bits in order to exceed 256. If it had been defined as 8 bits then the values would range between 0 and 255 and the expression would never evaluate FALSE (terminating condition)

for loop / repeat / forever

- Similar syntax to “C”
- Loop variable must be appropriately declared

```
for (i=0; i < 16; i = i + 1)
        rega[i] = rega[i] & en;
```

Make sure the loop variable “i” is defined to be large enough. If i is defined as reg [2:0] then it would never reach 16 and the loop would never terminate.

- Repeat and forever are for test benches or interactive debug

```
repeat (condition)
        statement;


forever @(posedge clk)
        $display(“register a = %h”,rega);


repeat (5)
        @(posedge clk) ;
```

The last example will cause the simulator to wait here until 5 clocks occur - and will then continue. Notice that after the qualifier @(posedge clk) that there is just a semicolon - this is called a NULL STATEMENT (do nothing)

Case Statements

- There are three forms of the case statement in Verilog. In our case we will use either the case or the casez. We should not need to use a casex

- **case**: (0:0, 1:1) // x and z are “no match”)
- **casex**:(0:0, 1:1, x,z:{0,1,x,z}, ?:{0,1,x,z}) // a “?” maps to a “z”
- **casez**:(0:0, 1:1, z:{0,1,x,z}, ?:{0,1,x,z}) // x is a “no match”
- Table options: {0, 1, x, z, ?}
- Case variable options: {0, 1, x, z}

```
reg [15:0] rega;
reg [9:0] result;
        •
        •
        •
case (rega)
        16’d0:   result = 10’b1010101010;
        16’d1:   result = 10’b1100110011;
        16’d2:   result = 10’b1110111011;
        16’d3:   result = 10’b1111011110;
        16’d4:   result = 10’b1111101111;
        16’d5:   result = 10’b1111110111;
        16’d6:   result = 10’b1111111011;
        16’d7:   result = 10’b1111111101;
        16’d8:   result = 10’b1111111110;
        16’d9:   result = 10’b1111111111;
        default result = ‘bx;
endcase
```

Casez Example

- The casez allows the use of “don’t cares” which greatly reduce the number of entries required in the case table
- Benefit of the casez is the use of the “?” in the case table

```
reg [7:0] rega;
reg [7:0] result;
        •
        •
        •
casez (rega)
        8’b1???_????:  result = 128; // here we ask if msb set
        8’b01??_????:  result =  64; // if not then we check next
        8’b001?_????:  result =  32;
        8’b0001_????:  result =  16;
        8’b0000_1???:  result =   8;
        8’b0000_01??:  result =   4;
        8’b0000_001?:  result =   2; // and so on
        8’b0000_0001:  result =   1; // until we check the lsb
            default:   result =   0; // if none set assign zero
endcase
```

Memories in Verilog

- A memory may be modeled in Verilog using procedural blocks
- A memory is defined as a two-dimensional array of registers
  - reg [15:0] mymem [0:1023];//1k x 16
    - [15:0] defines width and bit orientation
    - [0:1023] defines range of addresses
    - First address range value is the first address (applicable to system tasks that fill memory)
- Memory contents may be accessed on a word basis only (no doubly subscripted memory references)
- Contents must be copied to a register before bit access

```
reg [15:0] mymem [0:1023];
reg [15:0] word;
reg        thebit;
             .
             .
             .
word = mymem[100];
thebit = word[12];
```

```
// reading memory

always @(*)
   word = mymem[address];   // where address is 0 .. 1023


// writing memory

always @(posedge clk)
   if (MemWrite) mymem[address] <= detain;
```

Initializing Memories

- Power on

```
initial
      for (i=0; i<1024; i=i+1)
            mymem[i] = 16'b0;          //fill w/ 0's
```

- Load contents from a text file

```
$readmemh("memfileh", mymem); //hex-ASCII file
            -or-
    $readmemb("memfileb", mymem); //bin-ASCII file
```

- Memory content files are always text files (ASCII)
- Memory contents
  - Hex: $readmemh
  - Binary: $readmemb
- Address references are always Hex
- Loading starts at the first memory address and proceeds until "@hexad" encountered
  - @1A // jump load pointer to address 1A

Sample Files

$readmemh

```
1A    2C    01
32    15    29
@10
14    17    26
@120
32    15    79
61    3B    CD
@3FD
FF    FF    FF
```

$readmemb

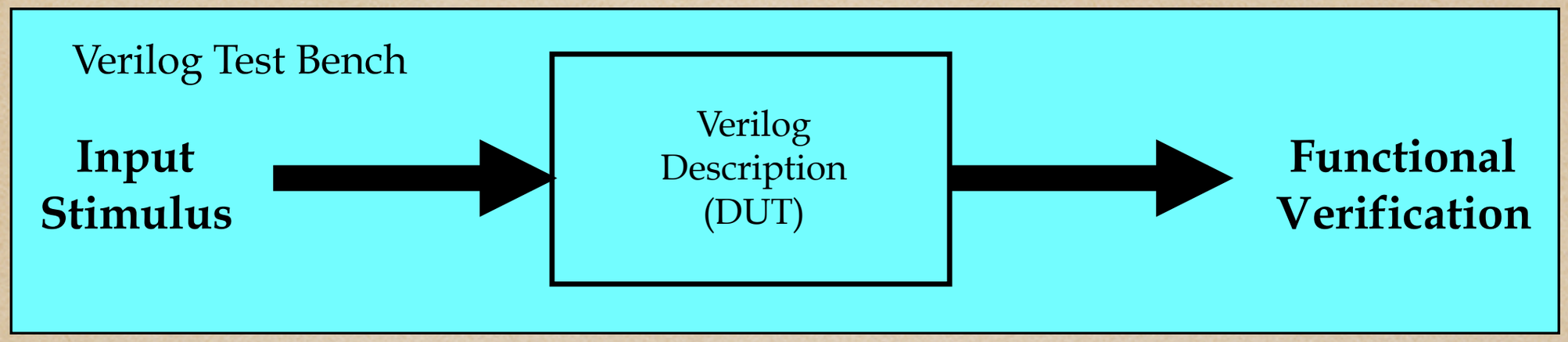
```
0001_1010 0010_1100 0000_0001
0011_0010 0001_0101 0010_1001
@10
0001_0100 0001_0111 0010_0110
@120
0011_0010 0001_0101 0111_1001
0110_0001 0011_1011 1100_1101
@3FD
1111_1111 1111_1111 1111_1111
```

Multidimensional Arrays in Verilog 2001

- Multidimensional arrays may now be declared
  - Any number of dimensions may be declared
  - Each dimension may have the lowest numbered address as first or last
- You may access one word at a specific reference

```
reg [15:0] mymem [0:1023];                  //one-dimensional array
reg [15:0] array_2d [0:4095][0:127];        //two-dimensional array
integer I [15:0][127:0][7:0][3:0];          //four-dimensional array

data = array_2d[n][m];                      //access a single element
```

Verifying Designs using Verilog



- Verilog Test Bench defined as "top level" module
- Define variables required to provide input stimulus to the logic circuit being tested
- Instantiate the logic circuit (DUT) (Design Under Test)
- Apply the input stimulus (assignments in test bench)
- Monitor functionality of DUT to confirm correctness (text or waveforms)
- Inputs to logic circuit typically defined as register data types
  - Similar in concept to a variable in a HLL
  - Value assigned and held until changed
- Outputs of DUT defined (at test bench level) as wire data types
- Assigning different values to the test bench variables will cause the inputs to the DUT to switch and allow verification of functionality

Running a Simulation

- Create a new project with the Xilinx ISE
- Create the Verilog source files using an editor and save them in files with the format “file.v”
- Create the Verilog test bench with the top level circuit module (of your design) instantiated
    - Save the test bench with the file name “file_tb.v”
- Add the files to the project by right clicking the project name and select “add source”
- Compile the source code and initiate the simulation
    - Select “Behaviorial Simulation”
    - Select “file_tb.v”
    - Double click to run the simulation
- Simulator will execute the simulation until:
    - no more events to process
    - $finish; statement executed (included in test bench)
    - $stop; statement executed (included in test bench)

Sample Test Bench

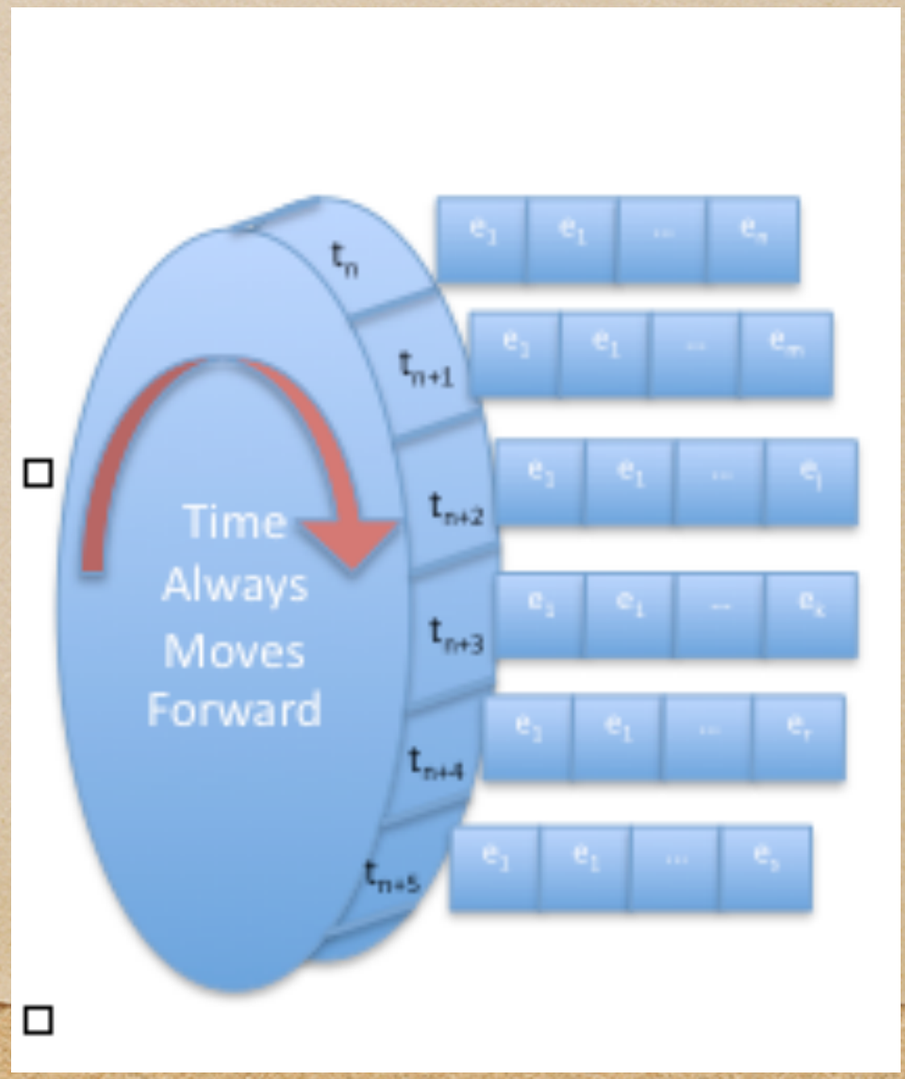
```
module test;

reg in1, in2, in3, in4;

wire out;

aoi a1 (.in1(in1),
        .in2(in2),
        .in3(in3),
        .in4(in4),

        .out(out)
        );

initial begin
   in1 = 0; in2 = 0;
   in3 = 0; in4 = 0;

   #20 in1 = 1; in4 = 1;
   #20 in2 = 1;
   #20 in2 = 0;
   #20 in3 = 1;
   #20 $stop;
   end


initial
   $monitor($time, in1, in2, in3, in4, out);

endmodule
```

Verilog is an Event Driven Simulator

```
`timescale unit/precision          // how simulator interprets time
`timescale 1ns/10ps                // simulator evaluates in 10ps windows
#2.42                              // 2.42 nanoseconds
#5                                 // 5.00 nanoseconds
$timeformat(-9, 2, " ns", 10);     // "-9": Power of 10 to use as time unit
                                   // "2":  Number of digits right of decimal point
                                   // "ns ":String to append to the time displayed
                                   // "10": Number of spaces used
```

t

4.11	e1 e2 e3
4.12	e4 e5
4.13	
4.14	e6 e7
4.15	e9 e10 e11
4.16	e12



Verilog Keywords (don't use)

always	and	assign	begin	buf	bufif0	bufif1
case	casex	casez	cmos	deassign	default	defparam
disable	edge	end	endcase	endfunction	endmodule	endprimitive
endtable	endtask	event	for	force	forever	fork
function	highz0	highz1	if	initial	inout	input
integer	join	large	medium	module	nand	negedge
nmos	nor	not	notif0	notif1	or	output
parameter	pmos	posedge	primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release	repeat	rnmos	rpmos
rtran	rtranif0	rtranif1	scalared	small	strong0	strong1
supply0	supply1	table	task	time	tran	tranif0
tranif1	tri	triand	trior	trireg	tri0	tri1
vectored	wait	wand	weak0	weak1	while	wire
wor	xnor	xor				

VHDL Keywords (don't use)

Reserved words in VHDL

abs	disconnect	is	out	sli
access	downto	label	package	sra
after	else	library	port	srl
alias	elsif	linkage	postponed	subtype
all	end	literal	procedure	then
and	entity	loop	process	to
architecture	exit	map	pure	transport
array	file	mod	range	type
assert	for	nand	record	unaffected
attribute	function	new	register	units
begin	generate	next	reject	until
block	generic	nor	return	use
body	group	not	rol	variable
buffer	guarded	null	ror	wait
bus	if	of	select	when
case	impure	on	severity	while
component	in	open	signal	with
configuration	inertial	or	shared	xnor
constant	inout	others	sla	xor