



TEC de Monterrey®
DEL SISTEMA TECNOLÓGICO DE MONTERREY

INTRODUCTION TO VERILOG HDL

LUIS F. GONZÁLEZ PÉREZ
ITESM-GDA

INTRODUCTION TO VERILOG

VERILOG OVERVIEW

AUG-20

LFGP / TE4003

2

WHAT IS VERILOG HDL?

IEEE industry standard Hardware Description Language (HDL) used to describe a digital system

HDL: A high level programming language used to model hardware

Hardware Description Languages

- Have special hardware related constructs
- Can be used to build digital system models for **simulation, synthesis and test**
- Have been extended to the system design level
- Concurrent hardware description
 - Expresses parallelism in the hardware

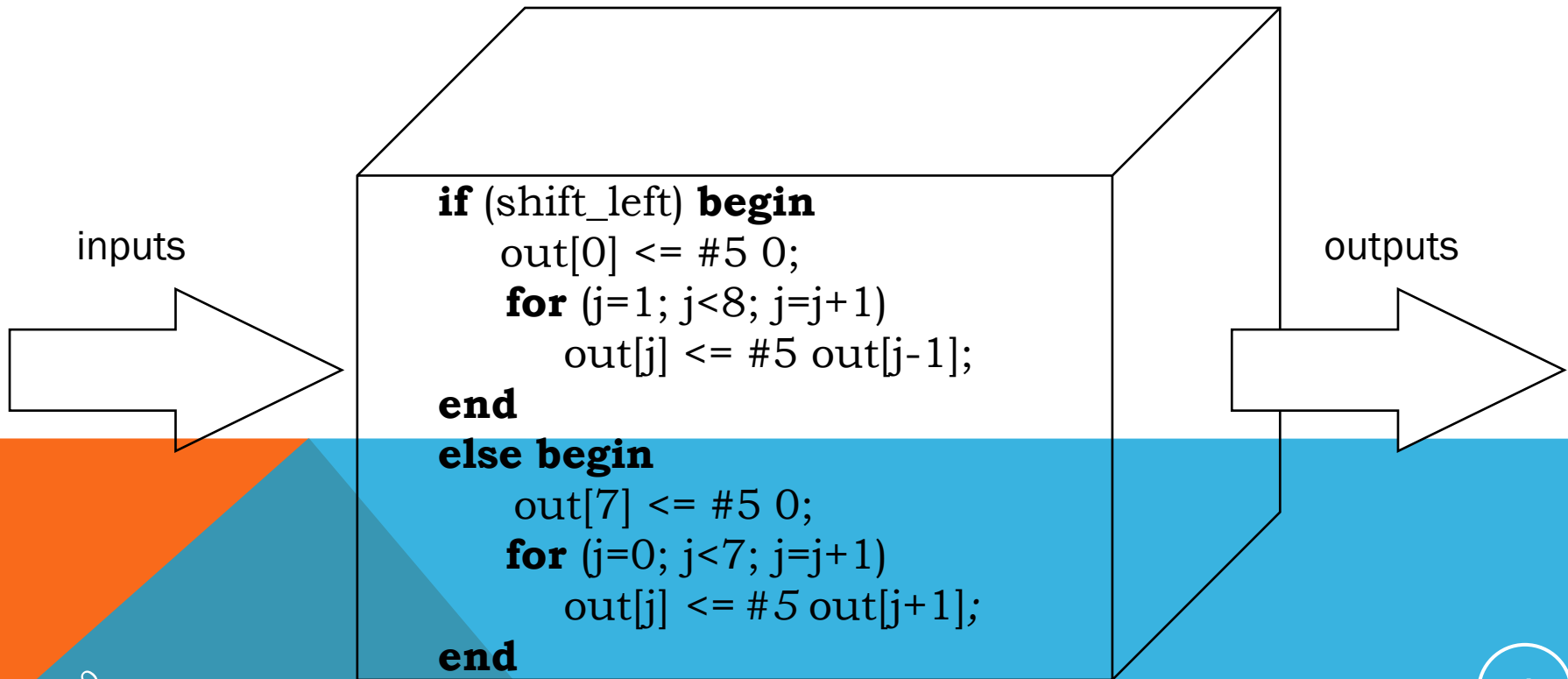
Benefits of using HDLs

- True abstract behavior modeling
- Hardware structure modeling

WHAT IS VERILOG HDL?

Abstract behavior modeling

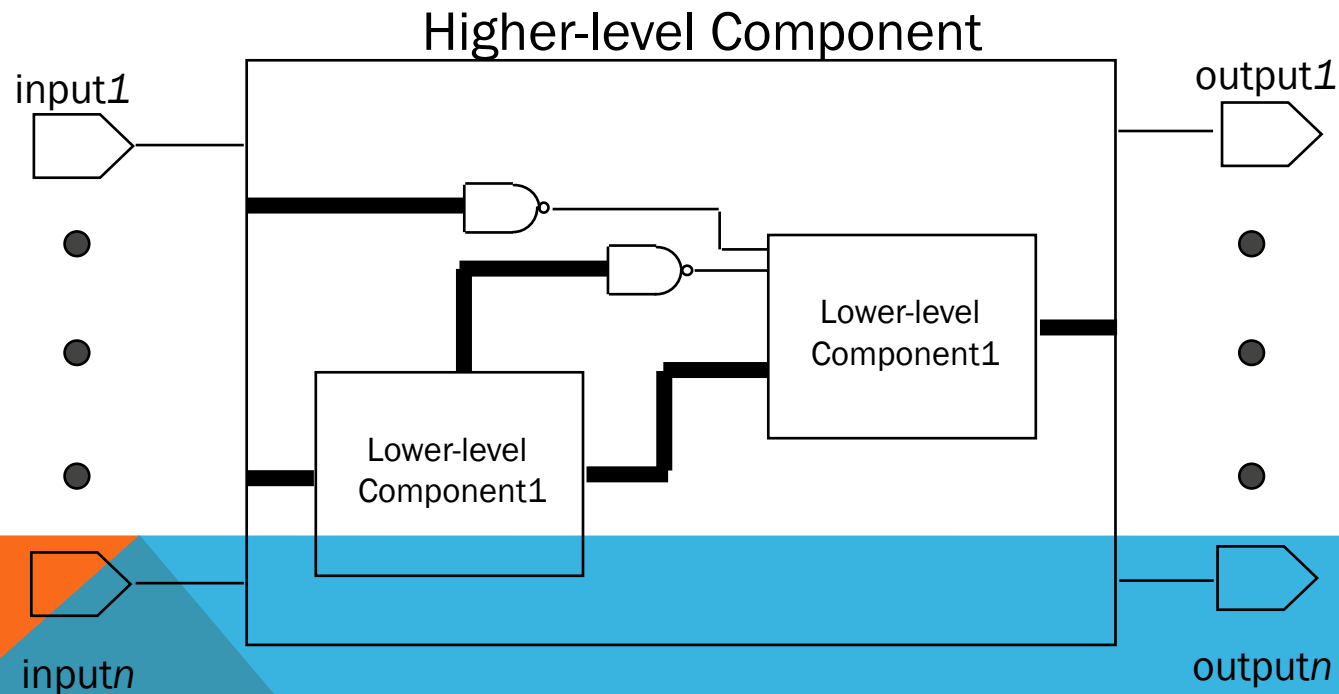
- Declarative statements that describe behavior of hardware
- The component is described by its input/output response
- This behavior is not prejudiced by structural or design aspects of the hardware intent
- Synthesis tools creates correct logic



WHAT IS VERILOG HDL?

Hardware structure modeling

- A component is described by interconnecting lower-level components/primitives
- Components are modeled in more detail – call out the specific hardware
- Functionality and structure of the circuit



WHAT IS VERILOG HDL?

**DO NOT EVER code a HDL program
like a C or any other HLL program**

- Serializes the hardware operations
- Leads to BIG increase in the amount of hardware synthesized

HISTORY OF VERILOG

Introduced in 1983 by Gateway Design Automation

1983 – 1987 Verilog gained a strong foothold among advanced high-end designers

1987

- Synopsys began using Verilog as an input to their synthesis product
- IEEE released VHDL

1989 Cadence purchased Gateway

1990

- Cadence released Verilog to the public and kept Verilog-XL simulator as a separate product
- **Open Verilog International (OVI)** was formed to control the language specifications
- Nearly all ASIC foundries supported Verilog and used Verilog-XL as a “golden” simulator

1993

- 85% of submitted designs to ASIC foundries were designed with Verilog
- OVI released version 2.0

1995 IEEE accepted OVI Verilog as a standard, IEEE standard 1364

2001 IEEE revised standard

2005 IEEE accepted new revision for the standard

VERILOG VS OTHER HDL

Verilog

- “Tell me how your circuit should behave, and I will give you the hardware that does the job

VHDL

- Similar to Verilog

PALASM, ABEL, AHDL

- “Tell me what hardware you want, and I will give it to you”

HDL TERMINOLOGY

Register Transfer Level (RTL): A type of behavioral modeling, for the purpose of synthesis

- Hardware is implied or inferred
- Synthesizable

Synthesis: Translating HDL to a circuit and then optimizing the represented circuit

RTL Synthesis: Translating a RTL model of hardware into an optimized technology specific gate level implementation

RTL SYNTHESIS

```
always @(a, b, c, d, sel)
```

```
  case (sel)
```

```
    2'b00: mux_out = a;
```

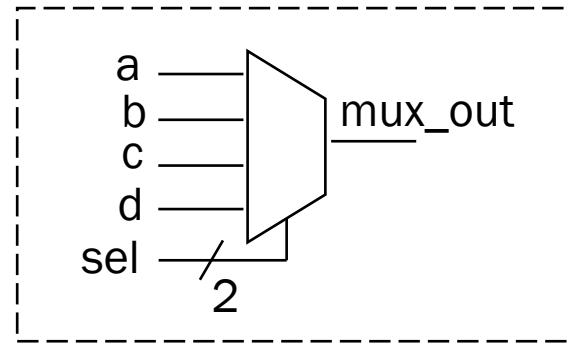
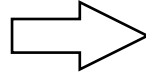
```
    2'b01: mux_out = b;
```

```
    2'b10: mux_out = c;
```

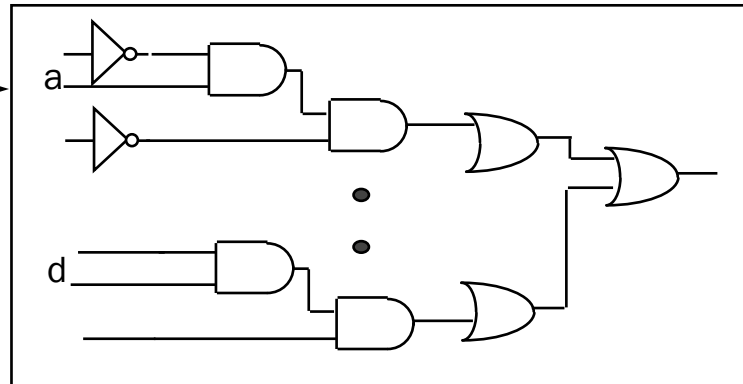
```
    2'b11: mux_out = d;
```

```
  endcase
```

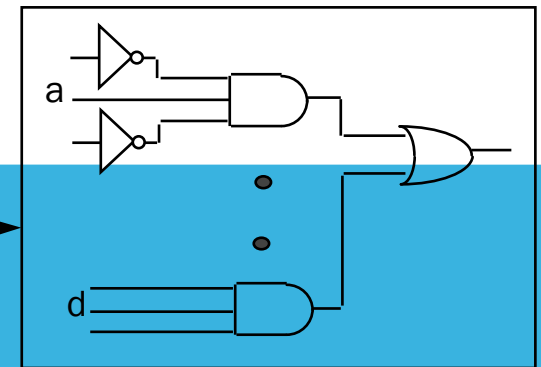
inferred



Translation



Optimization



SIMULATION VS SYNTHESIS

The Verilog language has two sets of constructs

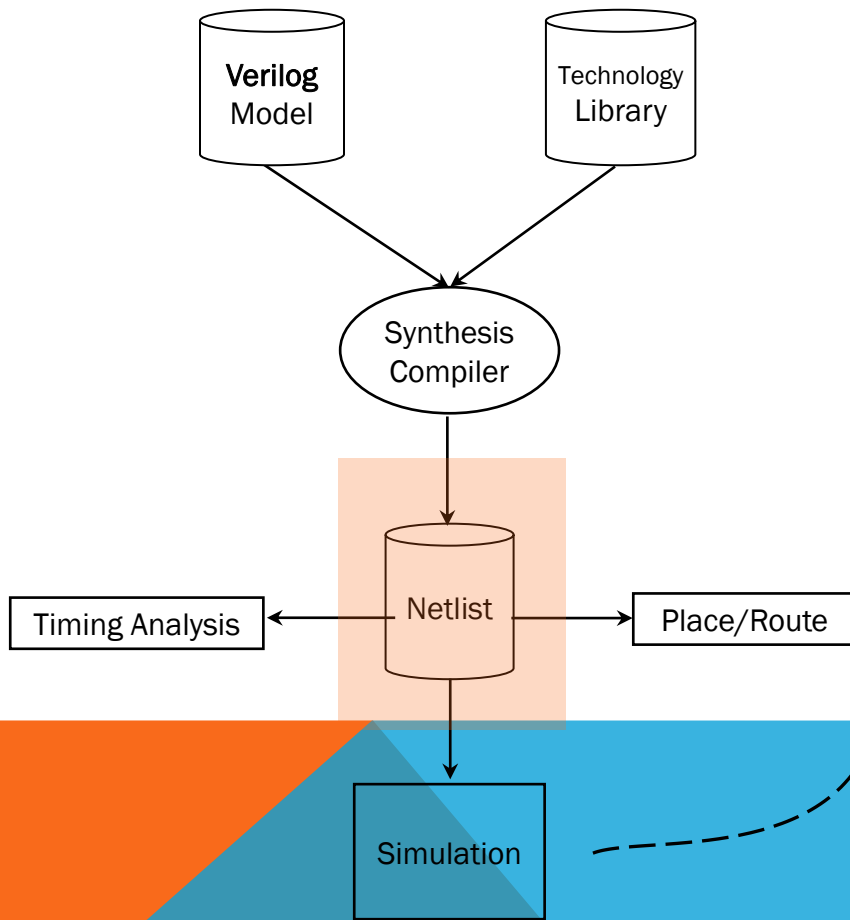
- Simulation
- Synthesis

Most (**but not all**) simulation constructs are synthesizable

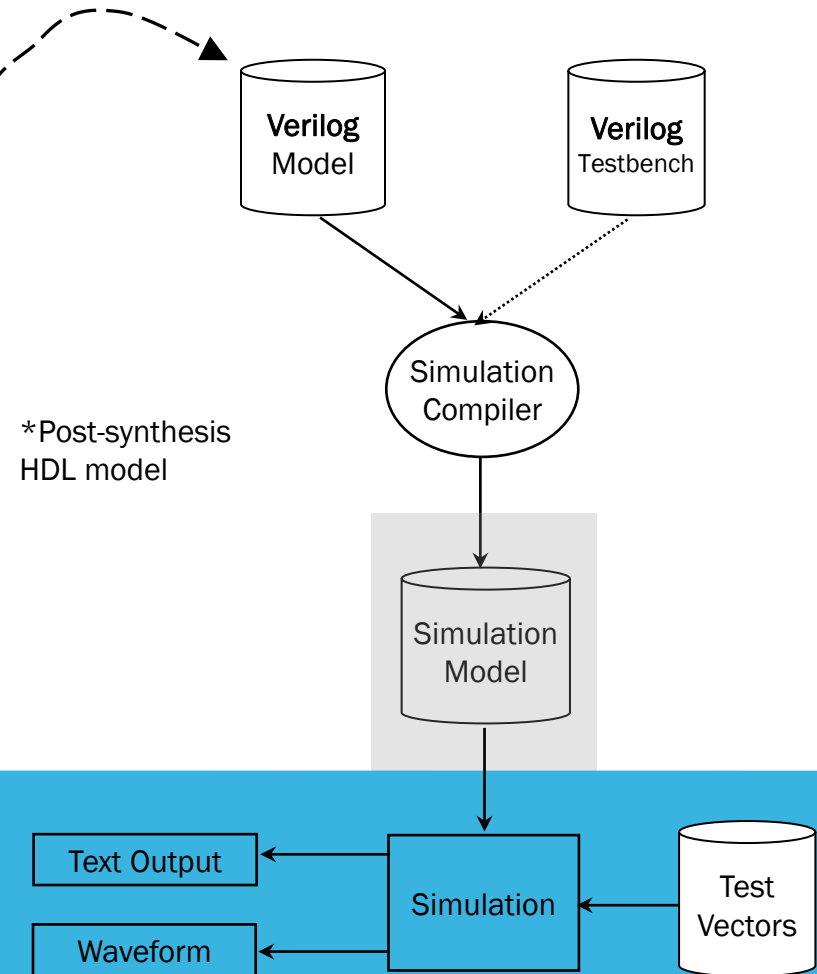
Check help or documentation for your synthesis tool to be sure of the supported constructs

TYPICAL RTL SYNTHESIS & RTL SIMULATION FLOWS

Synthesis



Simulation



*Post-synthesis
HDL model

INTRODUCTION TO VERILOG

VERILOG MODELING

AUG-20

BASIC MODELING STRUCTURE

module *module_name* (*port_list*);

port declarations

data type declarations

circuit functionality

timing specifications

endmodule

Begins with keyword **module** &
ends with keyword **endmodule**

Case-sensitive

All keywords are lowercase

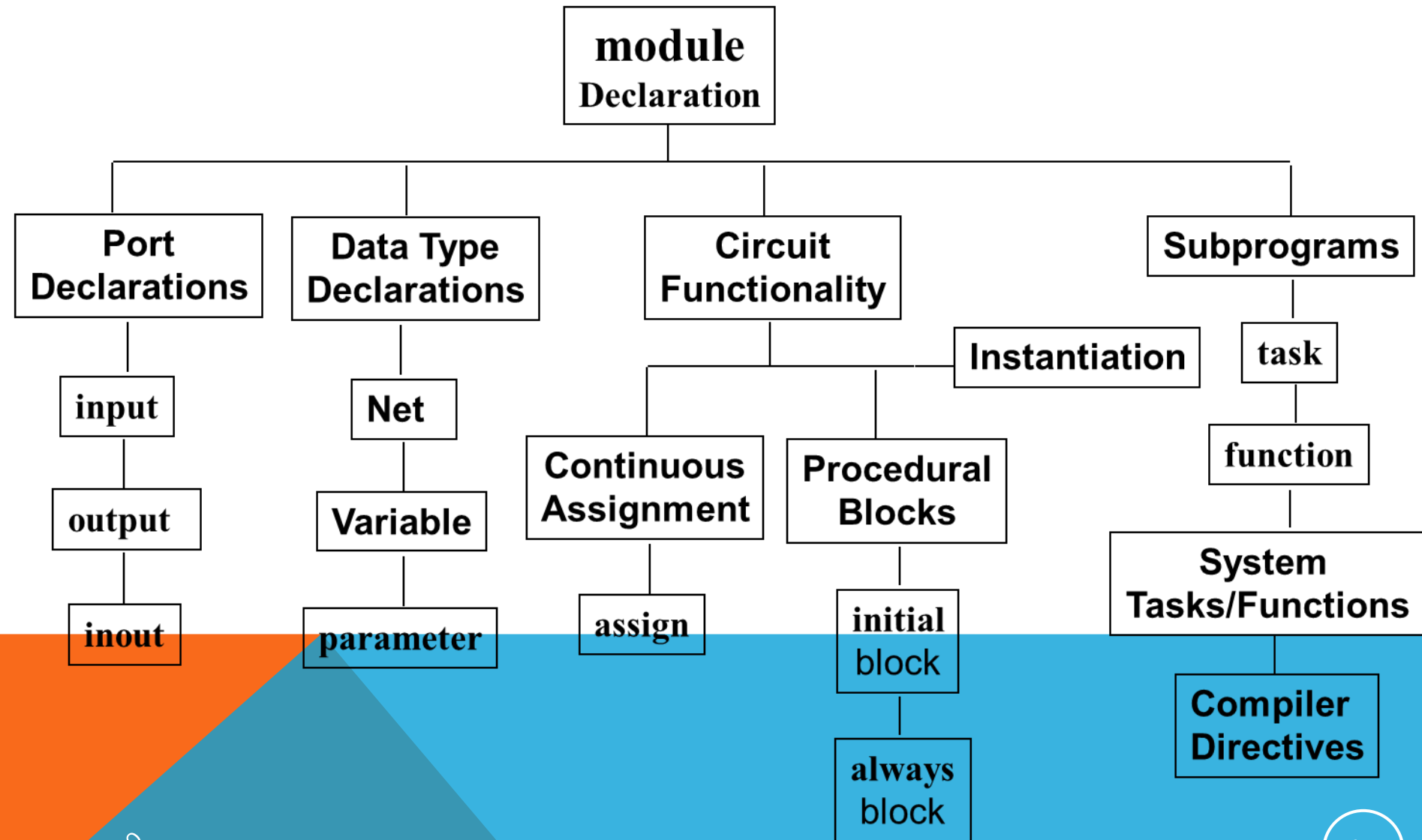
Semicolon is the statement
terminator

// is a single line comment

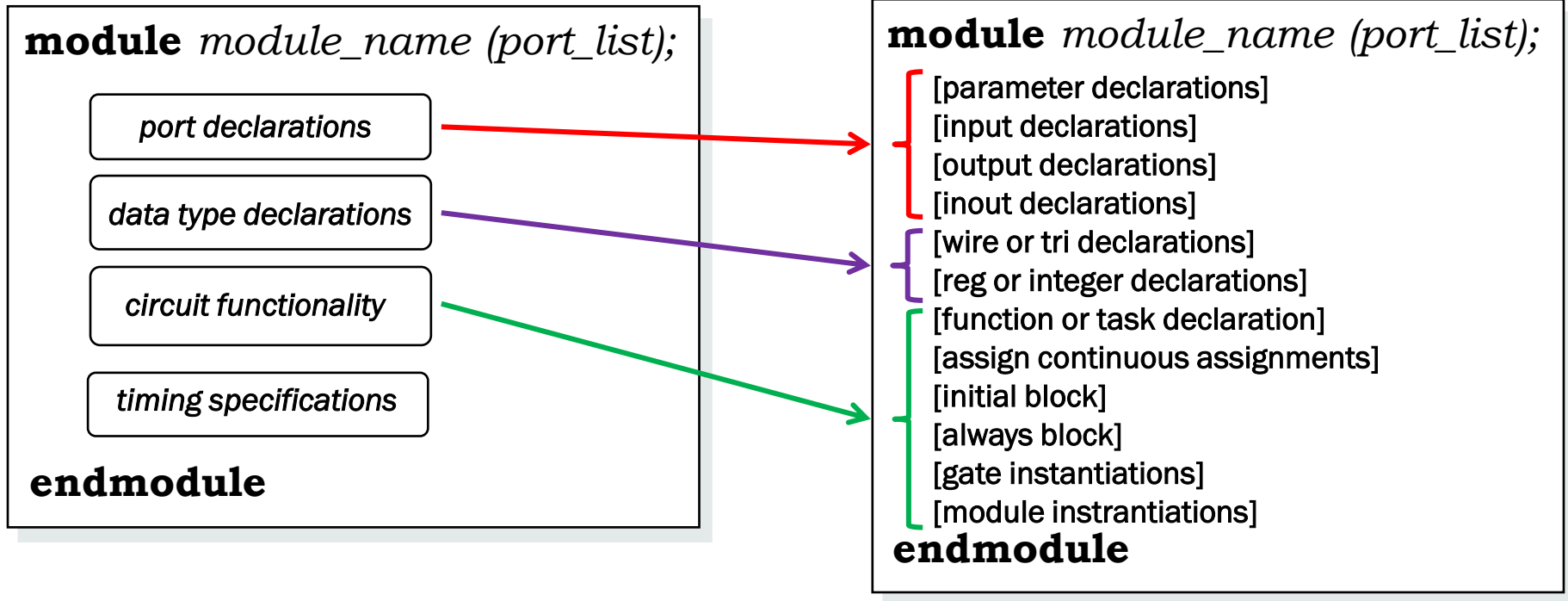
/* */ is a multi-line comment

Timing specifications is for
simulation only

COMPONENTS OF A VERILOG MODULE

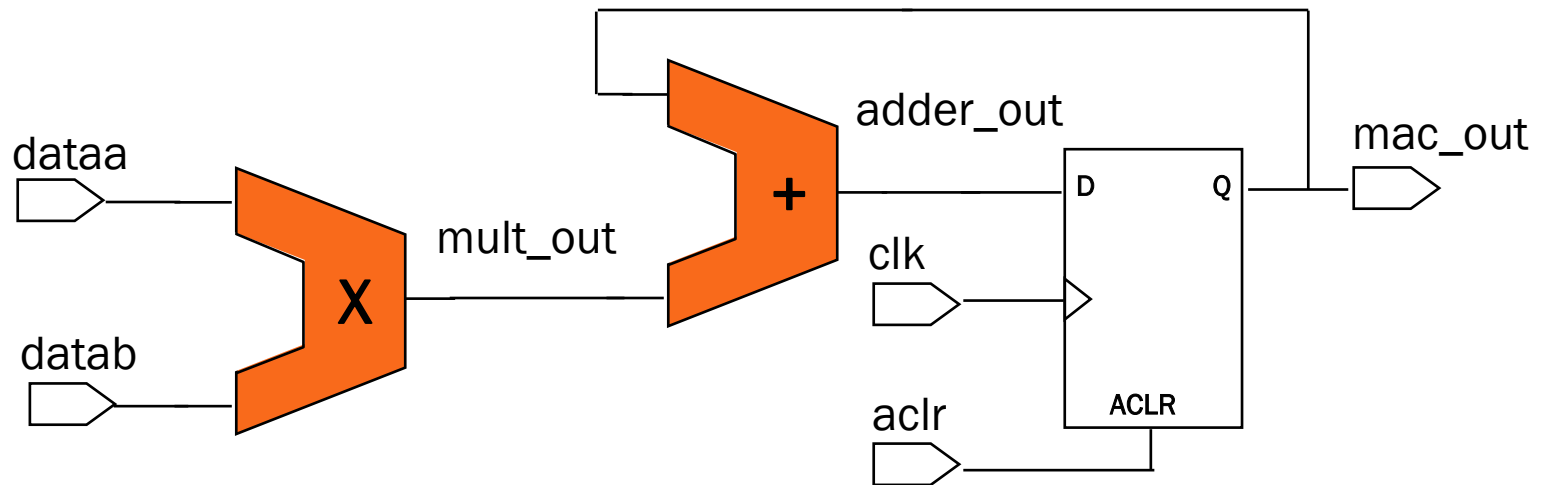


A MORE DETAILED MODELING STRUCTURE



A SIMPLE EXAMPLE - MAC

Schematic representation



A SIMPLE EXAMPLE - MAC

Verilog model

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

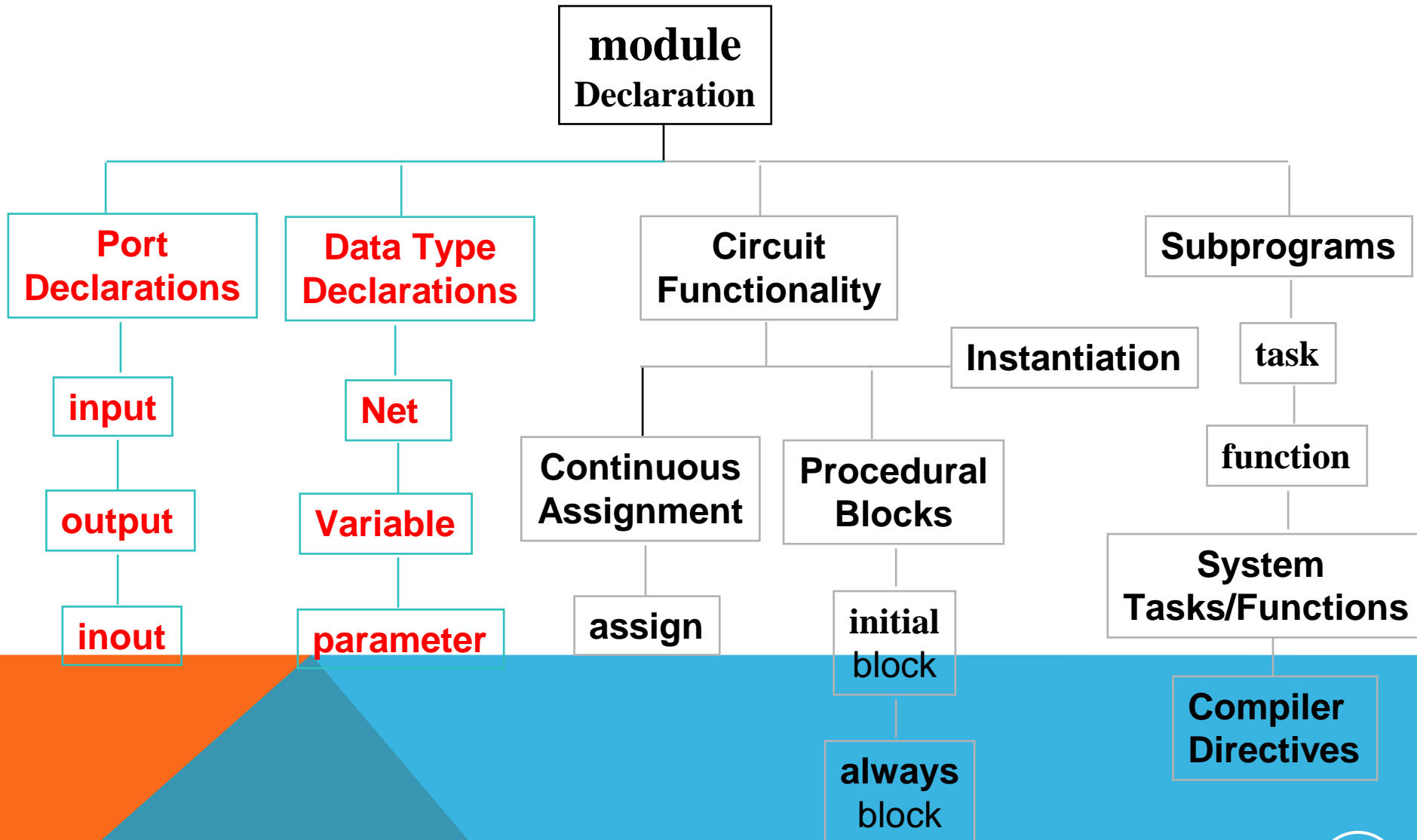
    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    mult_a #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab), .mult_out(mult_out));

endmodule
```

LET'S TAKE A LOOK AT



MODULE DECLARATION

Begins with keyword **module**

Provides the Verilog block (module) name

Includes port list, if any

- A listing of all module I/O names

```
module mult_acc (mac_out, dataa, datab, clk, aclr);
```

PORT DECLARATION

Defines the names, sizes, types & directions for all ports

Format

```
<port_type> port_name;
```

Example

```
input [7:0] dataa, datab;  
input clk, aclr;  
output [15:0] mac_out;
```

Port types

- **input** \Rightarrow input port
- **output** \Rightarrow output port
- **inout** \Rightarrow bidirectional port

VERILOG '2001 AND LATER MODULE/PORT DECLARATION

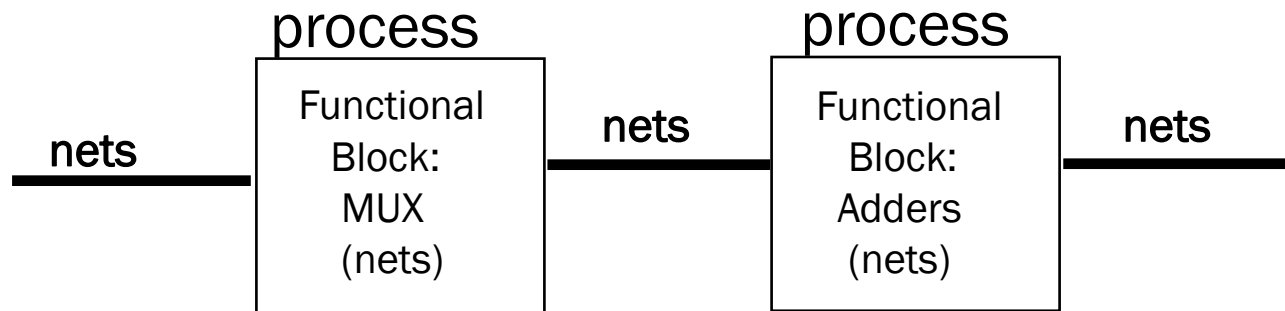
Beginning in Verilog '2001, module and port declarations can be combined

- More concise declaration section
- Parameters (shown later) may also be included

```
module mult_acc (  
    input [7:0] dataa, datab,  
    input clk, aclr,  
    output [15:0] mac_out  
);
```

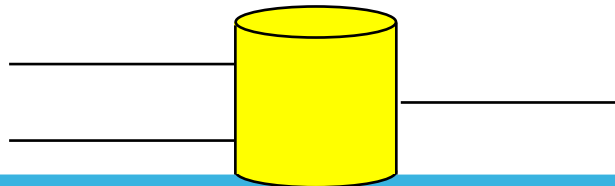
DATA TYPES

Net data type - represents physical interconnect between structures (activity flows)



Variable data type - represents element to store data temporarily

Temporary Storage



NET DATA TYPE & NET ARRAYS

wire \Rightarrow represents a node

tri \Rightarrow represents a tri-state node

Bus Declarations:

- **<data_type>** [*MSB* : *LSB*] *<signal name>* ;
- **<data_type>** [*LSB* : *MSB*] *<signal name>* ;

Examples:

- **wire** *<signal name>* ;
- **wire** [15:0] mult_out, adder_out;

NET DATA TYPES

Net Data Types	Functionality	Synthesis Support?
wire	Used for interconnect	Y
tri		Y
supply0	Represents constant value (i.e. power supply)	Y
supply1		Y
wand	Represents wired logic	Y
triand		Y
wor		Y
trior		Y
tri0	Tri-state node with pull-up/pull-down	Y
tri1		Y
triereg	Stores last value when not driven	N

Note: There is no functional difference between wire & tri; wand & triand; wor & trior

VARIABLE DATA TYPES

reg⁽¹⁾ - unsigned variable of any bit size

- Use **reg signed** for a signed implementation⁽²⁾

integer - signed variable (usually 32 bits)

Bus Declarations:

- **<data_type>** [*MSB* : *LSB*] *<signal name>* ;
- **<data_type>** [*LSB* : *MSB*] *<signal name>* ;

Examples:

- **reg** *<signal name>* ;
- **reg** [7 : 0] *out* ;

Notes:

- 1) Type **reg** does not refer to a physical register
- 2) The **signed** representation is also supported on net data types

VARIABLE DATA TYPES

Variable Data Types	Functionality	Synthesis Support?
reg	Unsigned variable (by default) Use reg signed for signed representation	Y
integer	Signed variable (usually 32 bits)	Y
time	Unsigned integers (usually 64 bits) used for storing and manipulating simulation time	N
real	Double precision floating point variable	N
realtime	Double-precision floating point variable used with time	N

MEMORY

Multi-dimensional variable array

- Cannot be a net type

Examples:

```
reg [31:0] mem[0:1023]; // 1Kx32
reg [31:0] instr;

...
instr = mem[2];
mem [1000][5:0] = instr[5:0]; // Unsupported by synthesis
```

Cannot write to multiple elements in one assignment

```
mem = 32'd0;     Illegal!!!
```

PARAMETER

Value assigned to a symbolic name

Must resolve to a constant at compile time

Can be overwritten at compile time

- Exception: local parameters (**localparam**)
- Discussed later

```
parameter size = 8;
```

```
reg [size-1:0] dataa, datab;
```

VERILOG '2001 & LATER

MODULE/PORT/PARAMETER DECLARATION

Module, port and parameter declarations can be combined

- Illegal for local parameters

```
module mult_acc
  #(parameter size = 8)
  (
    input [size-1:0] dataa, datab,
    input clk, clr,
    output [(size*2)-1:0] mac_out
  );
```

DATA TYPES

Every signal (which includes ports) must have an assigned data type

Data types for signals must be explicitly declared in the declarations of your module

Ports are wire (net) data types by default if not explicitly declared

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    mult_a #(.width_in(mult_size))
    u1 (.in_a(dataa), .in_b(datab), .mult_out(mult_out));

endmodule
```

NET vs REG

Net

- Represents and model the physical connection of signals
- Net objects must always be assigned using a *continuous assignment* statement

Register

- Holds its value over simulation from one *procedural assignment* statement to the next and *means it holds its value over simulation delta cycles*.
- It does not imply that a physical register will be synthesized, although it is used for this purpose
- It is used to assign values under *trigger conditions* such as **if** and **case** statements
- A procedural assignment stores a value in a register data type and is held until the next procedural assignment to that register data type

INTRODUCTION TO VERILOG

ASSIGNING VALUES - NUMBERS & OPERATORS

AUG-20

ASSIGNING VALUES – NUMBERS

Are sized or unsized: **<size>'<base format><number>**

- **Sized** example: **3'b010** = 3-bit wide binary number
 - The prefix (3) indicates the size of number
- **Unsized** example: **123** = 32-bit wide decimal number by default
 - **Defaults**
 - No specified <base format> defaults to **decimal**
 - No specified <size> defaults to **32-bit** wide number

Base Formats

- Decimal ('d or 'D) **16'd255** = 16-bit wide decimal number
- Hexadecimal ('h or 'H) **8'h9a** = 8-bit wide hexadecimal number
- Binary ('b or 'B) **'b1010** = 32-bit wide binary number
- Octal ('o or 'O) **'o21** = 32-bit wide octal number
- Signed ('s or 'S) **16'shFA** = signed 16-bit hex value

NUMBERS

Negative numbers - specified by putting a minus sign before the <size>

- Legal: `-8'd3` = 8-bit negative number stored as 2's complement of 3
- Illegal: `4'd-2` = **ERROR!!**

Special Number Characters

- `'_'` (underscore): used for readability
 - Example: `32'h21_65_bc_fe` = 32-bit hexadecimal number
- `'x'` or `'X'` (unknown value)
 - Example: `12'h12x` = 12-bit hexadecimal number; LSBs unknown
- `'z'` or `'Z'` (high impedance value)
 - Example: `1'bz` = 1-bit high impedance number

NUMBER EXTENSION

If MSB is 0, x, or z, number is extended to fill MSBs with 0, x, or z, respectively

- Examples
 - 3'b01 is equal to 3'b001
 - 3'bx1 is equal to 3'bxx1
 - 3'bz is equal to 3'bzzz

If MSB is 1, number is extended to fill MSBs with 0

- Example
 - 3'b1 is equal to 3'b001

OPERATORS

Arithmetic

Bitwise

Reduction

Relational

Equality

Logical

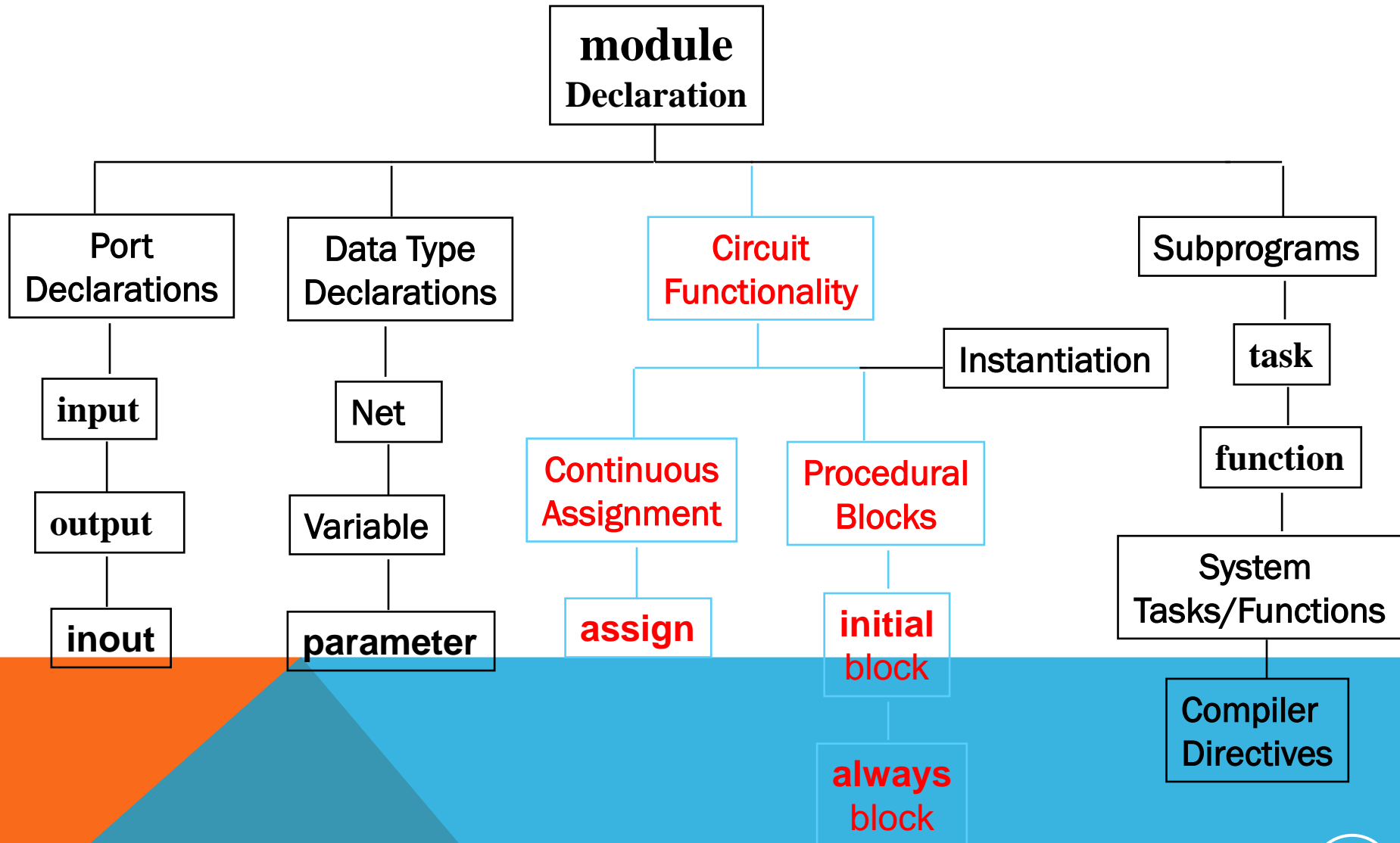
Shift

Miscellaneous

*Used in **expressions** to
describe model behavior*

Please, look for them in any Verilog book... 😊

LET'S TAKE A LOOK AT



INTRODUCTION TO VERILOG

BEHAVIORAL MODELING - CONTINUOUS ASSIGNMENTS

AUG-20

CONTINUOUS ASSIGNMENT

Model the behavior of **combinatorial logic** by using expressions and operators

- Continuous assignments can be made when the net is declared

```
wire [15:0] adder_out = mult_out + out;  
/*implicit continuous assignment */
```

is equivalent to

OR

- By using the **assign** statement

```
wire [15:0] adder_out;  
assign adder_out = mult_out + out;
```


CONTINUOUS ASSIGNMENT CHARACTERISTICS

- 1) Left-hand side of an assignment (LHS) must be a net data type
- 2) Always active: When one of the operands on the right-hand side of an assignment (RHS) changes, expression is evaluated and net on LHS is updated immediately
- 3) RHS can be an expression containing net data type, variable data type or function call (or combination of)
- 4) Delay values can be assigned to model gate delays (discussed later)

```
wire [15:0] adder_out = mult_out + out;  
/*implicit continuous assignment */
```

is equivalent to

```
wire [15:0] adder_out;  
assign adder_out = mult_out + out;
```

CONTINUOUS ASSIGNMENT – EXAMPLE

```
`timescale 1 ns/ 10 ps

module mult_acc (
    input [7:0] dataa, datab,
    input clk, aclr,
    output reg [15:0] mac_out
);

    wire [15:0] mult_out, adder_out;

    parameter mult_size = 8;

    assign adder_out = mult_out + mac_out;

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            mac_out <= 16'h0000;
        else
            mac_out <= adder_out;
    end

    mult_a #(.width_in(mult_size))
        u1 (.in_a(dataa), .in_b(datab), .mult_out(mult_out));

endmodule
```

Exercise 1

*Please go to Exercise 1
in the Exercise Manual*

CONTINUOUS ASSIGNMENT DELAY

Use #<value> notation to delay updating LHS

- Models propagation delay

```
assign #25 adder_out = mult_out + out;
```

Behavior

- 1) Operand on RHS changes
- 2) RHS reads input(s) and performs expression
- 3) If $RHS \neq LHS$, LHS scheduled to be updated with new value after delay expires
- 4) If another RHS operand changes before delay expires and a new value for LHS is calculated, then current value scheduled for LHS is cancelled and the new value for LHS is scheduled to be updated (if needed) after new delay period expires (inertial delay)
 - Requires value of RHS expression be stable for length of delay

Ignored by synthesis

INTRODUCTION TO VERILOG

BEHAVIORAL MODELING - PROCEDURAL BLOCKS

AUG-20

LFGP / TE4003

45

TWO PROCEDURAL BLOCKS

initial

- Used to initialize behavioral statement for simulation

always

- Used to describe the circuit functionality using behavioral statements
-
- ⇒ Each **always** and **initial** block represents a separate process
 - ⇒ Processes run in **parallel** and start at simulation time 0
 - ⇒ Statements inside a process execute **sequentially**
 - ⇒ **always** and **initial** blocks cannot be nested

TWO PROCEDURAL BLOCKS

always *and* **initial** blocks

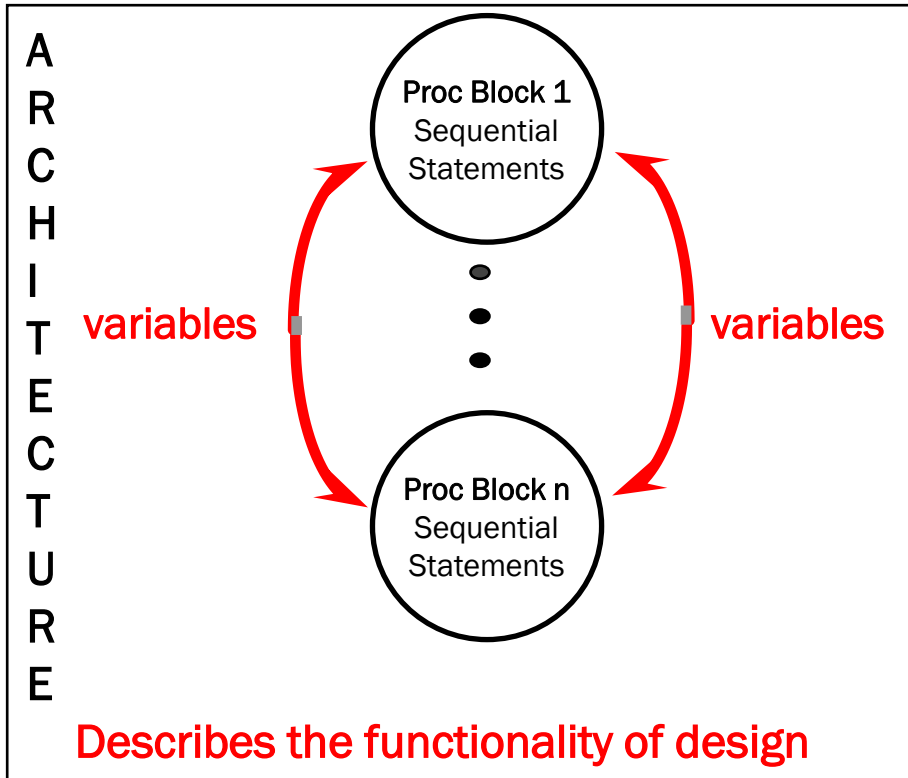
Behavioral Statements

Assignments:

blocking
nonblocking

Timing Specifications

PROCEDURAL BLOCK BEHAVIOR



Each procedural block
executes in parallel with
other procedural blocks

- Order of always/initial blocks does not matter

Within a procedural block,
statements are executed
sequentially

- Order of statements within an always/initial block does matter

PROCEDURAL BLOCK CHARACTERISTICS

LHS must be a variable data type (e.g. reg, integer, real, time)

LHS can be a bit-select or part-select

LHS can be a concatenation of any of the above

RHS can be expression containing net data type, variable data type or functional call (or combination of)

initial BLOCK

Consists of behavioral statements

Each **initial** block executes concurrently starting at time 0, executes only once and then does not execute again

Must use keywords **begin** and **end** to group behavioral statements when initial block contains more than one behavioral statement

Example uses

- Initialization
- Monitoring
- Any functionality that needs to be turned on just once

⇒ Note that though the **initial** block executes only once, the duration of **initial** block may be infinite (i.e. functionality inside may continue running for the duration of the model execution)

initial BLOCK EXAMPLE

```
module system;
```

```
    reg a, b, c, d;
```

```
    // single statement
```

```
    initial a = 1'b0;
```

```
    /* multiple statements:  
       needs to be grouped */
```

```
    initial begin
```

```
        b = 1'b1;
```

```
        #5 c = 1'b0;
```

```
        #10 d = 1'b0;
```

```
    end
```

```
    initial #20 $finish;
```

```
endmodule
```

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

always BLOCK

Consists of behavioral statements

Each always block executes concurrently starting at time 0 and executes continuously in a looping fashion

Must use keywords begin and end to group behavioral statements when always block contains more than one behavioral statement

Example uses

- Modeling a digital circuit
- Any process or functionality that needs to be executed continuously

always BLOCK EXAMPLE

```
module clk_gen
  #(parameter period = 50)
  (
    output reg clk
  );

  initial clk = 1'b0;

  always
    #(period/2) clk = ~clk;

  initial #100 $finish;

endmodule
```

Time	Statement(s) Executed
0	clk = 1'b0;
25	clk = 1'b1;
50	clk = 1'b0;
75	clk = 1'b1;
100	\$finish;

NAMING PROCEDURAL BLOCKS

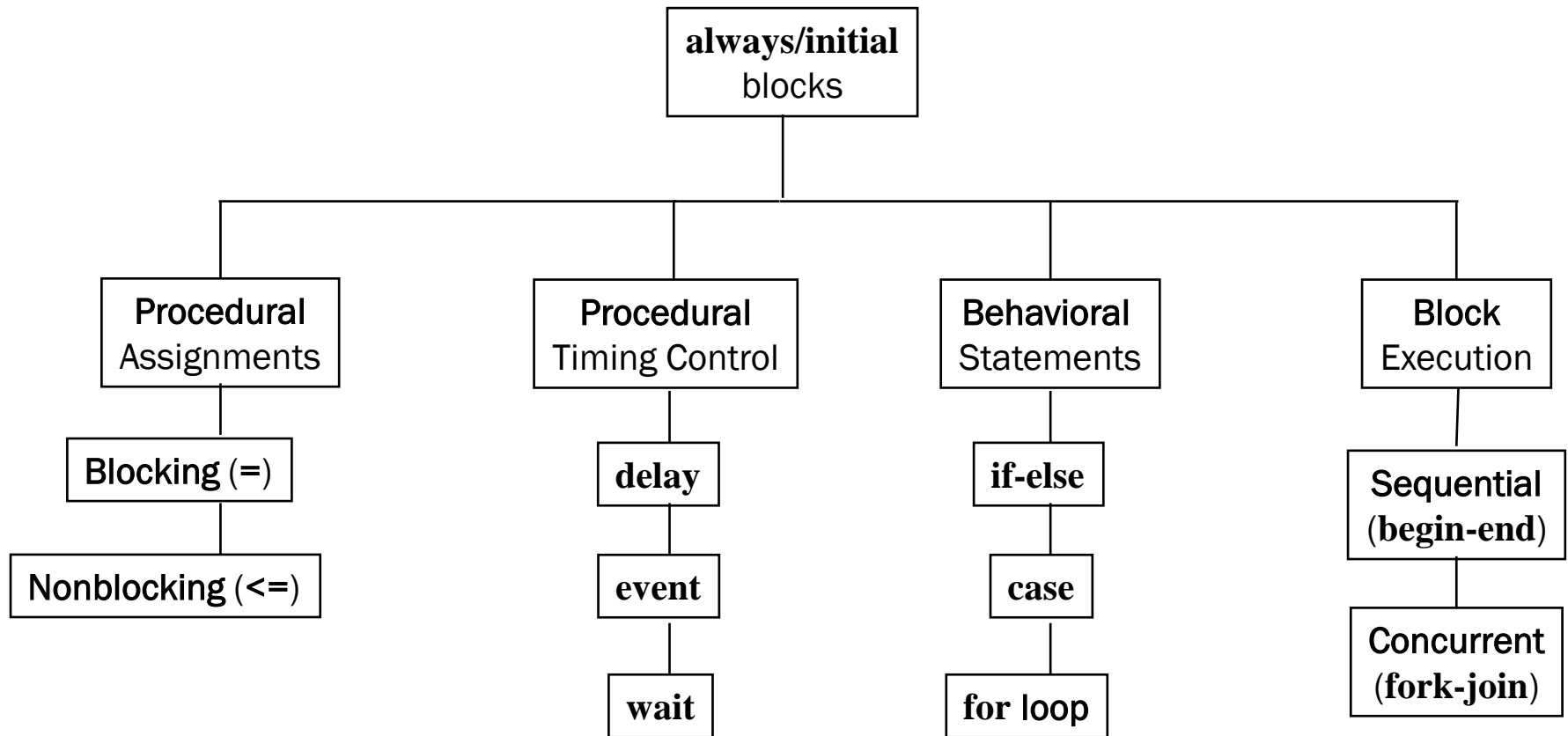
Procedural blocks may be named
by adding : <name> after
begin

Advantages

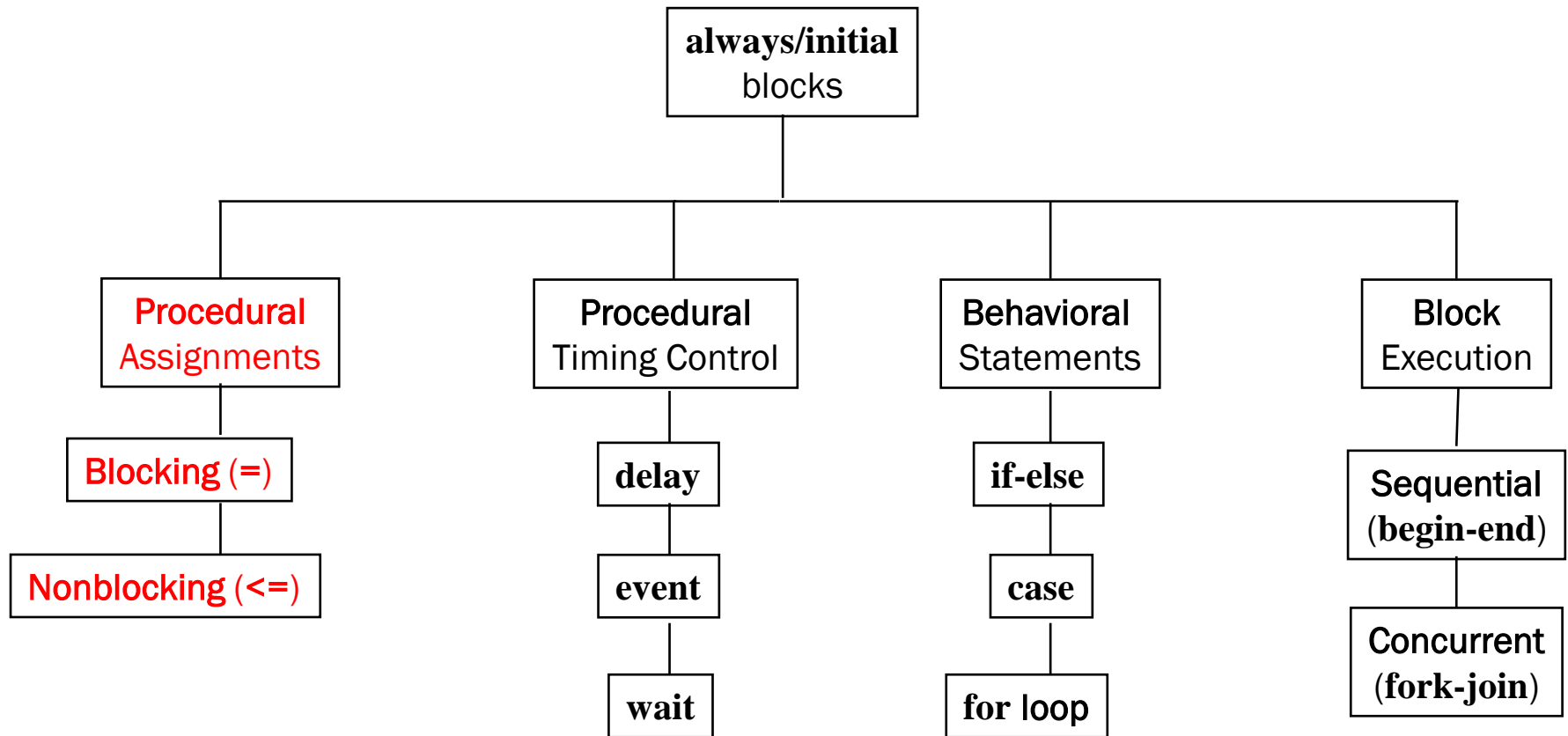
- Allows the procedural block to be referenced in other places within the code by name
- Allows declaration of objects local to the procedural block
- Allows monitoring of procedural block by name in simulation tools

```
initial  
begin : clock_init  
    clk = 1'b0;  
end  
  
always  
begin : clock_proc  
    #(period/2) clk = ~clk;  
end
```

always/initial BLOCKS



always/initial BLOCKS



PROCEDURAL ASSIGNMENT STATEMENTS

Made inside the procedural blocks (**initial/always**)

Update values of variable data types (i.e. **reg, integer, real, time**)

Place values on a variable that will **remain unchanged** until another procedural assignment updates the variable with a different value

Verilog has two types of procedural assignment statements

- Blocking
- Nonblocking

PROCEDURAL ASSIGNMENT TYPES

Blocking (=) : updates LHS assignments blocking execution of other assignments in the procedural block until finished

- RHS (inputs) sampled when statement executed
- LHS (outputs) updated immediately or after defined delay
- Statements following must wait until blocking statement completely finished and LHS assignments are made (including delay) to begin execution
- **Successive blocking procedural assignments in an edge triggered always statement do not infer stages of flip-flops**

Non-blocking (<=) : Schedules LHS assignments without blocking execution of the statements that follow in a sequential block

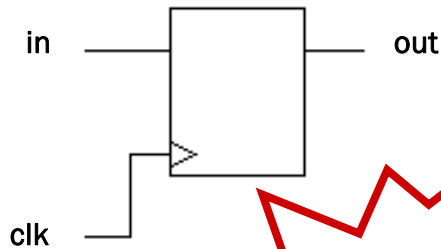
- The timing in an assignment is relative to the absolute time at which the procedural block was triggered into being executed
- RHS (inputs) sampled when statement executed
- LHS (outputs) scheduled to be updated at end of the time step or after delay expires
- Statements following **do not wait** until non-blocking LHS assignments are made to begin execution
- **As synthesis tools ignore all timing from the model, and non-blocking assignments are scheduled to occur at the same time, successive assignments in an edge triggered always statement will each infer flip-flops**

BLOCKING (=) VS NONBLOCKING (<=)

Blocking (=)

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

Synthesized result:

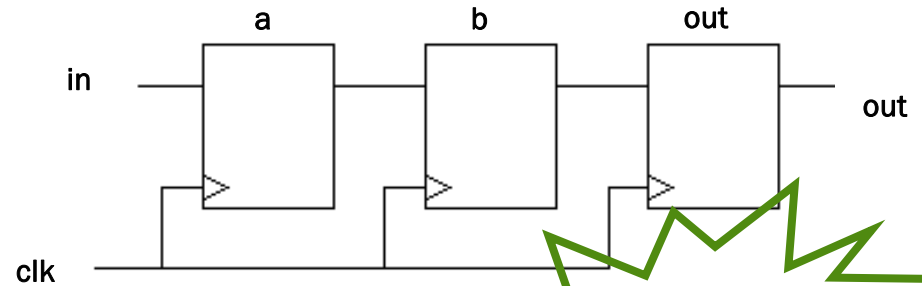


Incorrect pipeline
implementation

Nonblocking (<=)

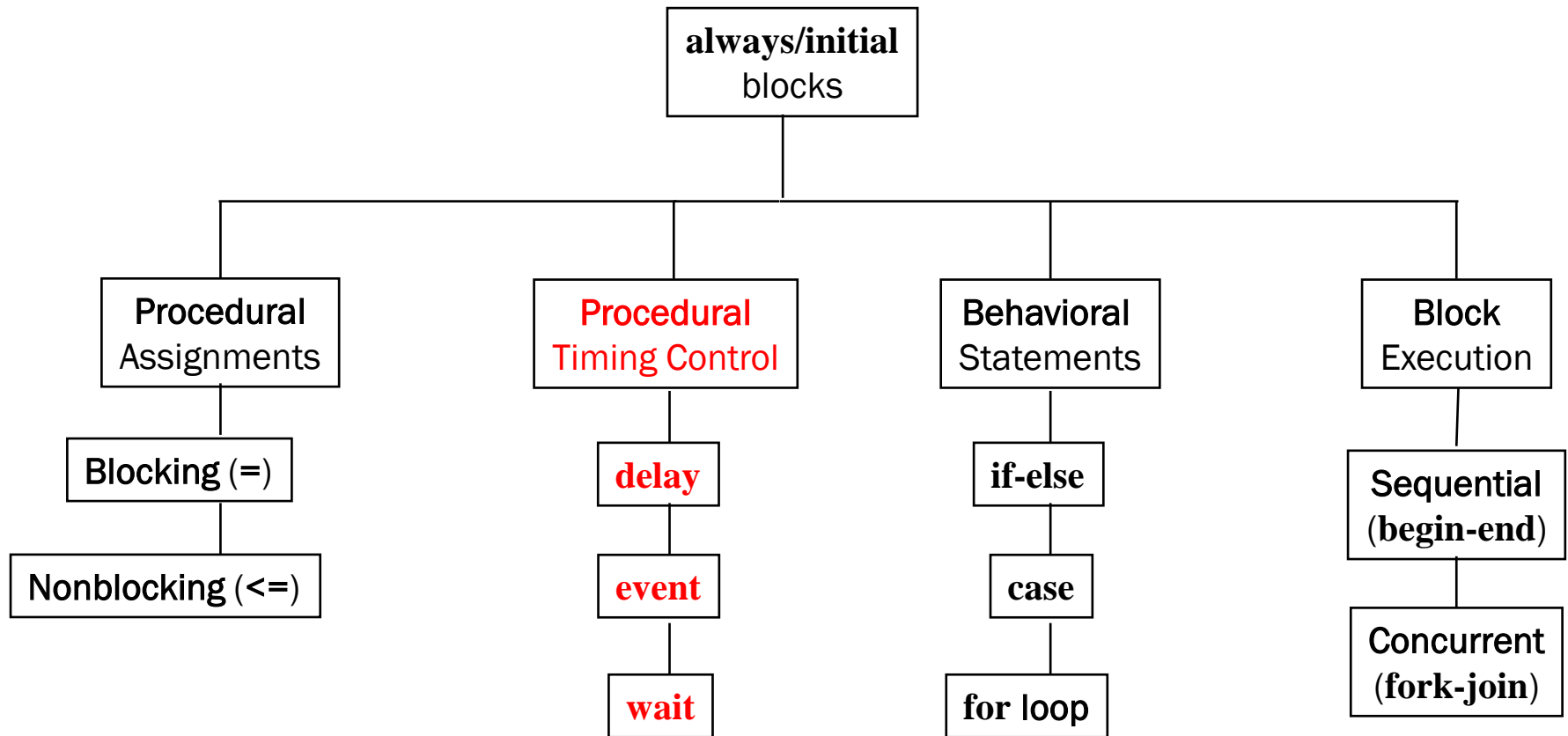
```
always @ (posedge clk)
begin
    a <= in;
    b <= a;
    out <= b;
end
```

Synthesized result:



Correct pipeline
implementation

always/initial BLOCKS



EVENT CONTROL

Provides edge-sensitive control

Symbolized by the @ symbol

@(expression)

Pauses execution of procedural statements until event occurs (i.e. expression changes value)

To test for multiple events (logical OR of events list)

- Comma (,)
 - Verilog '2001 and later
- or

initial begin

// Rising edge control (inter-assignment)

@ (**posedge** clk) r1 = r2;

// Falling edge control (intra-assignment)

r3 = @(**negedge** clk) r4;

// Either edge control

@ (a) r5 = r6;

// Either edge control using more

// complex expression

@ (a ^ b & c) r7 = r8;

// Logical OR of two events using comma

@ (**posedge** clk, enable) r9 = r10;

// Logical OR of two events using

// “or” keyword

r11 = @ (**posedge** clk **or** enable) r12;

end

EVENT CONTROL & SENSITIVITY LISTS

Use event control at the beginning of always block to control when block begins execution

- Each execution of always block requires event to be satisfied
- Forces always block to be “sensitive” to the items in event control

Also referred to as a **sensitivity list**

Supported by synthesis tools

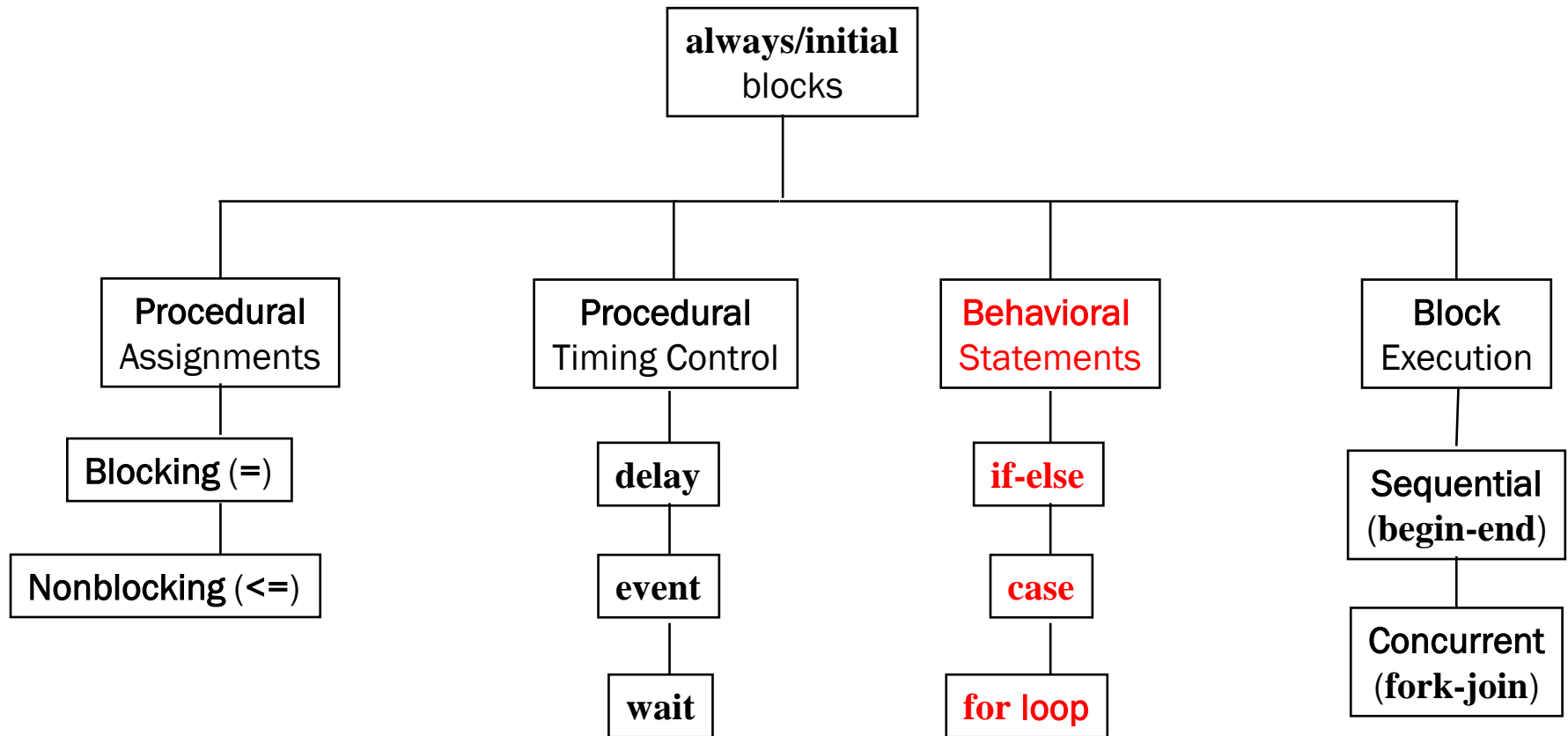
Format

```
always @(sensitivity_list) begin  
  -- Statement_1  
  -- .....  
  -- Statement_N  
end
```

Example

```
// Process executes whenever  
//   a, b, c or d changes in value  
always @(a, b, c, d) begin  
  #15 y = (a ^ b) & (c ~ | d);  
end
```

always/initial BLOCKS



BEHAVIORAL STATEMENTS

Describe behavior and express order

Must be used inside procedural block

Behavior statements

- If – else
- Case
- loop

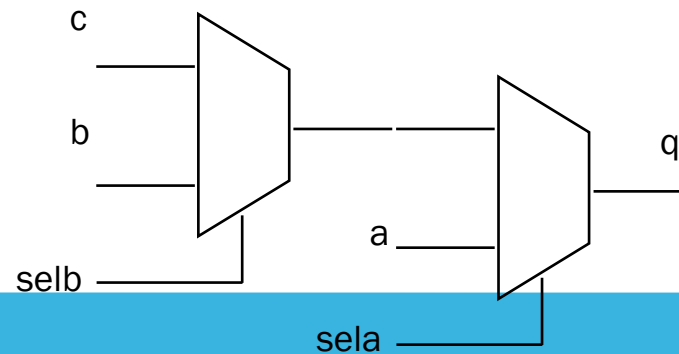
if – else STATEMENT

Format

```
if <condition1>  
    {sequence of statement(s)}  
else if <condition2>  
    {sequence of statement(s)}  
    ...  
else  
    {sequence of statement(s)}
```

Example

```
always @ (sela, selb, a, b, c) begin  
    if (sela)  
        q = a;  
    else if (selb)  
        q = b;  
    else  
        q = c;  
end
```



if – else STATEMENT

Conditions are evaluated in order from top to bottom

- Prioritization

The first condition, that is true, causes the corresponding sequence of statements to be executed

If all conditions are false, then the sequence of statements associated with the final “else” clause are evaluated

Exercise 2

*Please go to Exercise 2
in the Exercise Manual*

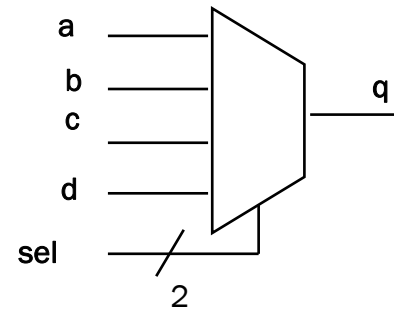
case STATEMENT

Format

```
case {expression}
  <condition1> :
    {sequence of statements}
  <condition2> :
    {sequence of statements}
  ...
  default : -- (optional)
    {sequence of statements}
endcase
```

Example

```
always @ (sel, a, b, c, d) begin
  case (sel)
    2'b00 : q = a;
    2'b01 : q = b;
    2'b10 : q = c;
    default : q = d;
  endcase
end
```



case STATEMENT

Conditions are evaluated in order

First matching value is chosen

Treats both X and Z as actual logic values

default clause represents all other possible conditions that are not specifically stated

Verilog does not require (though it is recommended) that

- All possible conditions be considered
- All conditions be unique

OTHER FORMS OF case STATEMENT

casez

- Treats both Z and ? in the case condition as don't cares

```
casez (encoder)
    4'b1??? : high_lvl = 3;
    4'b01?? : high_lvl = 2;
    4'b001? : high_lvl = 1;
    4'b0001 : high_lvl = 0;
    default : high_lvl = 0;
endcase
```

- if **encoder** = 4'b1z0x, then **high_lvl** = 3

casex

- Treats X, Z and ? in the case conditions as don't cares, instead of logic values

```
casex (encoder)
    4'b1xxx : high_lvl = 3;
    4'b01xx : high_lvl = 2;
    4'b001x : high_lvl = 1;
    4'b0001 : high_lvl = 0;
    default : high_lvl = 0;
endcase
```

- if **encoder** = 4'b1z0x, then **high_lvl** = 3

Exercise 3

*Please go to Exercise 3
in the Exercise Manual*

loop STATEMENT

forever loop - executes continually

repeat loop - executes a fixed number of times

while loop - executes if expression is true

for loop - executes once at the start of the loop and then executes if expression is true

⇒ Loop statements - used for repetitive operations

FOREVER AND REPEAT LOOP

forever loop - executes continually

```
initial begin  
    clk = 0;  
    forever #25 clk = ~clk;  
end
```

Clock with period
of 50 time units

Not synthesizable!

repeat loop - executes a fixed number of times

```
if (rotate == 1)  
    repeat (8) begin  
        tmp = data[15];  
        data = {data << 1, tmp};  
    end
```

Repeats a rotate
operation 8 times

while LOOP

while loop – executes if **expression** is true

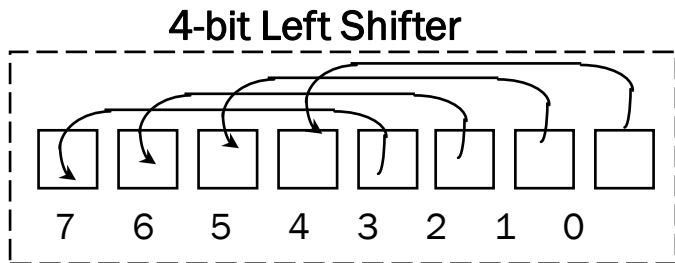
```
initial begin  
  count = 0;  
  while (count < 101) begin  
    $display ("Count = %d", count);  
    count = count + 1;  
  end  
end
```

Counts from 0 to 100
Exits loop at count 101

Not synthesizable!

for LOOP

- for loop - executes once at the start of the loop and then executes if expression is true



```
// declare the index for the FOR LOOP  
integer i;
```

```
always @(inp, cnt) begin
```

```
    result[7:4] = 0;
```

```
    result[3:0] = inp;
```

```
    if (cnt == 1) begin
```

```
        for (i = 4; i <= 7; i = i + 1) begin
```

```
            result[i] = result[i-4];
```

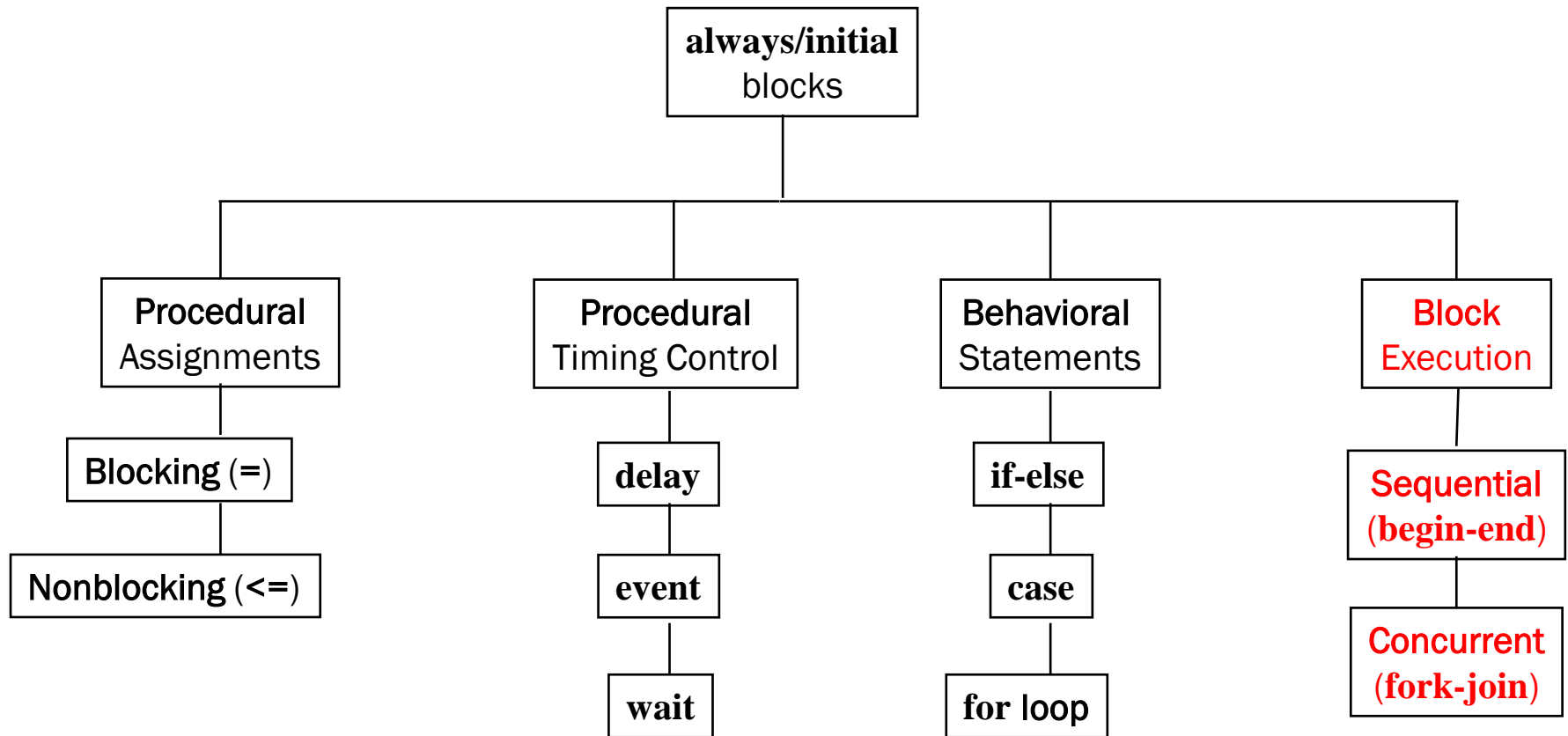
```
        end
```

```
        result[3:0] = 0;
```

```
    end
```

```
end
```

always/initial BLOCKS



BLOCK EXECUTION

Groups statements together within a procedural block so they are perceived as a single statement

Two types:

- Sequential blocks
 - Statements between **begin** and **end** execute sequentially
 - If there are multiple behavioral statements inside an **initial** or **always** block and you want the statements to execute sequentially, the statements must be grouped using the keywords **begin** and **end**
- Parallel blocks
 - Statements between **fork** and **join** execute in parallel
 - If there are multiple behavioral statements inside an **initial** or **always** block and you want the statements to execute in parallel, the statements must be grouped using the keywords **fork** and **join**
 - **Not supported by synthesis**
 - Use nonblocking assignments to achieve this behavior

SEQUENTIAL vs PARALLEL BLOCKS

Sequential and parallel blocks can be nested

initial fork

#10 a = 1;

#15 b = 1;

begin

#20 c = 1;

#10 d = 1;

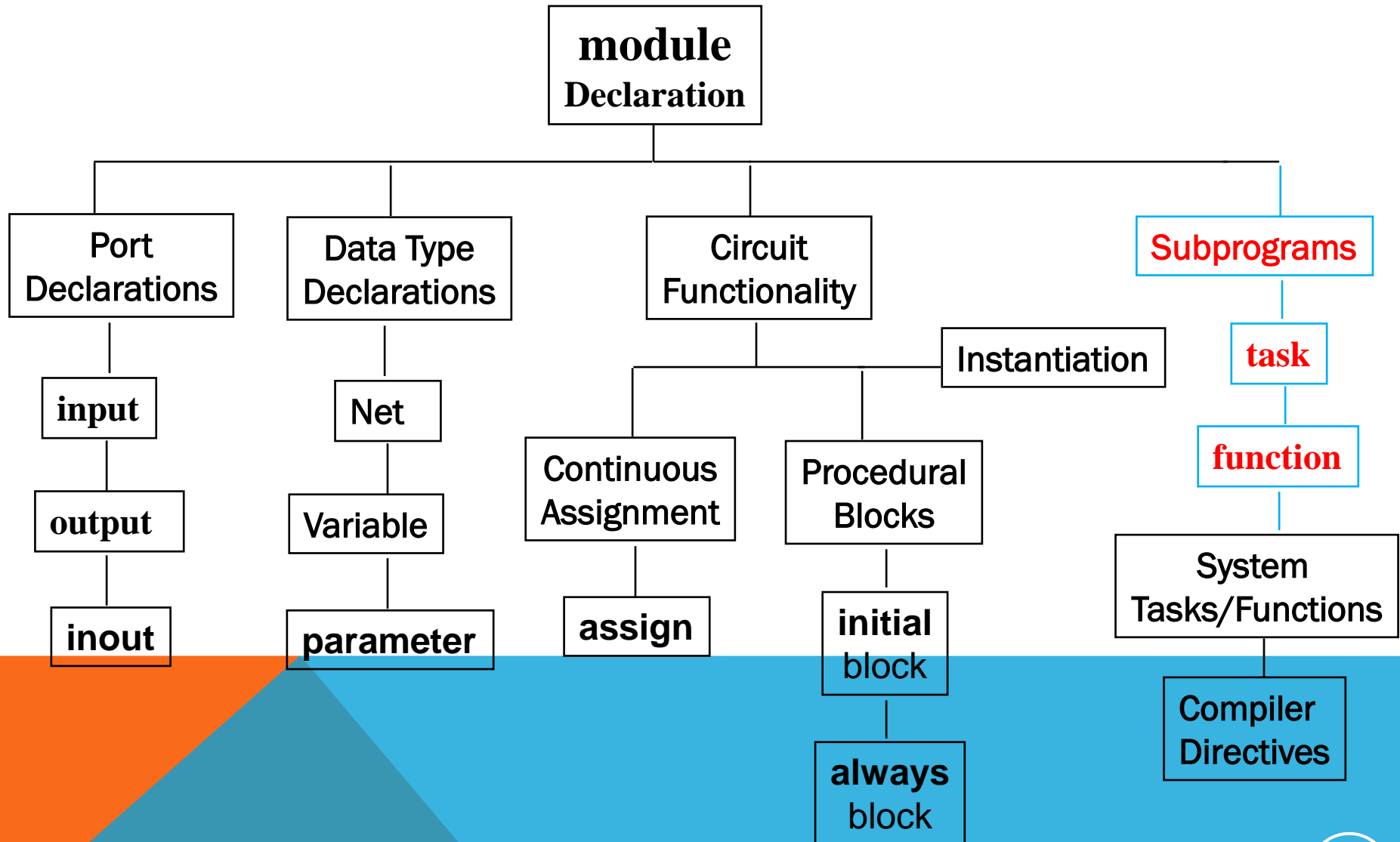
end

#25 e = 1;

join

Time	Statement(s) Executed
10	a = 1'b1;
15	b = 1'b1;
20	c = 1'b1;
25	e = 1'b1;
30	d = 1'b1;

LET'S TAKE A LOOK AT



INTRODUCTION TO VERILOG

BEHAVIORAL MODELING - TASKS & FUNCTIONS

AUG-20

LFGP / TE4003

80

FUNCTIONS AND TASKS

Function and Tasks are subprograms

Consist of behavioral statements (like a procedural block)

Defined within a module

Uses

- Replacing repetitive code
- Enhancing readability

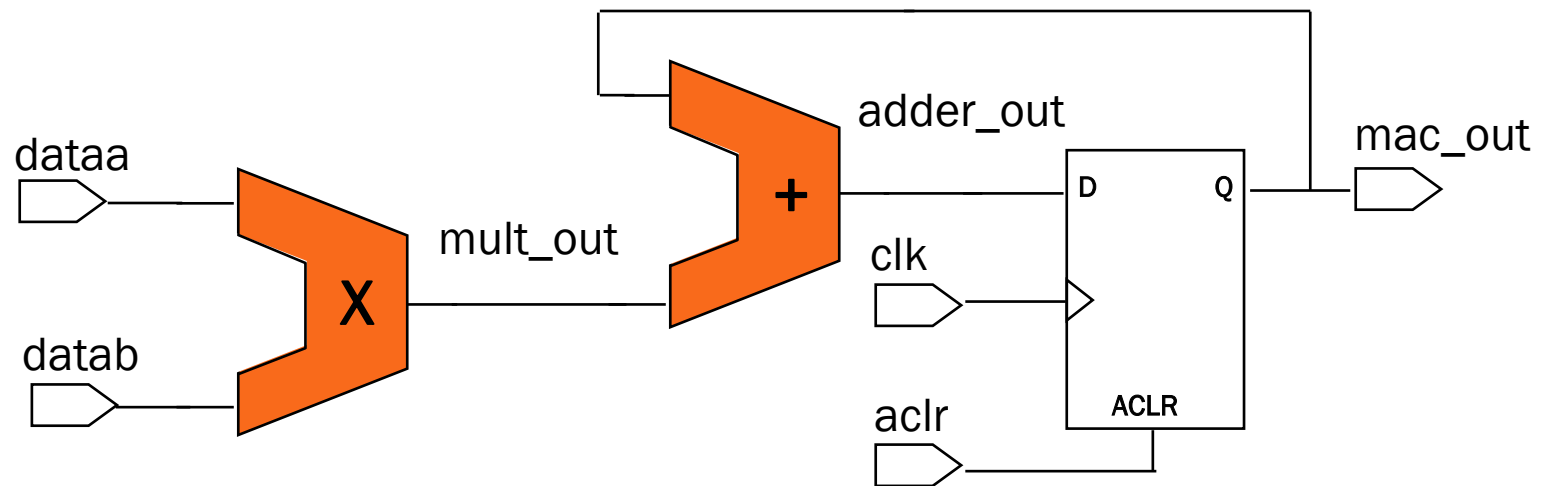
Function

- Return a value based on its inputs
- Produces combinatorial logic
- Used in expressions: **assign** mult_out = **mult** (ina, inb);

Tasks

- Like procedures in other languages
- Can be combinatorial or registered
- Task are invoked as statement: **stm_out** (nxt, first, sel, filter);

CREATE A FUNCTION FOR THE MULTIPLIER



FUNCTION DEFINITION – MULTIPLIER

```
function [15:0] mult;  
  input [7:0] a, b;  
  reg [15:0] r;  
  integer i;  
begin  
  if (a[0] == 1)  
    r = b;  
  else  
    r = 0;  
    for (i = 1; i <= 7; i = i + 1) begin  
      if (a[i] == 1)  
        r = r + b << i;  
    end  
    mult = r;  
end  
endfunction
```

FUNCTION INVOCATION

```
`timescale 1 ns/ 10 ps
```

```
module mult_acc (  
    input [7:0] dataa, datab,  
    input clk, aclr,  
    output reg [15:0] mac_out  
);  
  
    wire [15:0] mult_out, adder_out;  
  
    parameter set = 10;  
    parameter hld = 20;
```

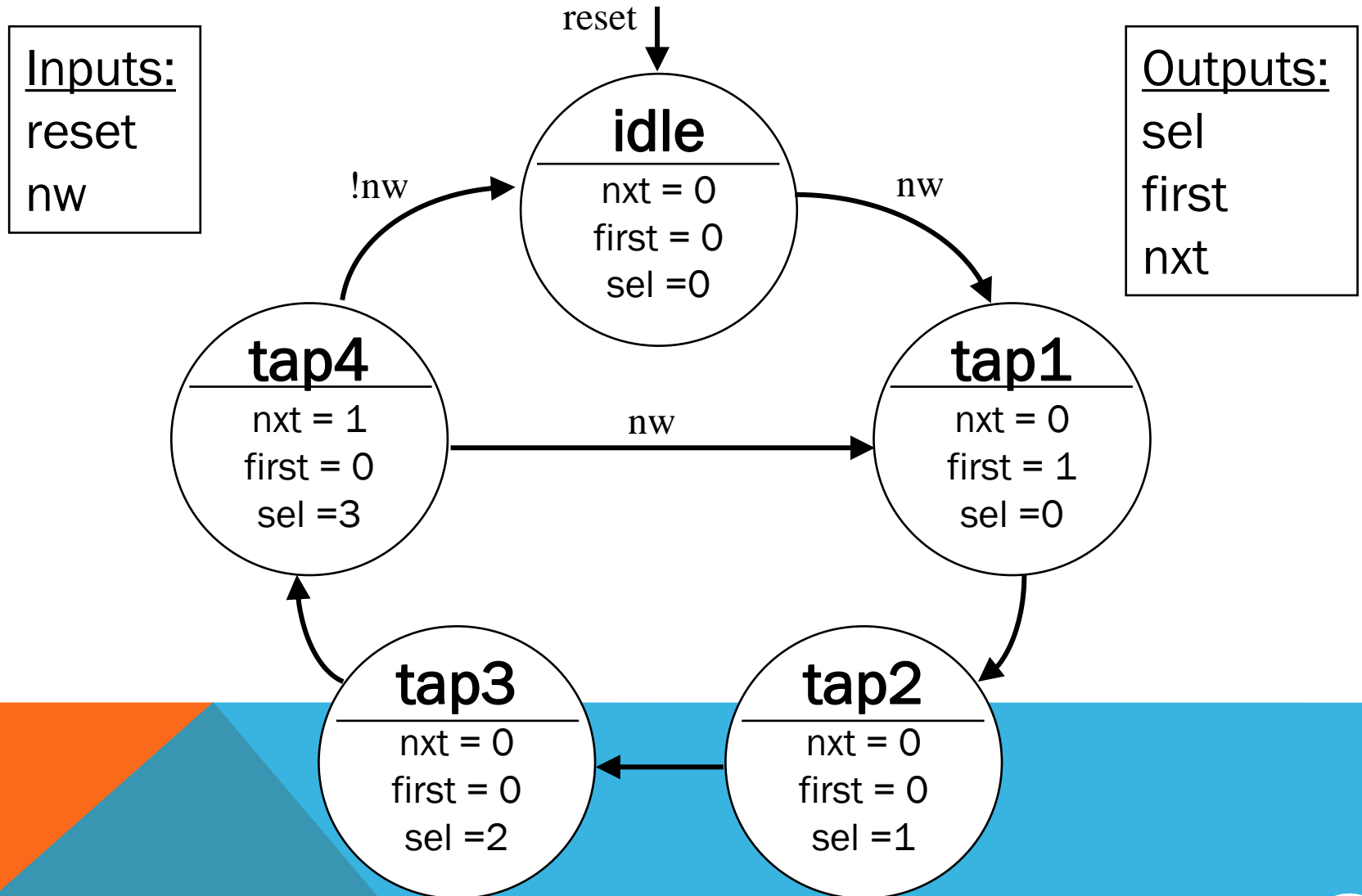
```
    assign adder_out = mult_out + mac_out;
```

```
    always @ (posedge clk, posedge aclr) begin  
        if (clr)  
            mac_out <= 16'h0000;  
        else  
            mac_out <= adder_out;  
    end
```

```
    assign mult_out = mult (dataa, datab)
```

```
endmodule
```

CREATE TASK FOR STATE MACHINE OUTPUT



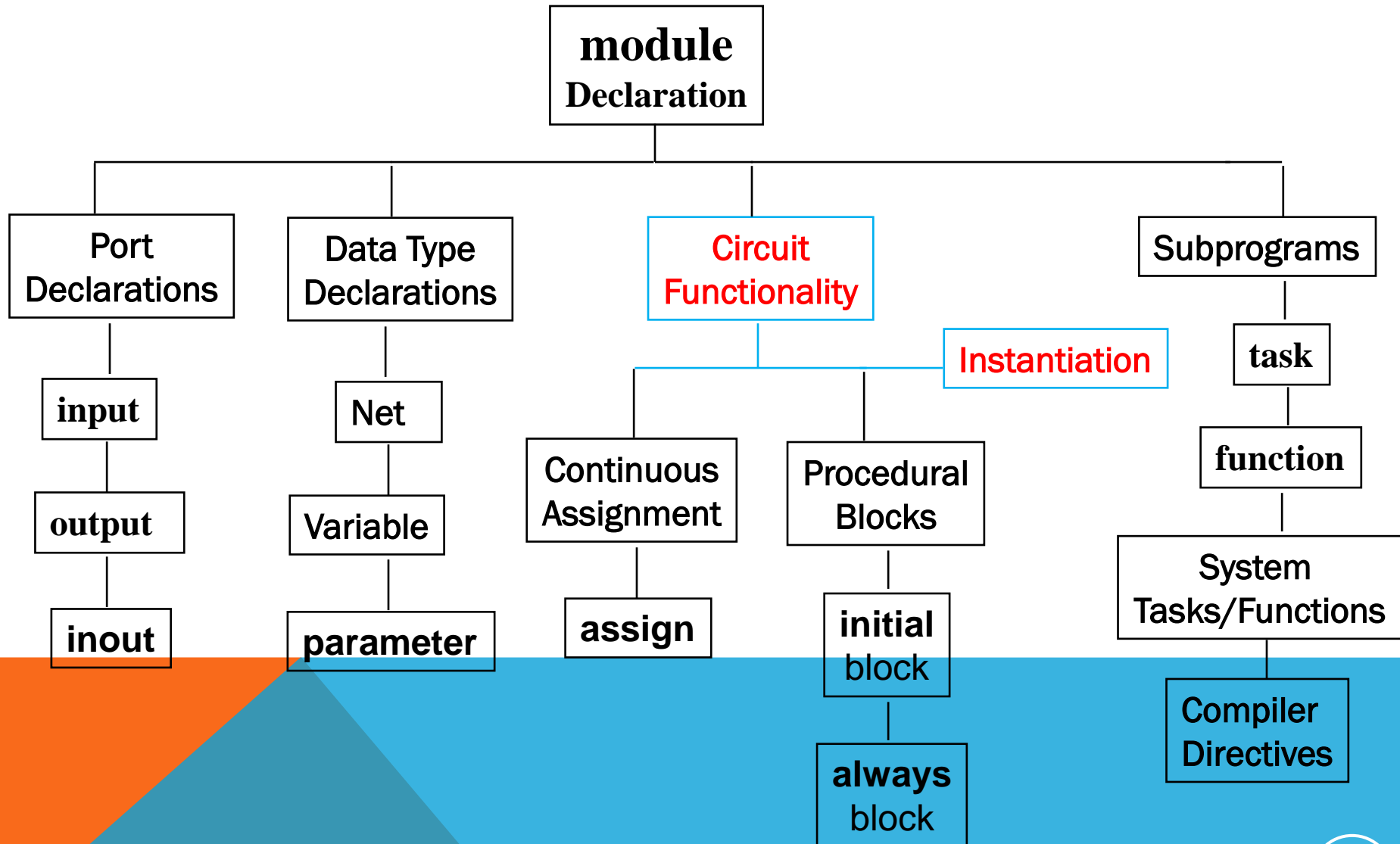
TASK DEFINITION

```
task stm_out;  
    output reg nxt, first;  
    output reg [1:0] sel;  
    input [2:0] filter;  
    parameter idle=0, tap1=1, tap2=2, tap3=3, tap4=4;  
begin  
    nxt = 0;  
    first = 0;  
    case (filter)  
        tap1: begin sel = 0; first = 1; end  
        tap2: sel = 1;  
        tap3: sel = 2;  
        tap4: begin sel = 3; nxt = 1; end  
        default: begin nxt = 0; first = 0; sel = 0; end  
    endcase  
end  
endtask
```

TASK INVOCATION

```
module stm_fir (  
    input clk, reset, nw,  
    output reg nxt, first,  
    output reg [1:0] sel  
);  
  
    reg [2:0] filter;  
    parameter idle=0, tap1=1, tap2=2, tap3=3, tap4=4;  
    always @(posedge clk or posedge reset) begin  
        if (reset)  
            filter = idle;  
        else  
            case (filter)  
                idle: if (nw==1) filter = tap1;  
                tap1: filter = tap2;  
                tap2: filter = tap3;  
                tap3: filter = tap4;  
                tap4: if (nw==1) filter = tap1;  
                    else filter = idle;  
            endcase  
        end  
  
        always @(filter)  
            // Task Invocation  
            stm_out (nxt, first, sel, filter);  
    endmodule
```

LET'S TAKE A LOOK AT



INTRODUCTION TO VERILOG

STRUCTURAL MODELING

AUG-20

LFGP / TE4003

89

STRUCTURAL MODELING

Defines a function and the structure of a digital design

Implies hierarchical design

Types

Gate-level modeling - instantiating built-in Verilog gate primitives

- and, nand, or, nor, xor, xnor
- buf, bufif0, bufif1, not, notif0, notif1

User-defined primitives – instantiating primitives created by designer













- Not discussed

Module instantiation - instantiating user-created lower-level designs (components)

Switch Level Modeling - instantiating Verilog built-in switch primitives

- nmos, rnmos, pmos, rpmos, cmos, rcmos
- tran, rtran, tranif0, rtranif0, tranif1, rtrainif1, pullup, pulldown
- ⇒ Not discussed

GATE-LEVEL MODELING

Primitive	Name	Function	Primitive	Name	Function
	and	n-input AND gate		buf	n-output buffer
	nand	n-input NAND gate		not	n-output buffer
	or	n-input OR gate		bufif0	tristate buffer lo enable
	nor	n-input NOR gate		bufif1	tristate buffer hi enable
	xor	n-input XOR gate		notif0	tristate inverter lo enable
	xnor	n-input XNOR gate		notif1	tristate inverter hi enable

INSTANTIATION OF GATE PRIMITIVES

Format

```
<gate_name> #<delay> <instance_name> (port_list);
```

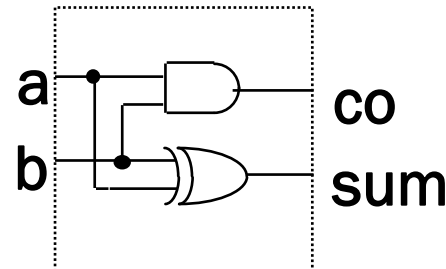
- **<gate_name>**
 - The name of gate (e.g. AND, NOR, BUFIF0...)
- **#<delay>**
 - Delay through gate
 - Optional
- **<instance_name>**
 - Unique name applied to individual gate instance
 - Optional
- **(port_list)**
 - List of signals to connect to gate primitive

CONNECTING GATE PRIMITIVE PORTS

For Verilog gate primitives, the first port on the port list is the output, followed by the inputs.

- **<gate_name>**
 - and
 - xor
- **#delay** (optional)
 - 2 time unit for the and gate
 - 4 time unit for the xor gate
- **<instance_name>** (optional)
 - u1 for the and gate
 - u2 for the xor gate
- **(port_list)**
 - (co, a, b) - (output, input, input)
 - (sum, a, b) - (output, input, input)

```
module half_adder (  
    output co, sum,  
    input a, b  
);  
  
    parameter and_delay = 2;  
    parameter xor_delay = 4;  
  
    and #and_delay u1(co, a, b);  
    xor #xor_delay u2(sum, a, b);  
  
endmodule
```



MODULE INSTANTIATION

Format

```
<component_name> #<delay> <instance_name> (port_list);
```

- **<component_name>**
 - The module name of your lower-level component
- **#<delay>**
 - Delay through component
 - Optional
- **<instance_name>**
 - Unique name applied to individual component instance
- **(port_list)**
 - List of signals to connect to component

CONNECTING MODULE INSTANTIATION PORTS

Two methods to define port connections

- By ordered list
- By name

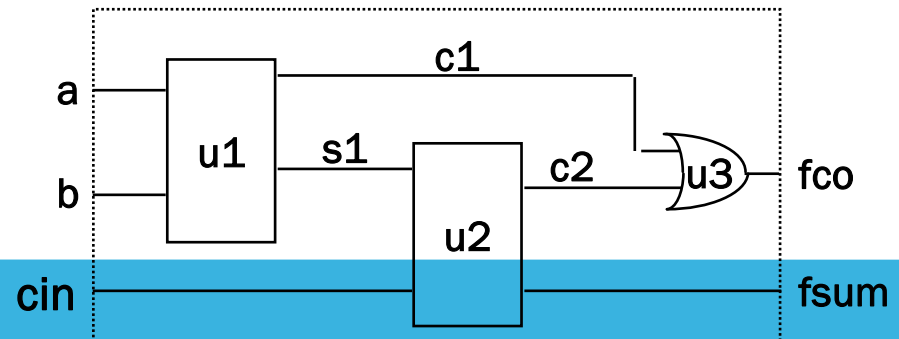
By ordered list (1st half adder*)

- Port connections defined by the order of the port list in the lower-level module declaration
 - **module** half_adder (co, sum, a, b);
- Order of the port connections does matter
 - co -> c1, sum -> s1, a -> a, b -> b

By name (2nd half_adder*)

- Port connections defined by name
- Recommended method
- Order of the port connections does not matter
 - a -> s1, b -> cin, sum -> fsum, co -> c2

```
module full_adder (  
    output fco, fsum,  
    input cin, a, b  
);  
    wire c1, s1, c2;  
  
    half_adder u1 (c1, s1, a, b);  
    half_adder u2 (.a(s1), .b(cin),  
        .sum(fsum), .co(c2));  
    or u3(fco, c1, c2);  
  
endmodule
```



INTRODUCTION TO VERILOG

SUMMARY

AUG-20

SUMMARY

Verilog is a hardware description language used to model hardware

Basic building block is the module statement

All objects must have a defined data type

Two types of assignments: continuous and procedural

Not all Verilog code is synthesizable

When writing RTL code, the two types of procedural blocks are **combinational** and **sequential**

**DO NOT EVER code a HDL program
like a C or any other HLL program**