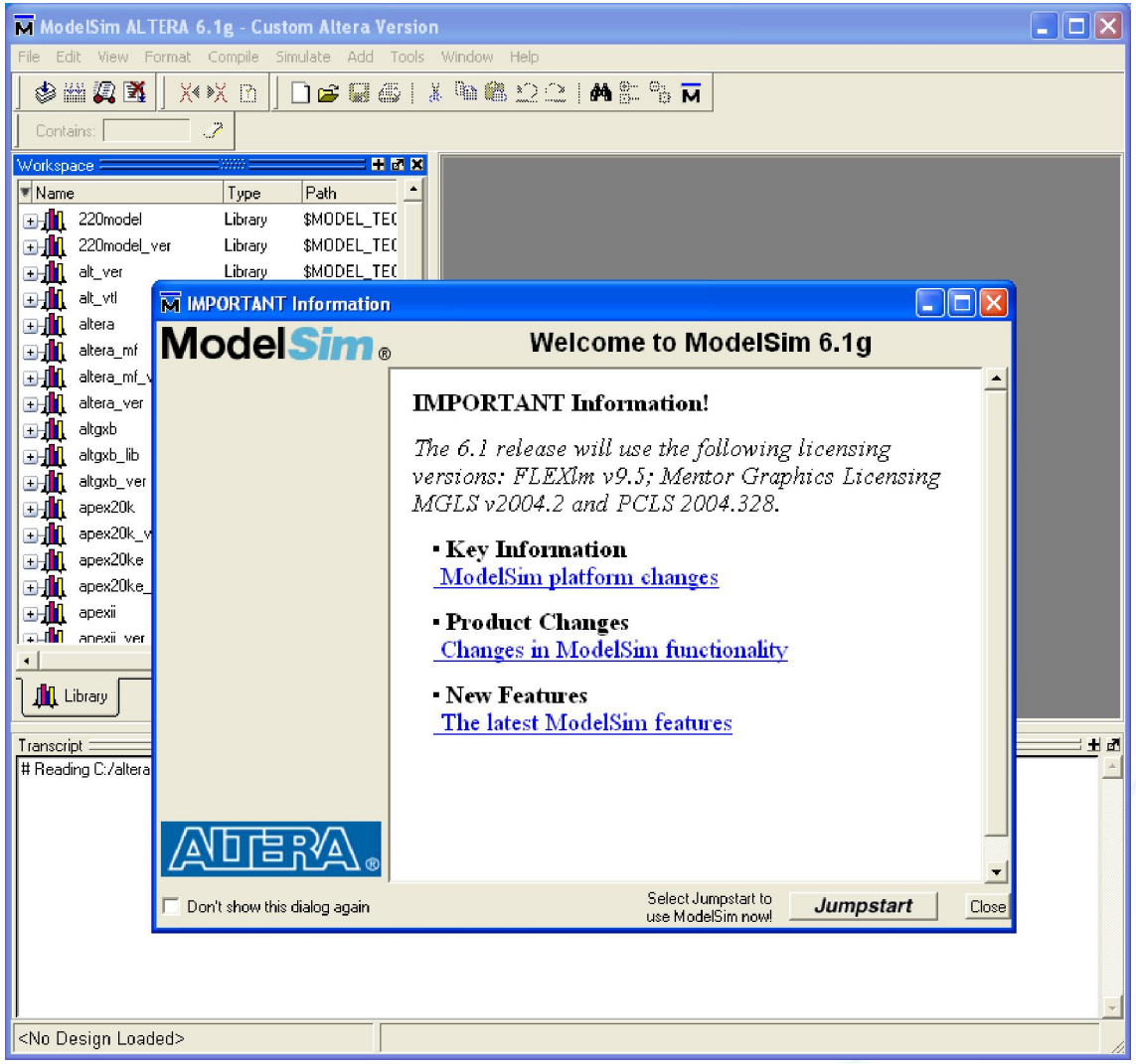
Introduction to ModelSim

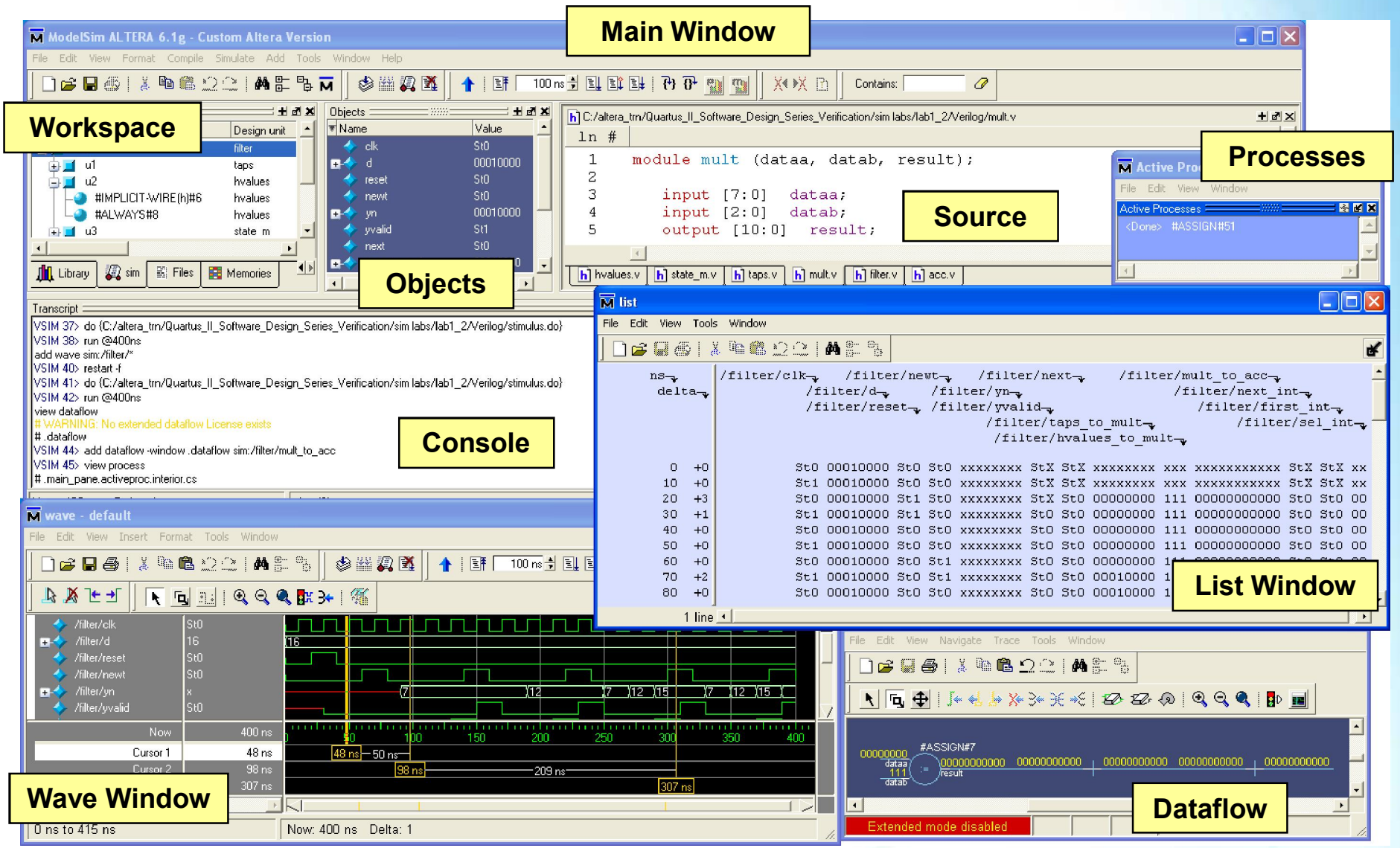
Starting ModelSim



modelsim.ini File

- ASCII file used by ModelSim, controlled by the user
- Default file provided in the ModelSim installation directory
- **modelsim.ini** used by the compiler and the simulator
- Stores initialization information
  - Location of libraries
  - Location of startup file
  - Other default settings for ModelSim
- ModelSim searches for the **modelsim.ini** in the following order:
  1. Environment variable called **MODELSIM** which points directly to the **modelsim.ini** file to be used
  2. A file called **modelsim.ini** located in the current working directory
  3. The default **modelsim.ini** file in the ModelSim software installation tree

Mentor Graphics ModelSim User Interface

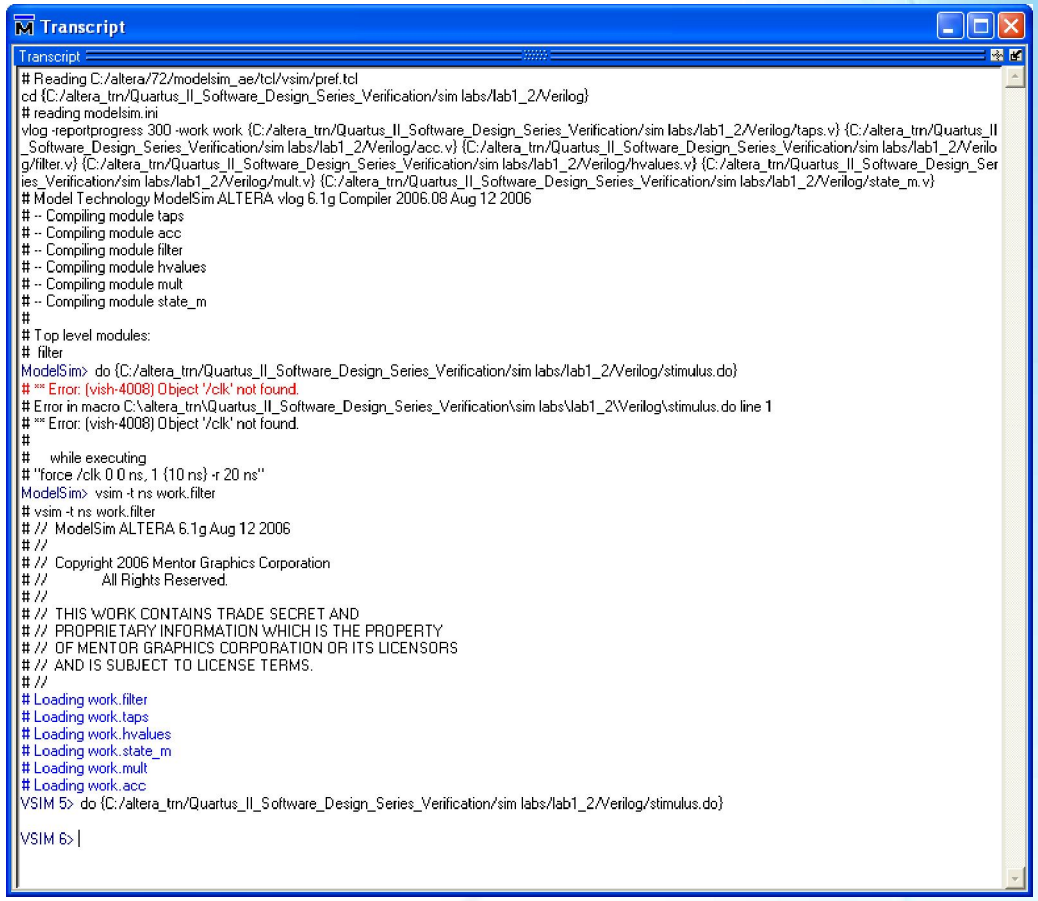


ModelSim User Interface Features

- Ten windows and panes: Main (Workspace), Active Process, Dataflow, List, Locals, Memory, Objects, Source, Watch, & Wave
- Open from the View menu
- Drag & drop
  - Selecting HDL items in one window, drag from one window to another
  - Drag from the Dataflow, Objects, Source, Structure, Locals, and Wave windows
  - Drop into Wave or List window to see simulation results

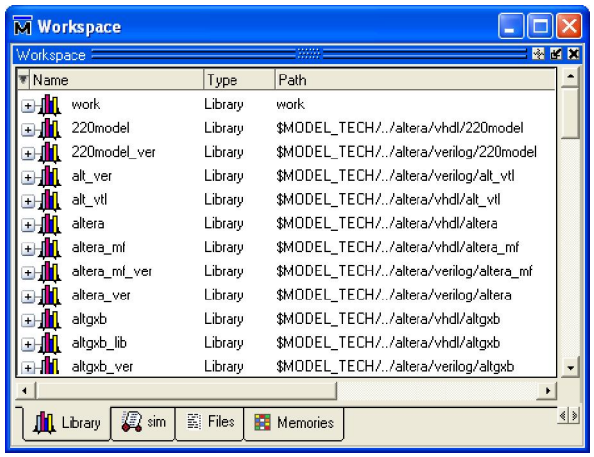
Main Window: Transcript

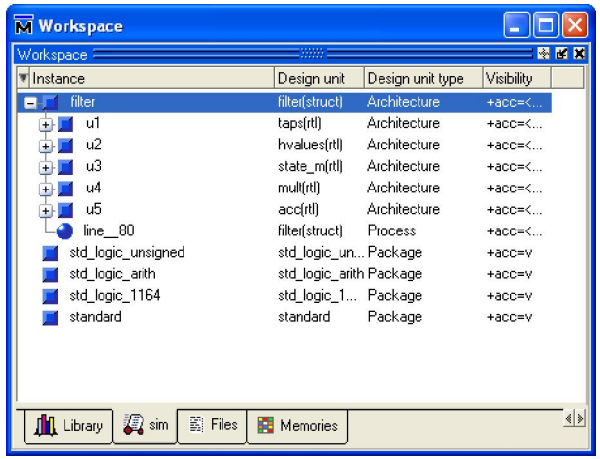
- `ModelSim>` prompt before design is loaded
  - View help, edit libraries, edit source code without invoking a design
- `VSIM>` prompt displayed after design is loaded
- Transcribes simulator activity
  - Commands
  - Messages
  - Assertion statements

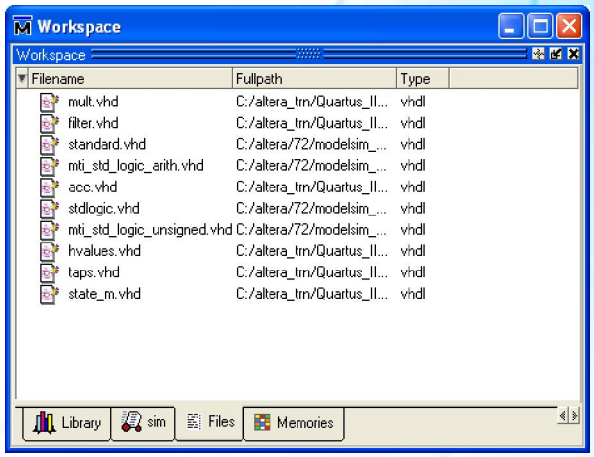


Main Window: Workspace

- **View ⇒ Workspace**
- Four tabs
  - **Library**: add new or edit existing libraries
  - **Sim**: view project structure after starting simulation
  - **Files**: files that make up project and simulation
  - **Memories**: manage memory files for simulation

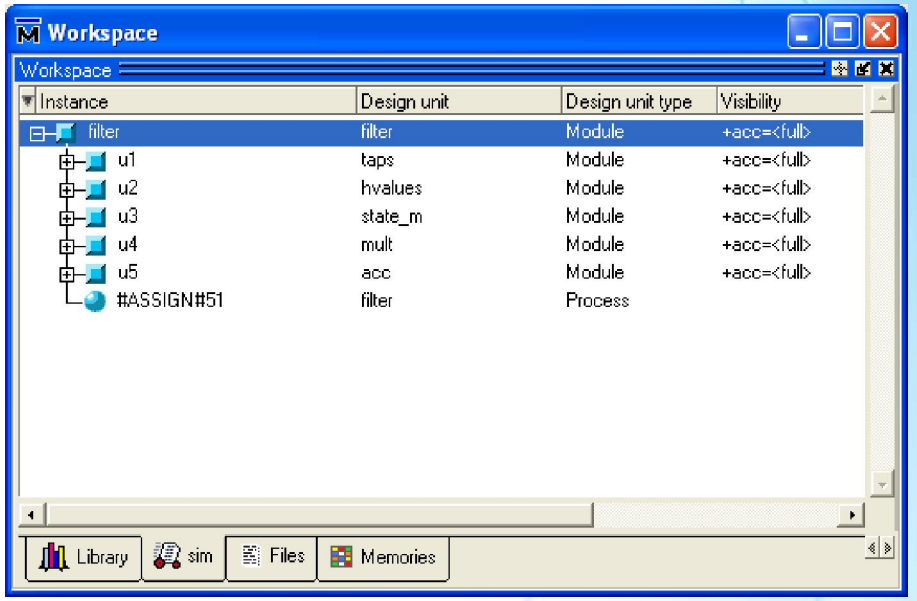






Workspace Window: sim Tab

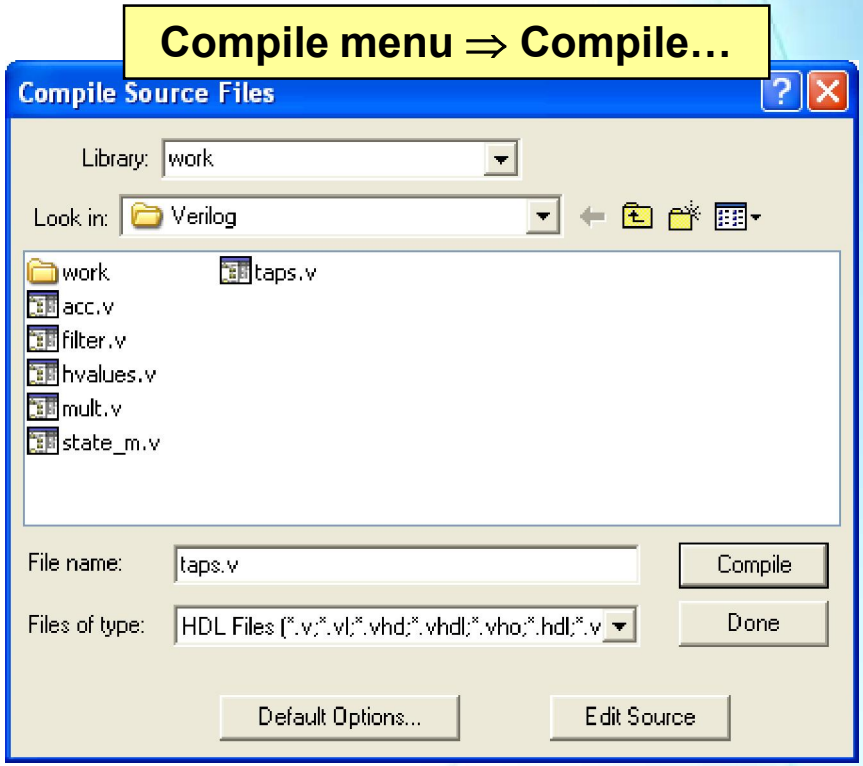
- Hierarchical view of the structure of the design
  - Verilog ( ) - module instantiation, named fork, named begin, task, and function
  - Instantiation label, entity/module, architecture
  - Selected item becomes current region for Source window, updates Active Processes window



Command: **`view structure`**

Compile Source Files

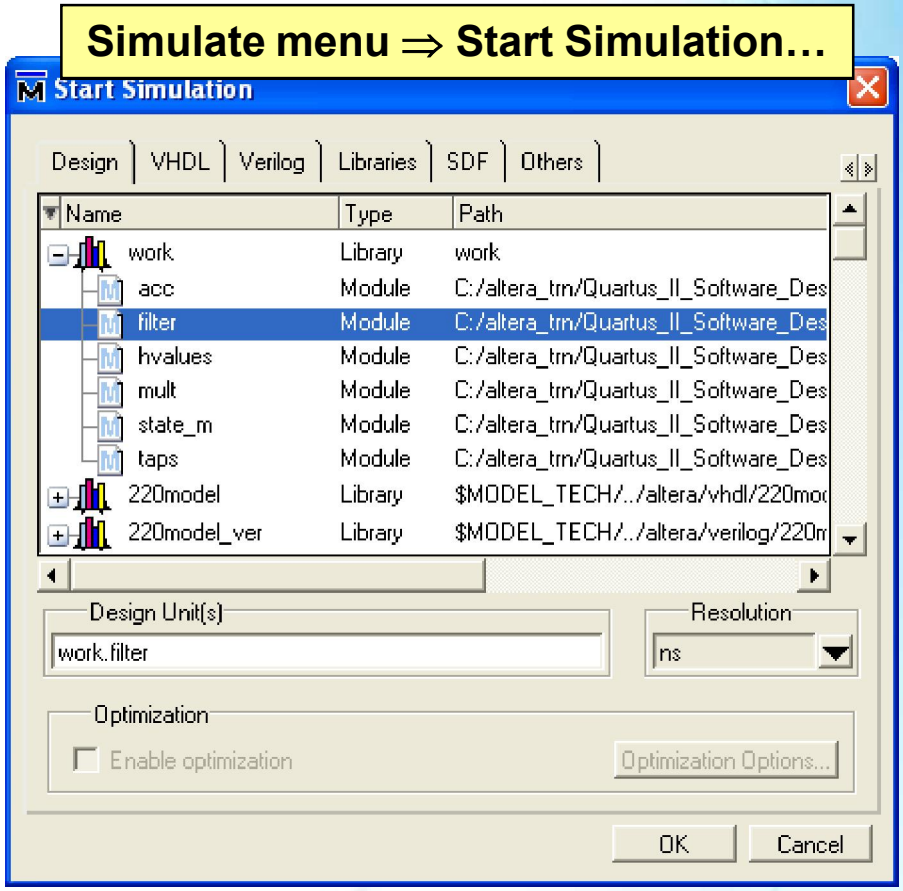
- Used to select design files to compile for simulation
- Select default options:
  - VHDL options
    - Language syntax (1987 or 1993)
  - Verilog options



VHDL command: **vcom -work** *<library_name>* *<file1>*.**vhd** *<file2>*.**vhd**
Verilog command: **vlog -work** *<library_name>* *<file1>*.**v** *<file2>*.**v**

Start Simulation

- Select design to load for simulation
- Select:
  - Simulator resolution
    - Supports multipliers of 1, 10, and 100 for each time scale
  - Library that contains top-level design unit
  - Top-level design unit
    - Entity/Architecture pair
    - Configuration
    - Module

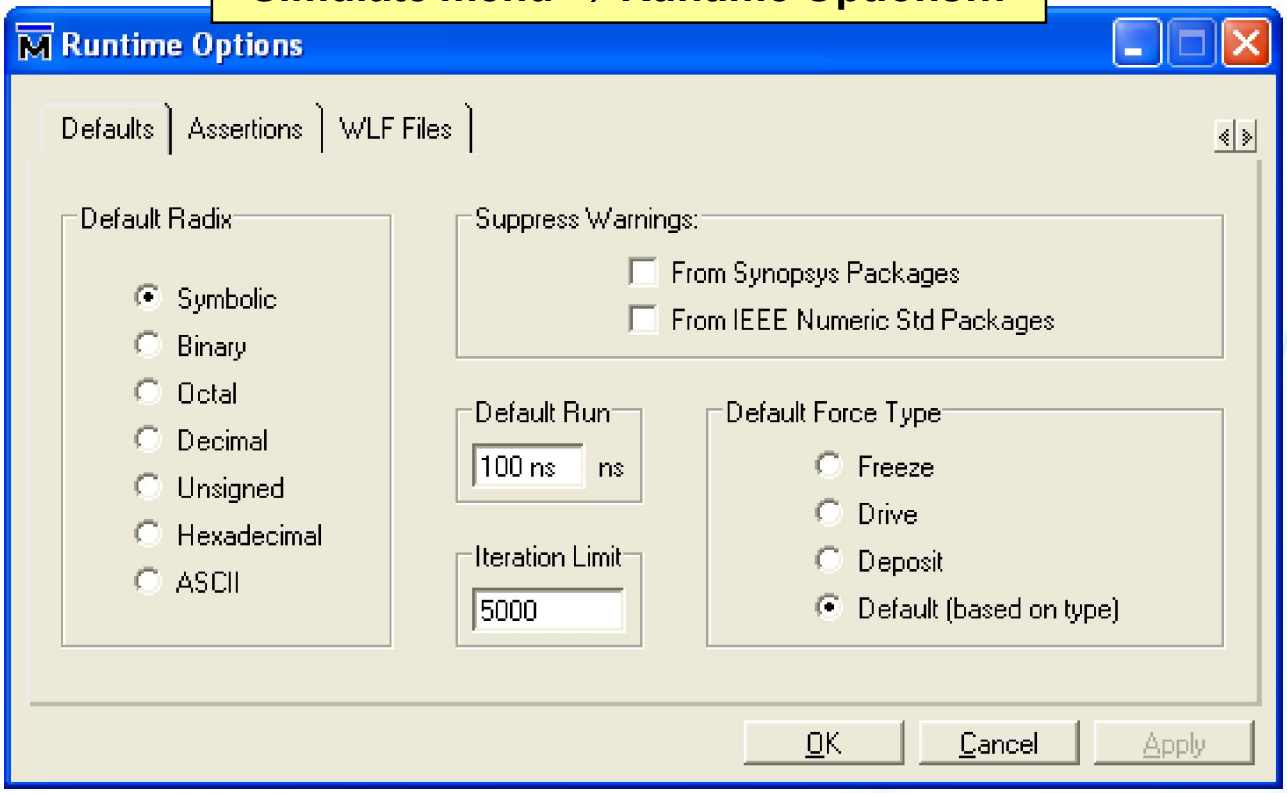


Command: **vsim** *<library_name>.<top_level_design_unit>*

Runtime Options

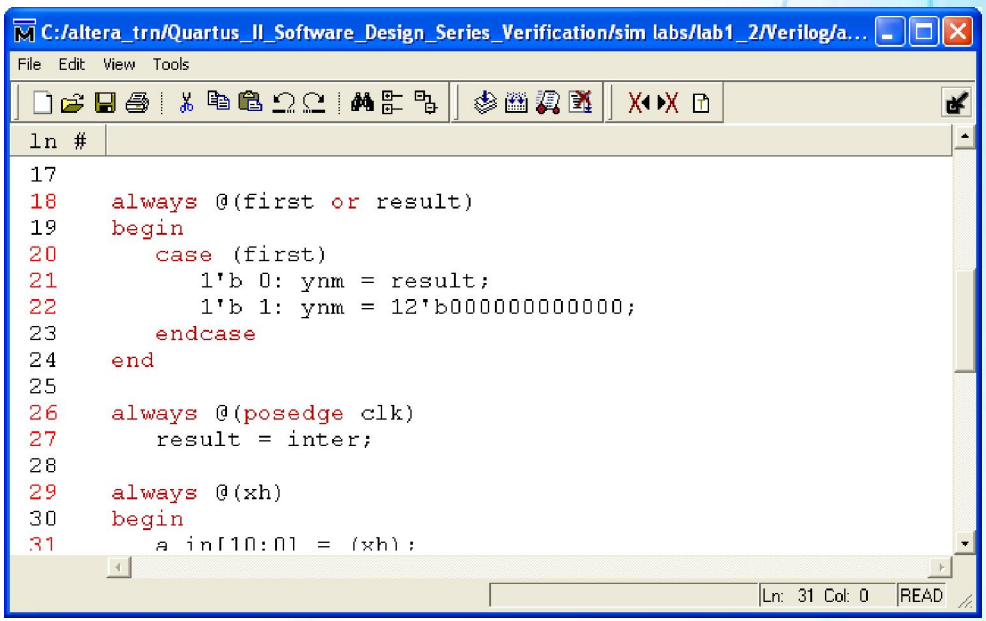
- Configure default settings for every new simulation

Simulate menu ⇒ Runtime Options…



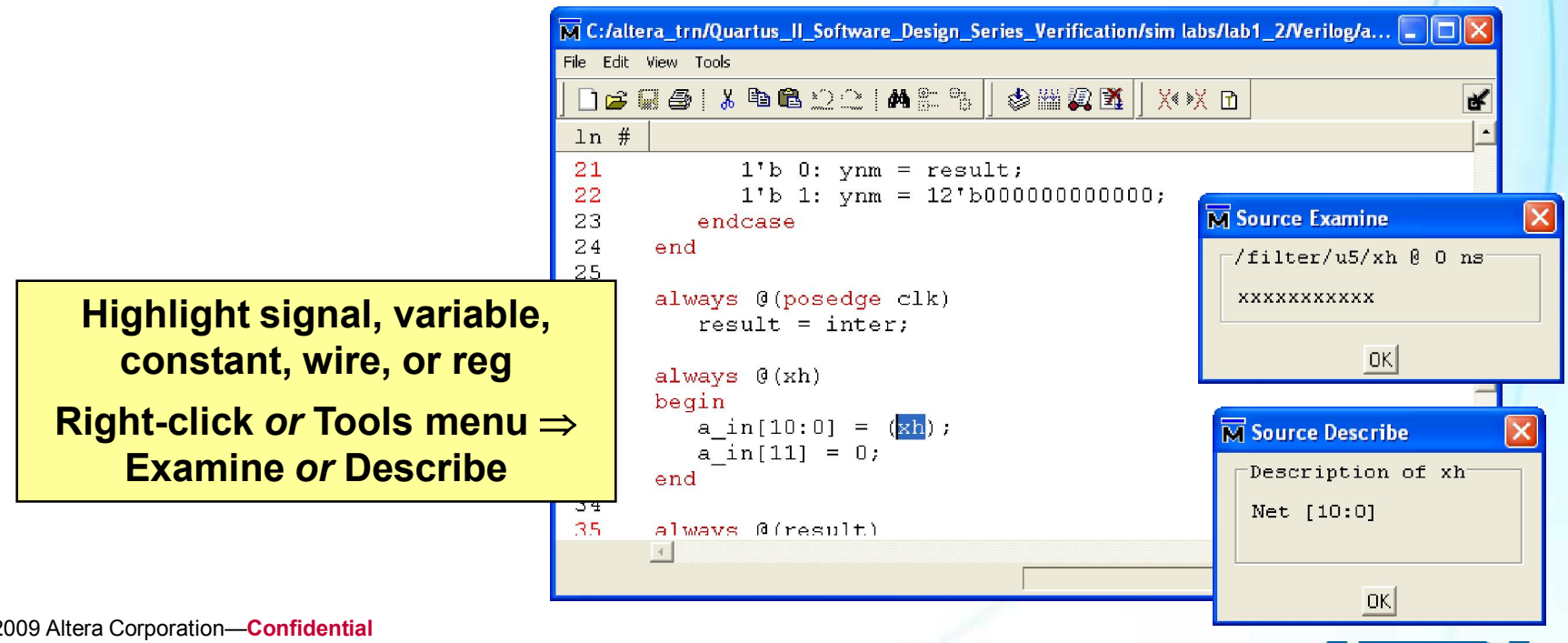
Source Window

- Double-click item in **sim** or Files tab of Workspace
- View or edit design entity source code
- Set breakpoints for simulation
- Color-coded
  - Comments, keywords, strings, numbers, executable lines, identifiers, system tasks, text
- Full edit capability
  - Save, compile, and restart sim
- Drag and drop
- Describe/examine
  - VHDL - signals, variables, constants, and generics
  - Verilog – nets, registers, and variables



Source Window

- Examine
  - Current simulation value of the selected HDL item
- Describe
  - General information about the selected HDL item

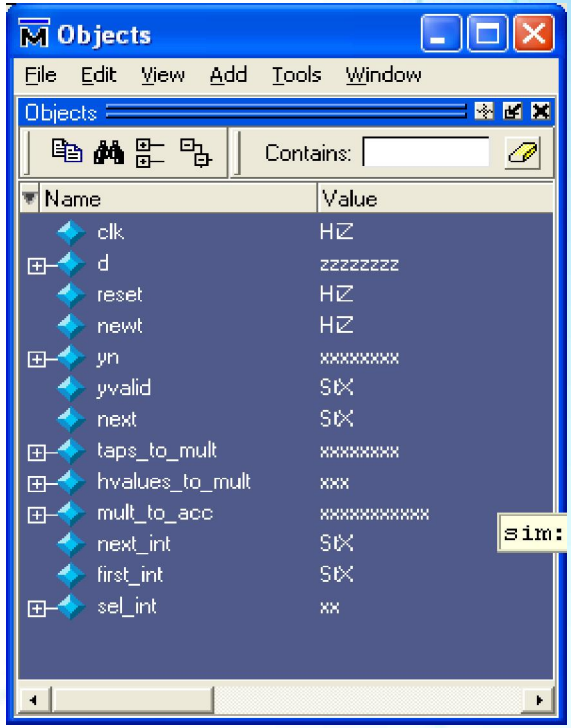


Highlight signal, variable, constant, wire, or reg

Right-click *or* Tools menu ⇒ Examine *or* Describe

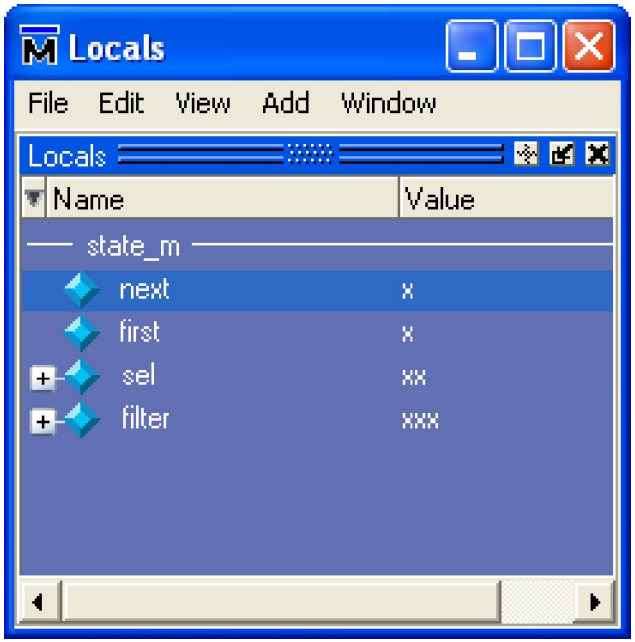
Objects Window

- Shows names and values of HDL items in selected region of **sim** tab in Workspace
- Sort by ascending, descending, or declaration order
- Verilog nets, register variables, named events, and module parameters
- Drag & drop to Wave & List windows
- Right-click object ⇒ **Force**
  - Apply stimulus
- **View** menu ⇒ **Filter**
  - Choose signal types for viewing
    - input, output, bidir, internal



Command: `view signals`

Locals Window

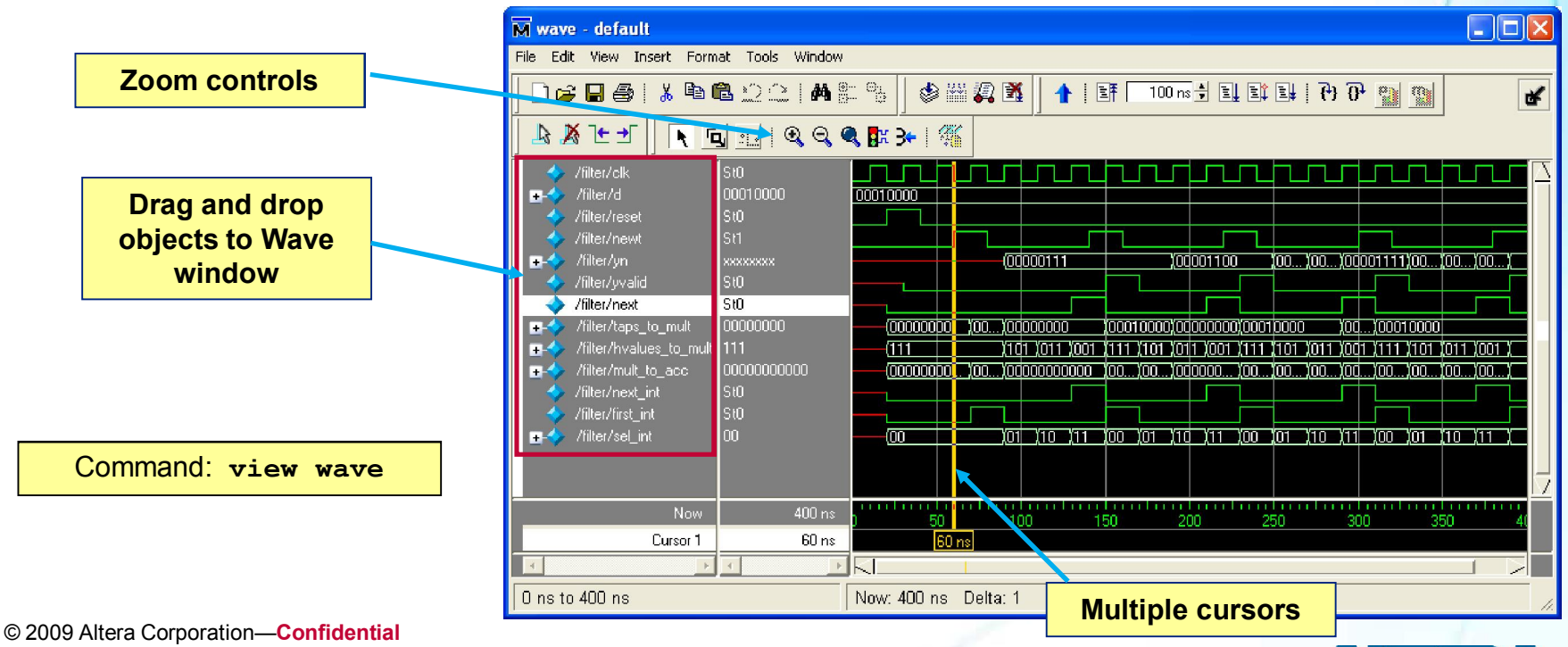


- Lists objects immediately visible from the next executed source code statement
- Contents will change from one statement to the next
- Value column lists the current value(s) associated with each name

Command: `view variables`

Wave Window

- Graphical history of simulation results in waveforms
  - Verilog - nets, register variables, named events, and module parameters
- Change radix of signals and vectors for easy readability
- Print waveforms

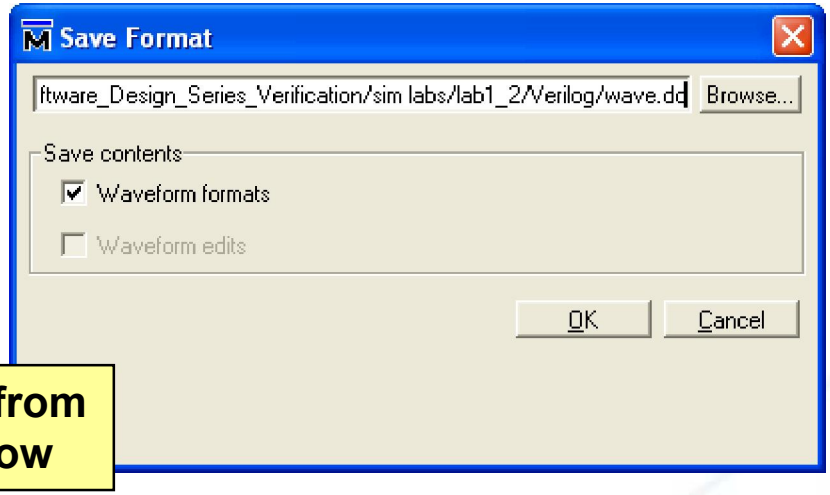


Wave Window: Adding Signals

- Move signals from Objects to Wave window to generate vectors
  - Drag and drop
  - With menu (**Add** ⇒ **Wave** ⇒ ***<type of signals to add>***)
    - Specific selected signals
    - Signals in a Workspace-selected region of the design
    - All design signals
- Vectors only generated for added signals

Wave Window: Save Format

- Saving the Wave window format allows for easy reloading of signals for next simulation session
- Format saved as a **.do** macro file
- Execute from Tools menu or command prompt to recreate Wave window format

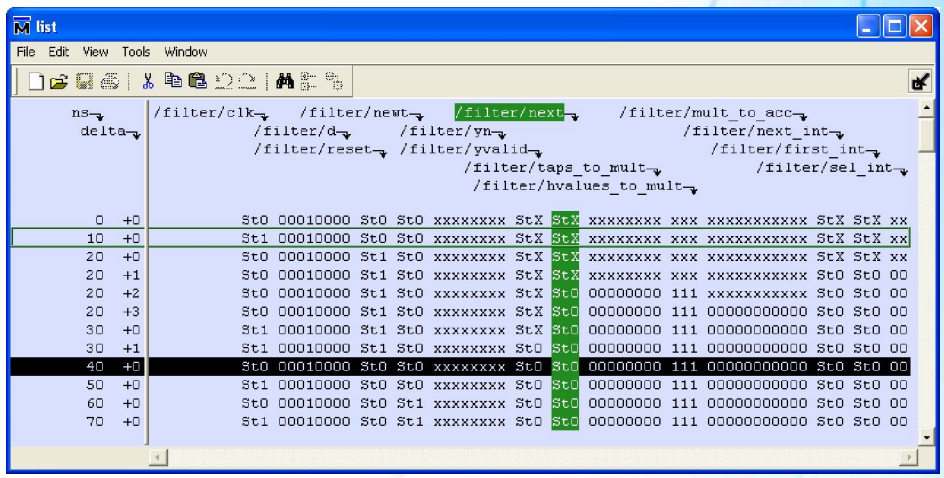


File ⇒ Save from Wave window

List Window

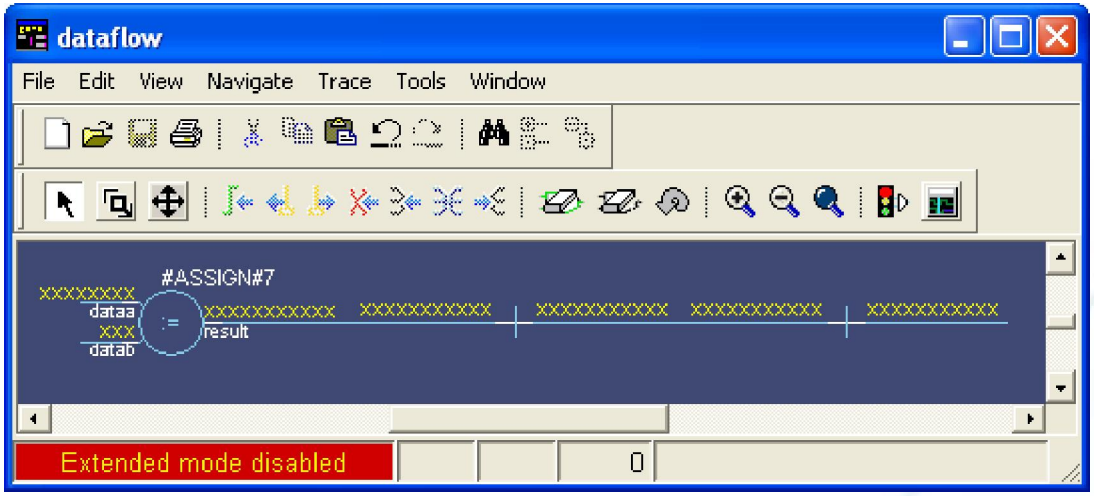
- Displays simulation results in tabular format
  - Verilog - nets and register variables
- Drag & drop objects to this window
- Edit functions
  - Find signals
  - Search for signal patterns
- Create user defined buses
  - **Tools** menu ⇒ **Combine Signals**
- Write List
  - Tabular, event, or TSSI
- Markers
  - Add, delete, or goto
- Set trigger and strobe properties

Command: **`view list`**



Dataflow Window

- Graphical tracing of VHDL signals or Verilog nets
  - Signals or nets in the center of window
    - Processes that drive signal or net on the left
    - Processes that read the signal or are triggered by the net on the right
    - Limited functionality in OEM version



Command: **`view dataflow`**

ModelSim Project Benefits

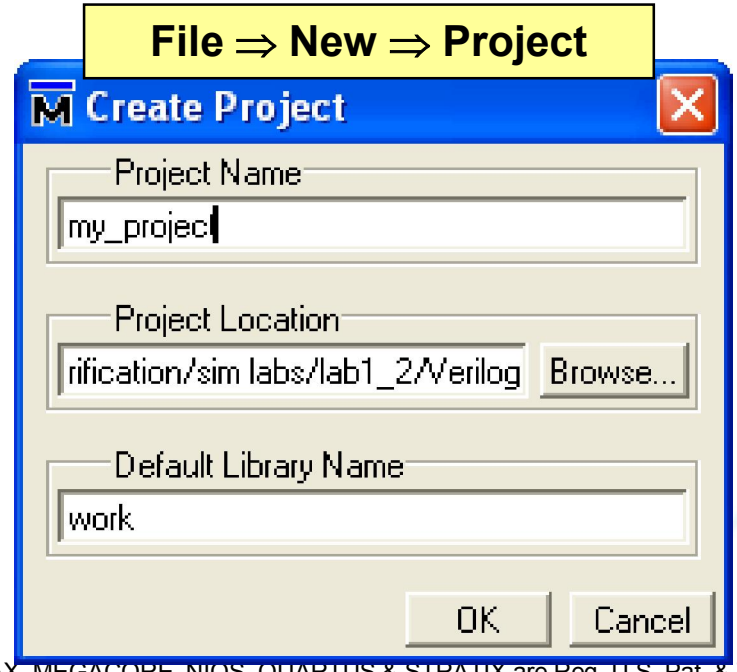
- Simplifies interaction with ModelSim
- Compile order of source files is maintained in project
- Compiler settings and switches are stored in project
- Allows for easy sharing of libraries (directories of compiled design units) without copying files to a local directory
- New project creates a work library for your design files automatically
- Project **.mpf** file contains all settings contained in main **modelsim.ini** file located in the ModelSim install directory

ModelSim Project Flow

1. Create a project (including **work** library)
2. Add design and testbench files to the project
3. Compile design files
4. Start simulator
5. Create a simulation configuration (optional)
6. Advance simulator

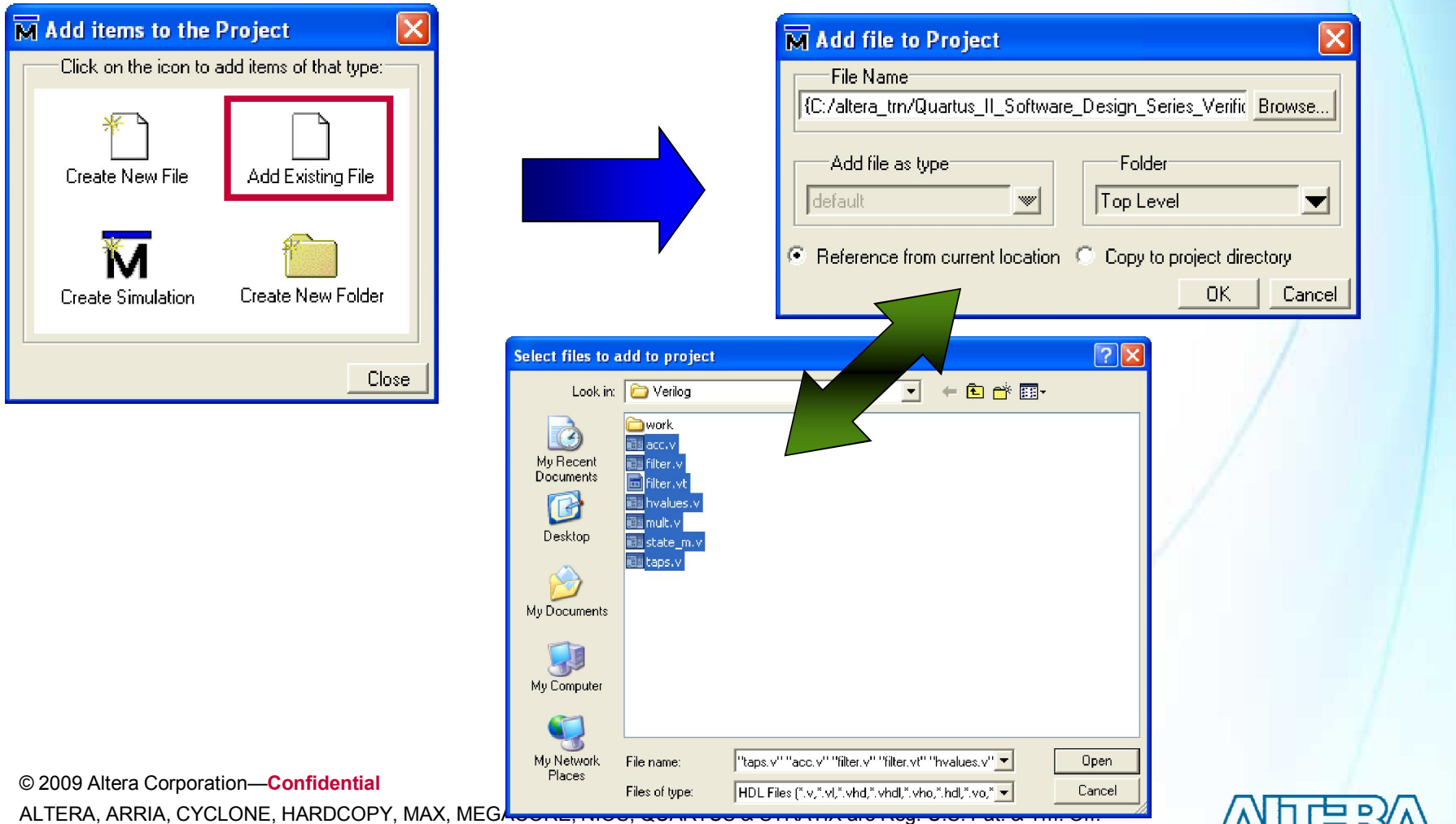
1) Create a Project

- Name project, set location, and name default library (default name is **work**)
- Creates metadata file (**.mpf**) in project location to store all project settings

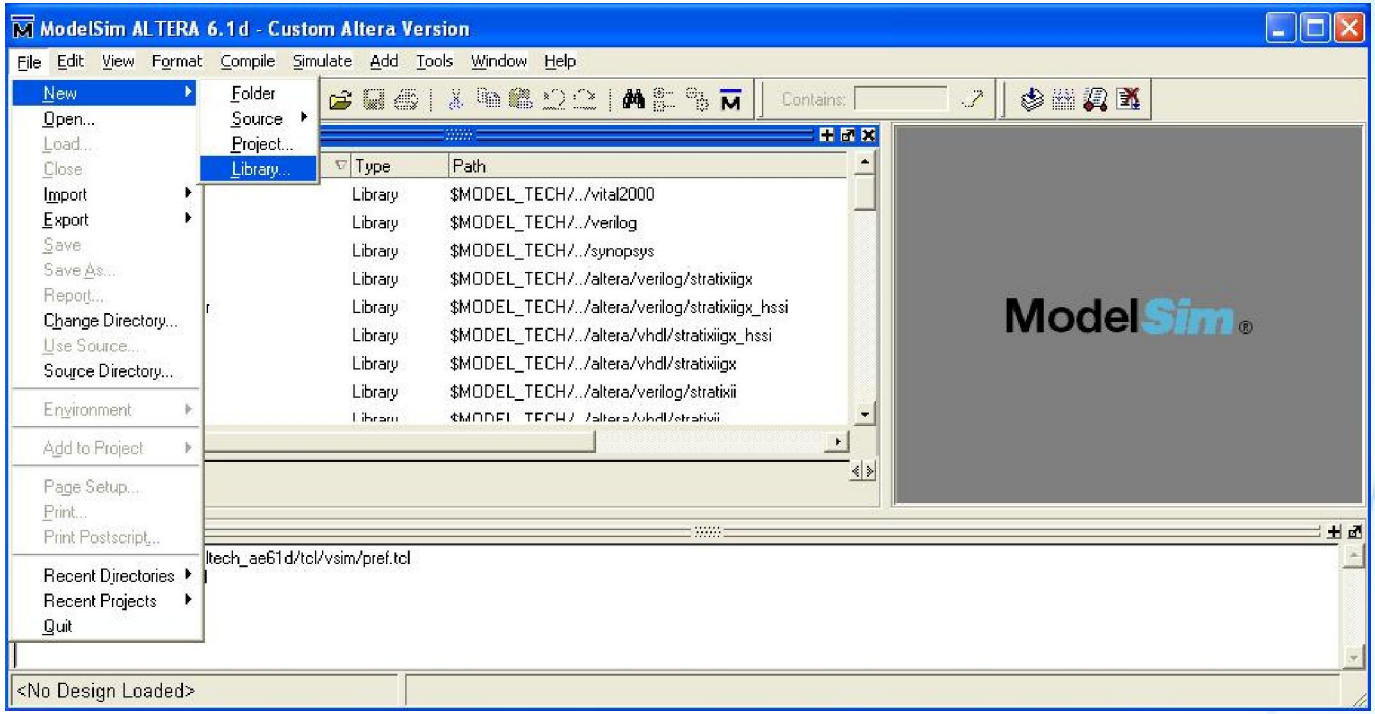


2) Add Files to the Project

- Add any existing design and testbench (HDL that defines simulation stimulus) files



Create ModelSim libraries (optional)

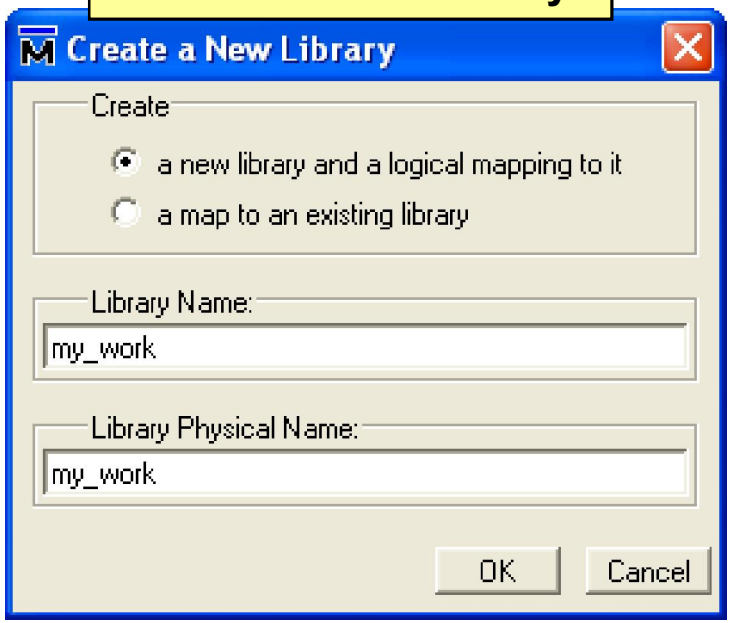


- From Main window
  - **File ⇒ New ⇒ Library…**
- Transcript command
  - `ModelSim> vlib <library name>`
  - Example: `vlib my_work`

Creating New Libraries

- Create a new library and the mapping to it for the project

File ⇒ New ⇒ Library

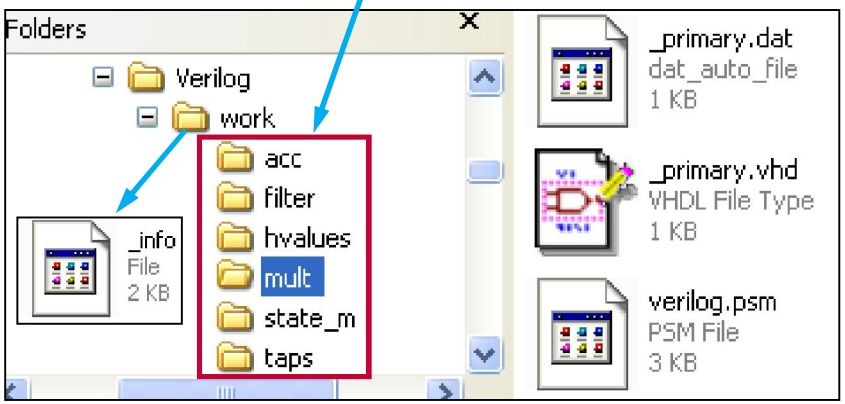



```
ModelSim> vlib my_work
ModelSim> vmap my_work my_work
```

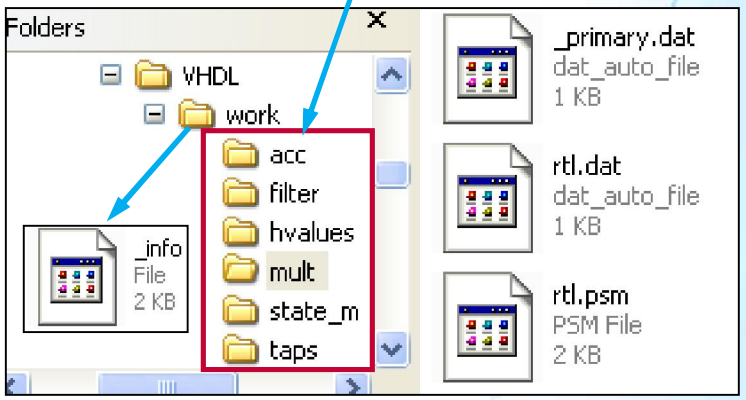
Creating New Libraries (cont.)

- Default is "**work**"
- New mapped libraries stored in project directory and appear in **Library** tab of Workspace

Verilog module names



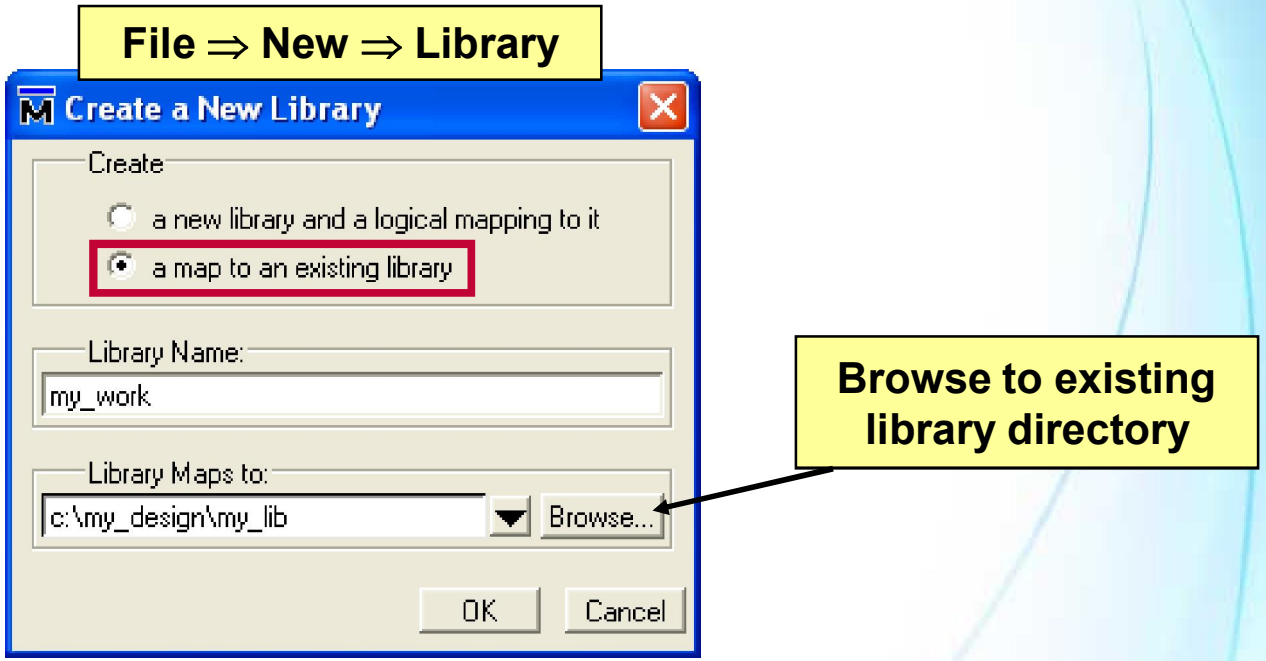
VHDL unit names



Where

_info: file created so ModelSim recognizes directory as a library
_primary.dat: encoded form of Verilog module or VHDL entity
_primary.vhd: VHDL entity representation of Verilog ports
***<arch_name>*.dat**: encoded form of VHDL architecture
verilog.psm and ***<arch_name>*.psm**: executable code files

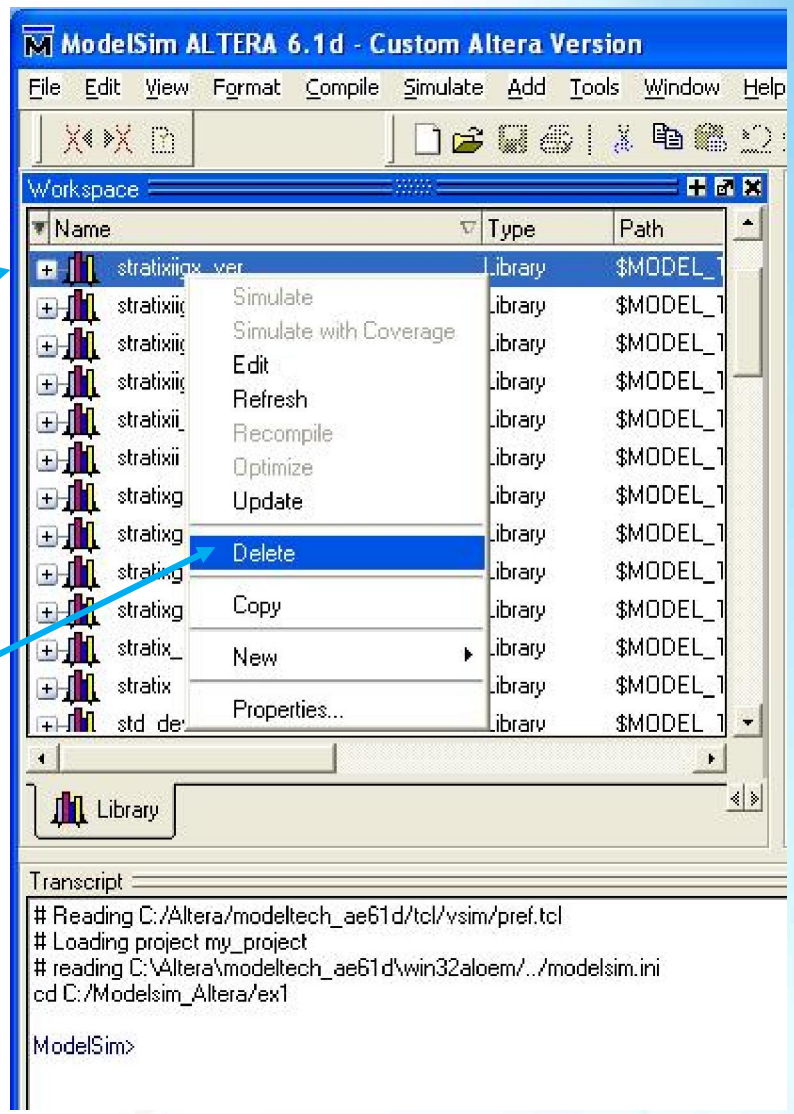
Map Existing Libraries to Physical Directories (optional)



- Transcript command
  - ModelSim> **vmap** *<logical_name> <directory_path>*
  - Example: **vmap my_work c:\my_design\mylib**

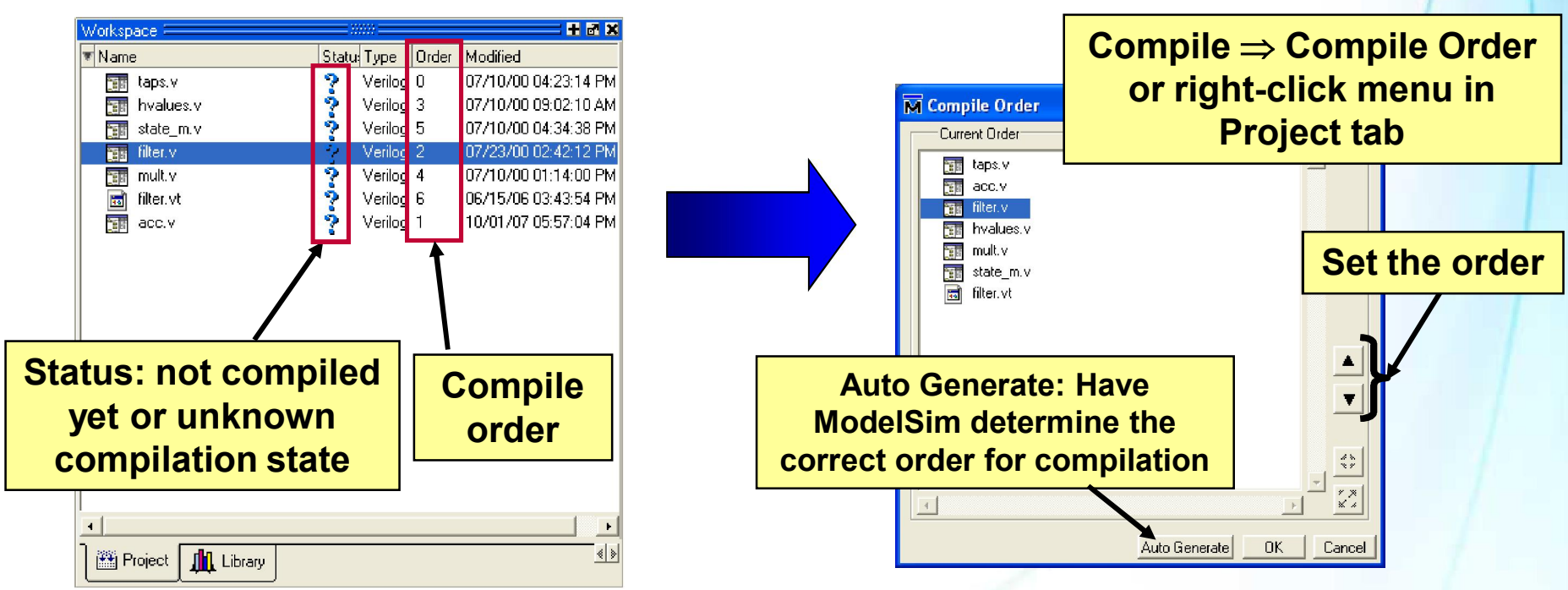
Other Library Commands

- **vdir**: Displays the contents of a specified library
  - GUI: click + to expand library
  - Transcript: **`vdir -lib`** *`<library_name>`*
- **vdel**: deletes an entire library or design unit
  - GUI: Right-click, select **Delete**
  - Transcript: **`vdel -lib`** *`<library_name>`* *`[<design_unit> ]`*



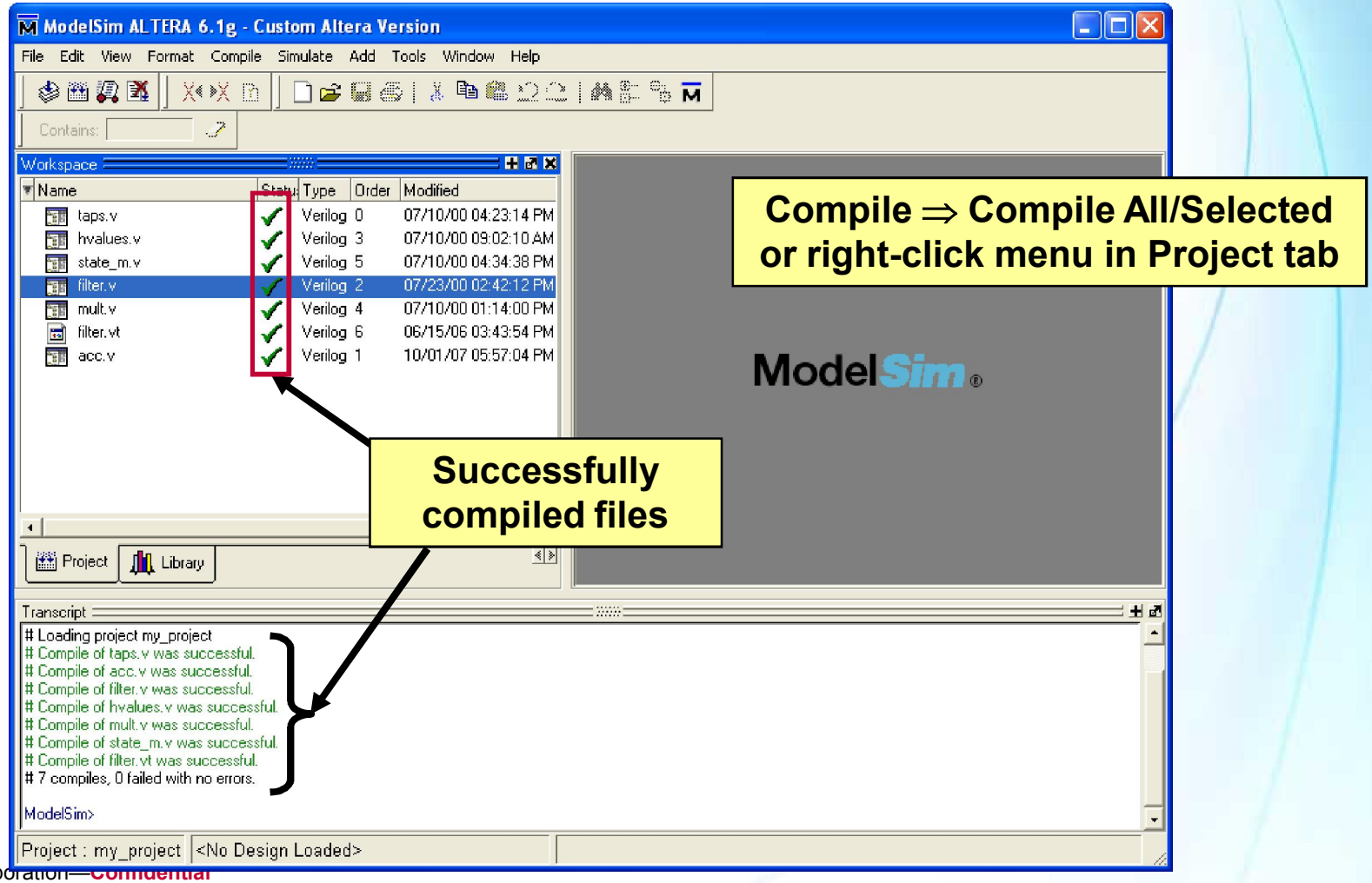
3) Compile Design Files

- Project tab in Workspace displays all design files
- Set compilation order and store with project
  - Required for VHDL dependencies; optional for Verilog
  - Useful in full version of ModelSim with mixed Verilog/VHDL



3) Compile Design Files (cont.)

- Compile all files or only selected ones

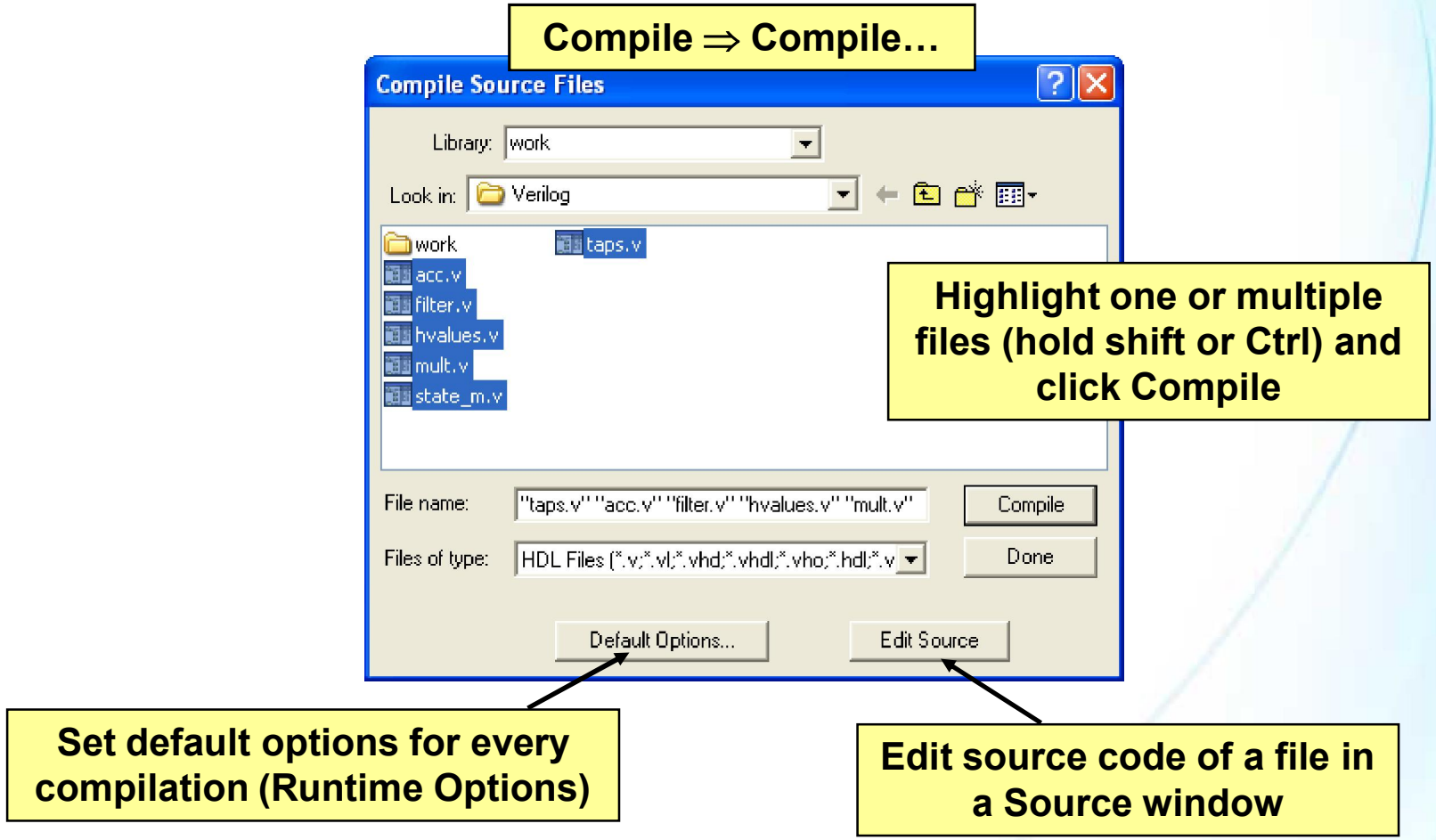


Manual File Compilation (Verilog)

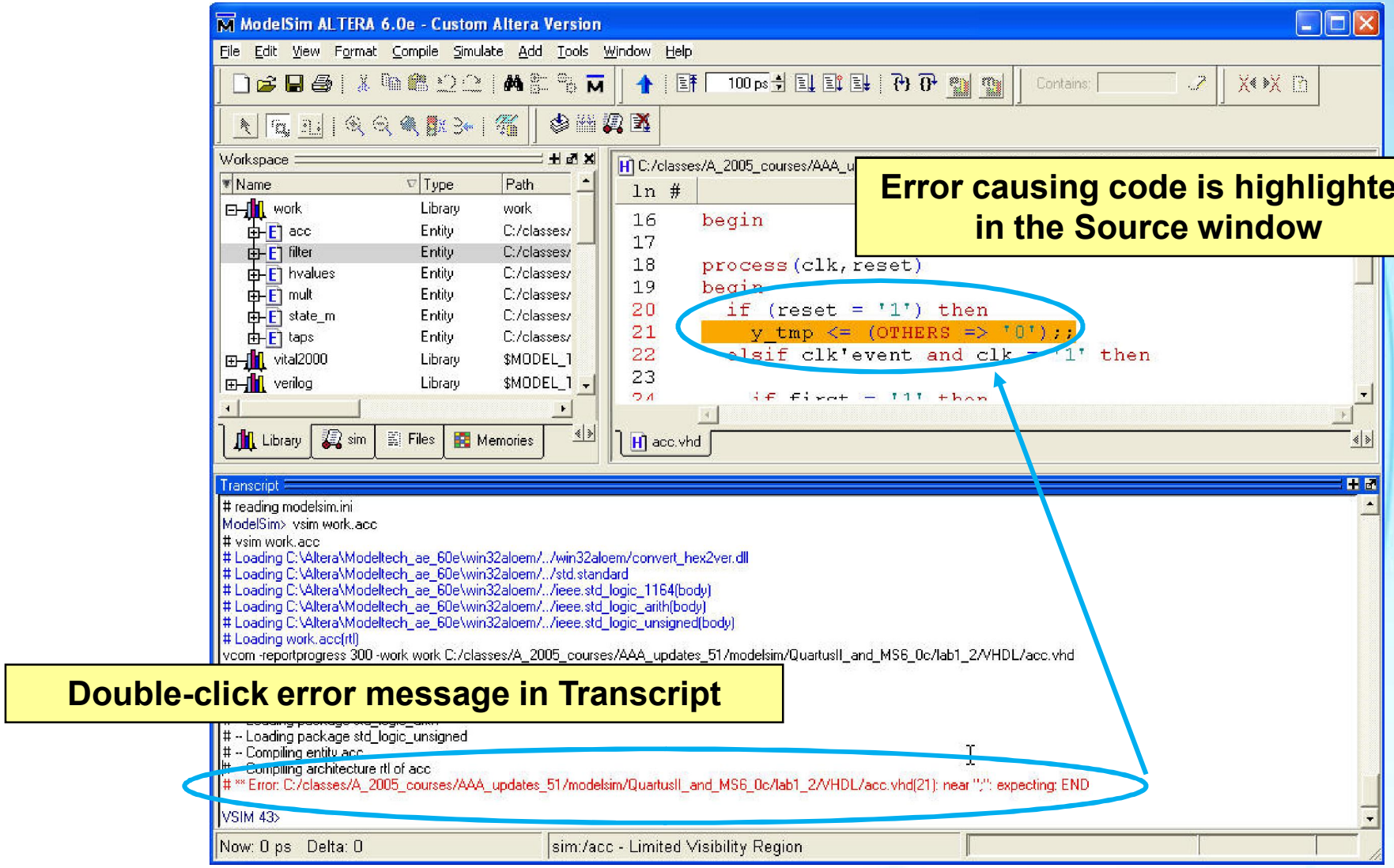
- GUI
  - **File ⇒ Change Directory** to locate target compilation library
  - **Compile ⇒ Compile…**
  - *Or* set target library for compilation from file **Properties** (right-click)
- Transcript
  - **`vlog -work`** *`<library_name>`* *`<file1>`***`.v`** *`<file2>`***`.v`**
  - Files are compiled in the order they are listed
  - Order of files or compilation does not matter
- Default compiles into library **work**
  - Example: **`vlog my_design.v`**

Note: Design units must be re-compiled when the design units they reference are changed in the library

Manually Compiling Files in GUI

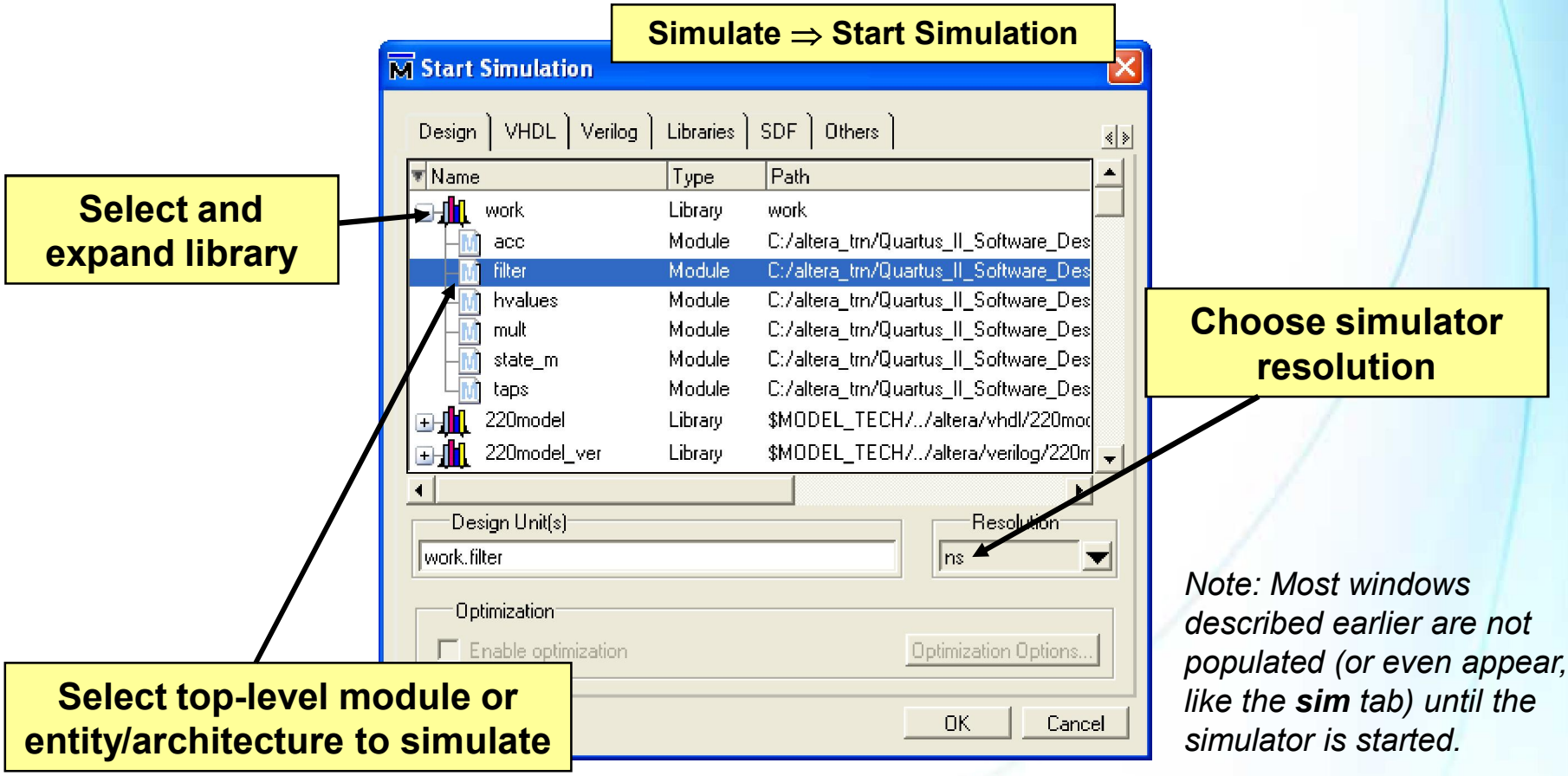


Error Messages



4) Start Simulator (GUI)

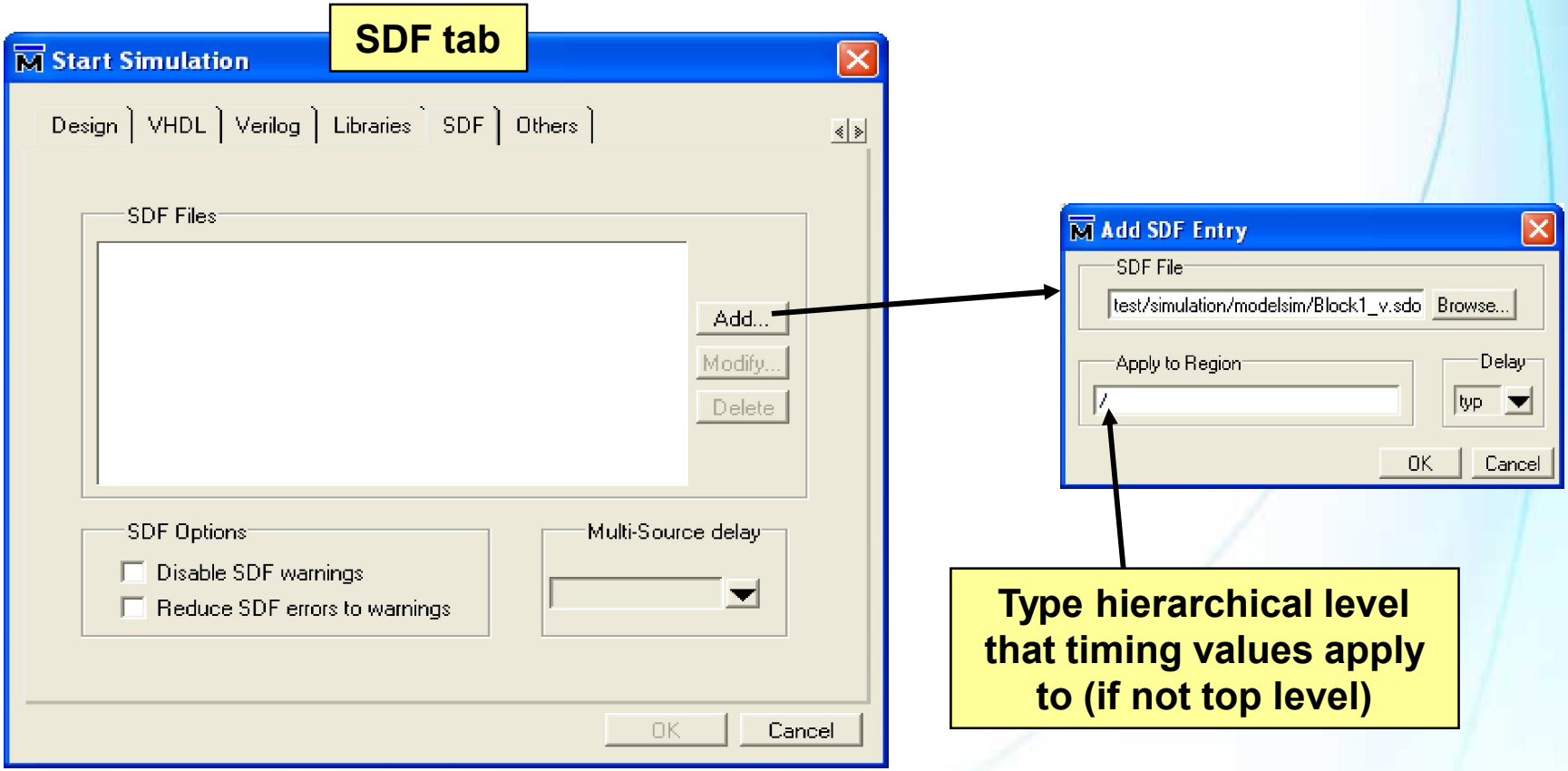
- Select options and start the simulation



*Note: Most windows described earlier are not populated (or even appear, like the **sim** tab) until the simulator is started.*

4) Start Simulator (GUI) (cont.)

- For gate-level, post-fit (timing) simulation, add **SDF** file(s)
  - See on-line help for more information about post-processing simulations



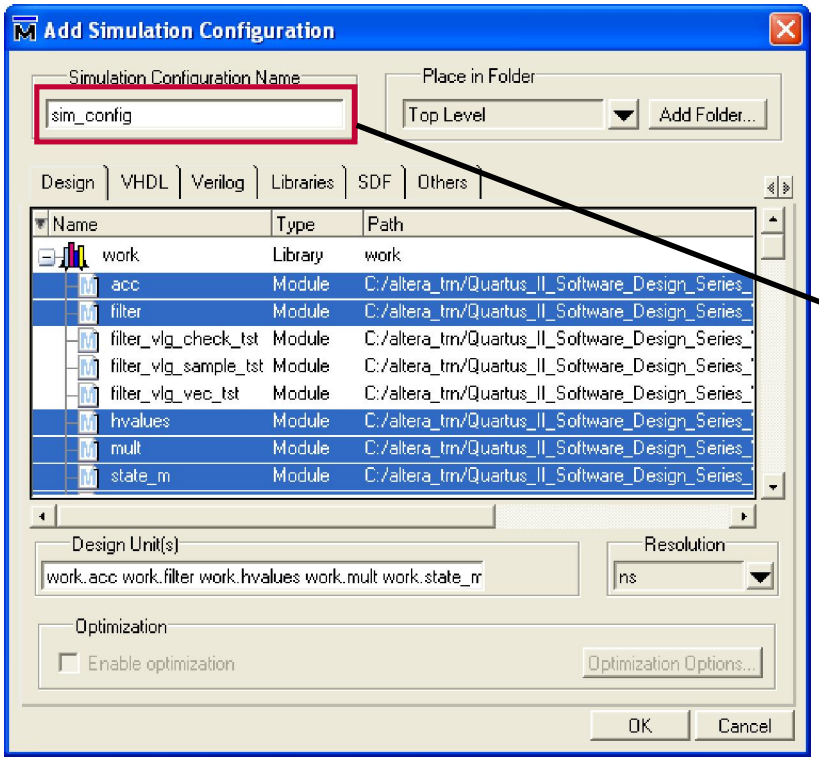
4) Start Simulator (Transcript)

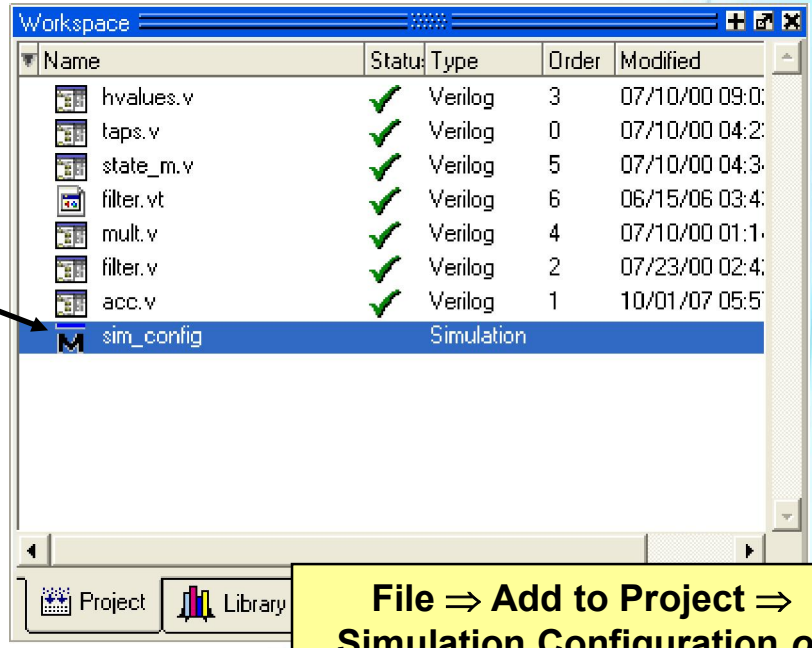
- Transcript: **`vsim`** *`<top_level_design_unit>`*

- Verilog
  - **`vsim`** *`<top_level1> <top_level2>`*
    - Simulates multiple top-level modules

5) Create a Simulation Configuration (optional)

- Simulation configuration holds all settings and simulation options
- Double-click to start simulation (instead of step 6)





File ⇒ Add to Project ⇒ Simulation Configuration *or* right-click menu in Project tab

6) Advance Simulator (GUI)

- **Simulate ⇒ Run ⇒ <*run_option*>**
  - Advance simulation by selected number of time steps
  - Continue running a paused simulation
- **Simulate ⇒ Run ⇒ Restart**
  - Select simulation result data to keep for next run
  - Reload any edited design elements
  - Reset simulation time to 0
  - Transcript: **`restart`**

6) Advance Simulator (Transcript)

- Transcript: **`run`** *`<time_steps> <time_units>`*
  - Advances the simulator in the amount of timesteps specified

- Options
  - `<time_steps> <time_unit>`
    - Specifies the number and unit of timesteps to run
    - Units can be {fs, ps, ns, ms, sec}
  - **`-step`**
    - Steps to the next HDL statement
  - **`-over`**
    - Steps to the next HDL statement
    - Treats Verilog tasks as single executable

6) Advance Simulator (Transcript) (cont.)

- **`-continue`**
  - Continues the last simulation after a **`-step`**, **`-over`**, or breakpoint
- **`-all`**
  - Runs simulator until no more events are scheduled

Command Examples (*run*)

- **`run 1000`**
  - Advances the simulator 1000 timesteps from the current position
- **`run 2500 ns`**
  - Advances the simulator the number of timesteps corresponding to 2500 ns from the current position
- **`run @3000`**
  - Advances the simulator to timestep 3000 from the current position
- **`run @7000 ns`**
  - Advances the simulator to 7000 ns from the current position
- **`restart`** (or **`restart -force`**)
  - Resets the simulator to time 0

Pause a Running Simulation

- Keyboard break key
- Toolbar button 
- Stop (pause) simulator while advancing

Simulator Stimulus

1. Force commands
   - Simple module simulation
   - Directly from command console
2. Testbench (user-generated)
   - Verilog or VHDL
   - Very complex simulation interactive simulation

1) *force* Command

- Apply stimulus directly to Verilog nets
- General syntax:
  - **force** *<item_name> <value> <time>, <value> <time>*
- Arguments
  - *<item_name>* (Required argument)
    - Name of the HDL item to be forced
    - '/' indicates hierarchical levels
    - Must be a scalar or one-dimensional array of characters
      - Can be an indexed array, array slice, or record sub-element as long as its of the above type
    - Can use wildcards as long as only one match is obtained

force Command (cont.)

- *<value>* (Required argument)
  - Value to which the item is forced
  - Must fit item's data type

Value	Description
1111	character sequence
2#1111	binary radix
10#15	decimal radix
16#F	hexadecimal radix

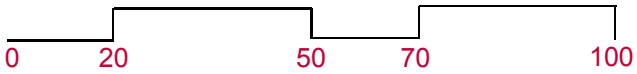
- *<time>*
  - Specifies the time unit for the value
  - Relative to current simulation time
    - Use @ character to specify absolute time
  - Time units can be specified
    - Default is simulation resolution units
  - Optional

force Command (cont.)

- **-r**[epeat] *<period>*
  - Repeats the force command for the specified period
  - Optional
- **-cancel** *<period>*
  - Cancels the force command after the specified period
  - Optional

force Command Examples

- **`force /clr 0`**
  - Forces clr to 0 at current simulation time
- **`force /bus1 01XZ 100 ns`**
  - Forces **bus1** to 01XZ at 100 nanoseconds after current simulation time
- **`force /bus2 16#4F @200`**
  - Forces **bus2** to 4F at 200 time units after simulation startup in the resolution chosen at simulation startup
- **`force /clk 0 0, 1 20 -repeat 50 -cancel 1000`**



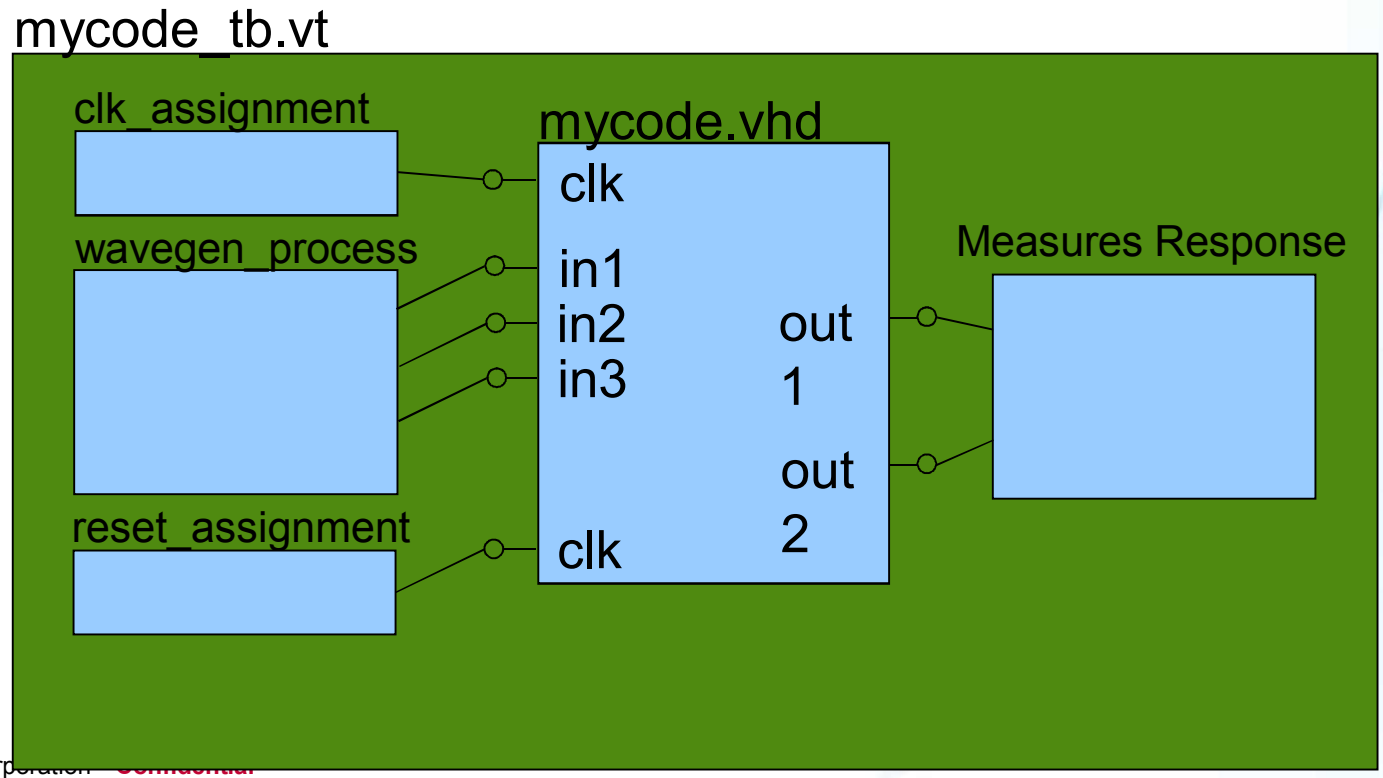
  - Forces **clk** to 0 at 0 time units and 1 at 20 time units after the current simulation time. This repeats every 50 time units until time unit 1000. Thus, the next 0 occurs at 50 and the next 1 occurs at 70.
- **`force /clk2 1 10 ns, 0 {20 ns} -r 100 ns`**
  - Similar to the previous example. Time unit expressions that precede the -r must be placed in curly braces. Pattern repeats every 100 ns.

2) Testbench

- Top-level code in the hierarchy that describes simulation parameters
  - Connects inputs and outputs
  - Provides stimulus
  - Measures response to the stimulus
  - Verilog (**.vt**)

Self-Verification Methods

- Add a process or at least the functions to an existing process so that the outputs can be monitored



Sample Verilog Testbench (.vt File)

```
module gatetest();

parameter period = 40;
reg A,B;
wire Y;

NAND1 u1(.A(A), .B(B), .Y(Y));

initial
begin
            A = 1;
            B = 1;
            #period;
            A = 1;
            B = 0;
            #period;
            A = 0;
            B = 1;
            #period;
            A = 0;
            B = 0;
            #period;
end
endmodule
```

Top-level entity has no ports

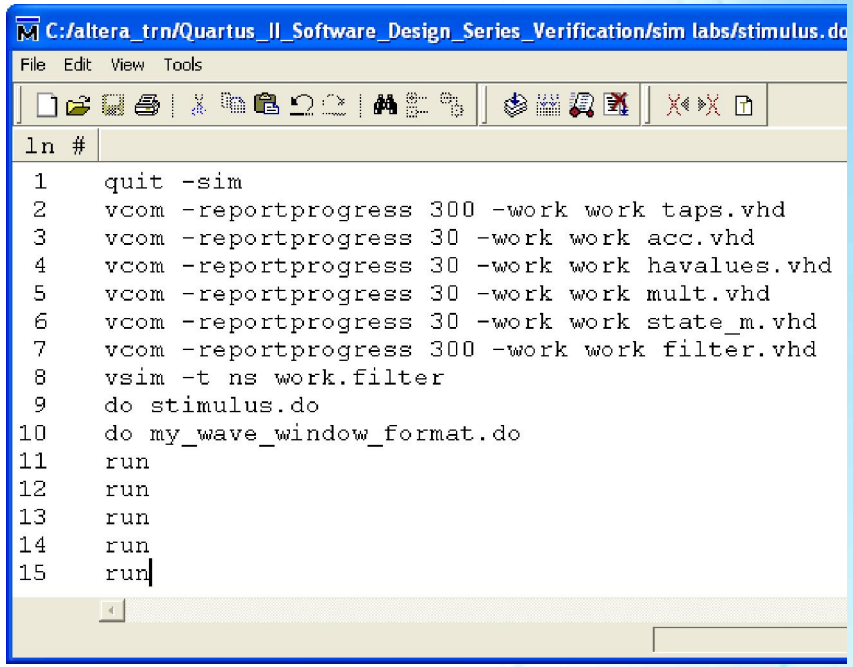
Signals to assign values & observe

Instantiate lower-level entity

Apply stimulus

DO Files

- Macro file for automating simulation steps
  - Library setup
  - Compiling
  - Simulating
  - Forcing simulator stimulus
- Can be invoked in all ModelSim modes
  - GUI: **Tools ⇒ Execute Macro…**
  - Transcript: **do** *<filename>*.**do**
- Can call other DO files



Example DO Files

my_sim.do

```
cd c:\mydir
vlib work
vcom counter.vhd
vsim counter
view *
do stimulus.do
```

stimulus.do

```
add wave /clk
add wave /clr
add wave /load
add wave -hex /data
add wave /q
force /clk 0 0, 1 50 -repeat 100
force /clr 0 0, 1 100
run 500
force /load 1 0, 0 100
force /data 16#A5 0
force /clk 0 0, 1 50 -repeat 100
run 1000
```

my_sim.do calls a second .do file named "stimulus.do"

Breakpoints

- Two types of breakpoints
  - Breakpoints on line(s) in source code window
    - Toggles: click line number; click again to delete existing breakpoint
    - No limit to the number of break points
    - Use command **bp**
      - **bp** *<file_name> <line#>*
  - Conditional break points
    - **when** *<condition> <action>*
    - **when {b=1 and c/=0} {stop}**
    - Used with VHDL signals and Verilog nets and registers
    - Use command **bp** also
      - **bp <file_name> <line#> {if{$now/=100}then{cont}}**
      - When breakpoint reached, if current time is 100, break; otherwise continue

startup.do File

- DO script automatically executed by **`vsim`** upon startup
- Example **startup.do** file might look like this:

```
view source
view structure
view wave
do wave.do
```

- To invoke a startup file, uncomment (remove the ";" from) the following line in the **modelsim.ini** file and provide path to the DO file:

```
;Startup = do /<path_to_startup>/startup.do
```

Exercise 1

Please go to Exercise 1

Introduction to Verilog

Verilog Overview

What is Verilog?

- IEEE industry standard Hardware Description Language (HDL) - used to describe a digital system
- Use in both hardware simulation & synthesis

Verilog History

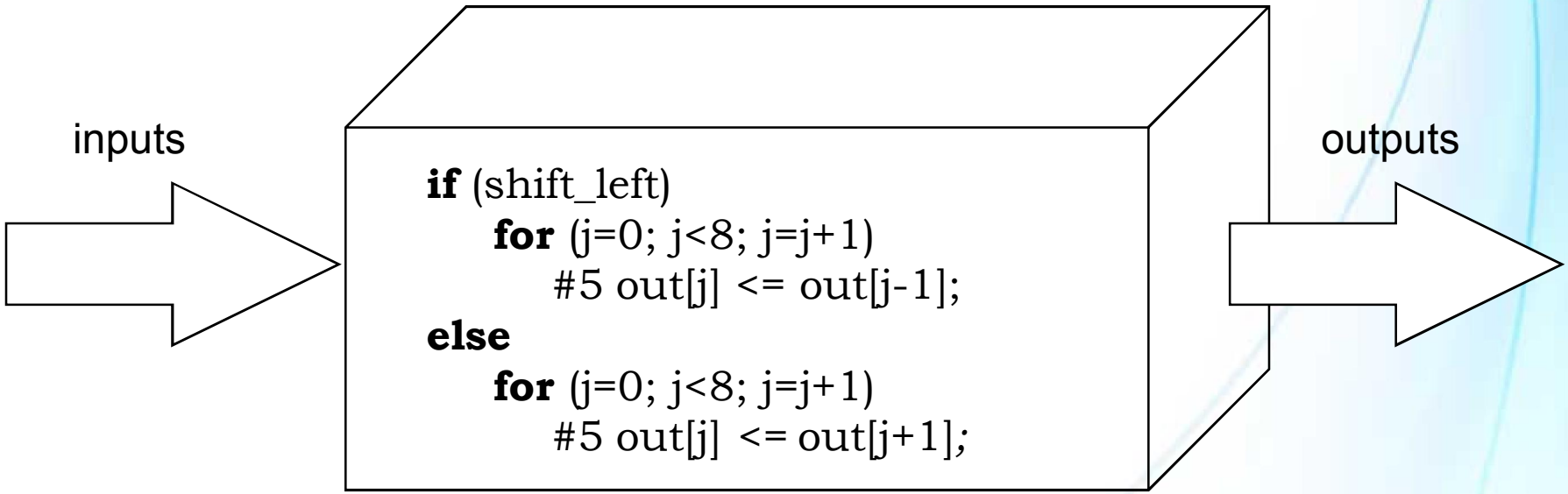
- Introduced in 1984 by Gateway Design Automation
- 1989 Cadence purchased Gateway (Verilog-XL simulator)
- 1990 Cadence released Verilog to the public
- **O**pen **V**erilog **I**nternational (OVI) was formed to control the language specifications
- 1993 OVI released version 2.0
- 1995 IEEE accepted OVI Verilog as a standard, Verilog 1364
- 2001 IEEE revised standard
- 2005 IEEE accepted new revision for the standard

Verilog HDL Terminology

- HDL: A software programming language that is used to model a piece of hardware
- Behavior Modeling: A component is described by its input/output response
- Structural Modeling: A component is described by interconnecting lower-level components/primitives

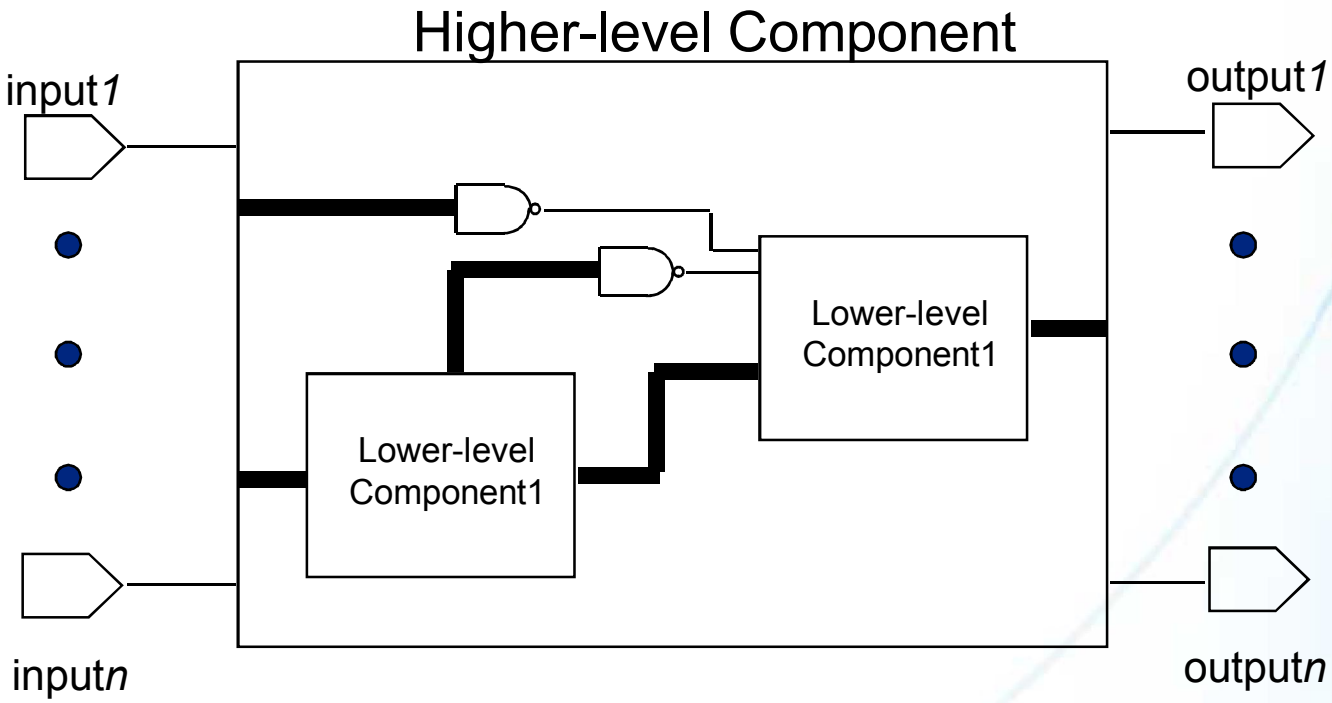
Behavior Modeling

- Only the functionality of the circuit, no structure
- Synthesis tool creates correct logic



Structural Modeling

- Functionality and structure of the circuit
- Call out the specific hardware



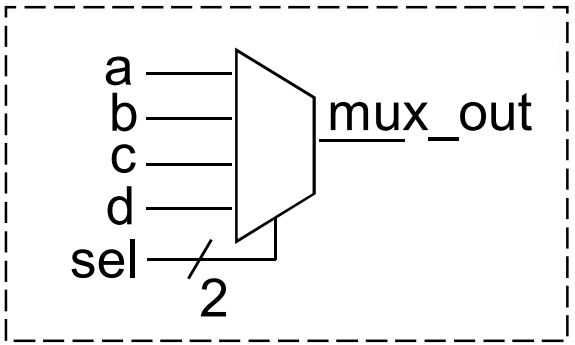
More Terminology

- 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

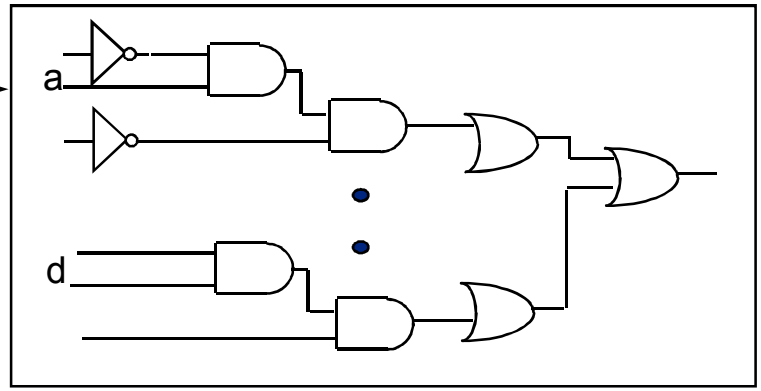
RTL Synthesis

```
always @(a, b, c, d, sel)
    case (sel)
        2'b00: mux_out = a;
        2b'01: mux_out = b;
        2b'10: mux_out = c;
        2'b11: mux_out = d;
    endcase
```

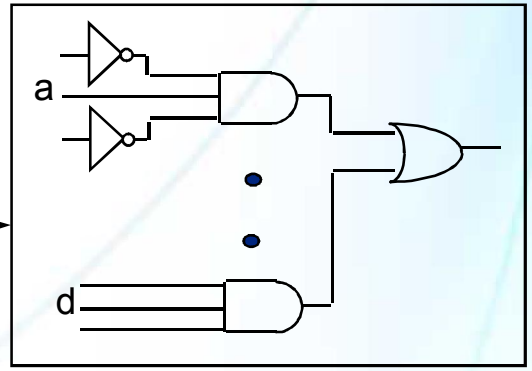
inferred



Translation



Optimization



Verilog vs. Other HDL Standards

- Verilog
  - "Tell me how your circuit should behave and I will give you the hardware that does the job."
- VHDL
  - Similar to Verilog
- ABEL, PALASM, AHDL
  - "Tell me what hardware you want and I will give it to you"

Verilog vs. Other HDL Standards (cont.)

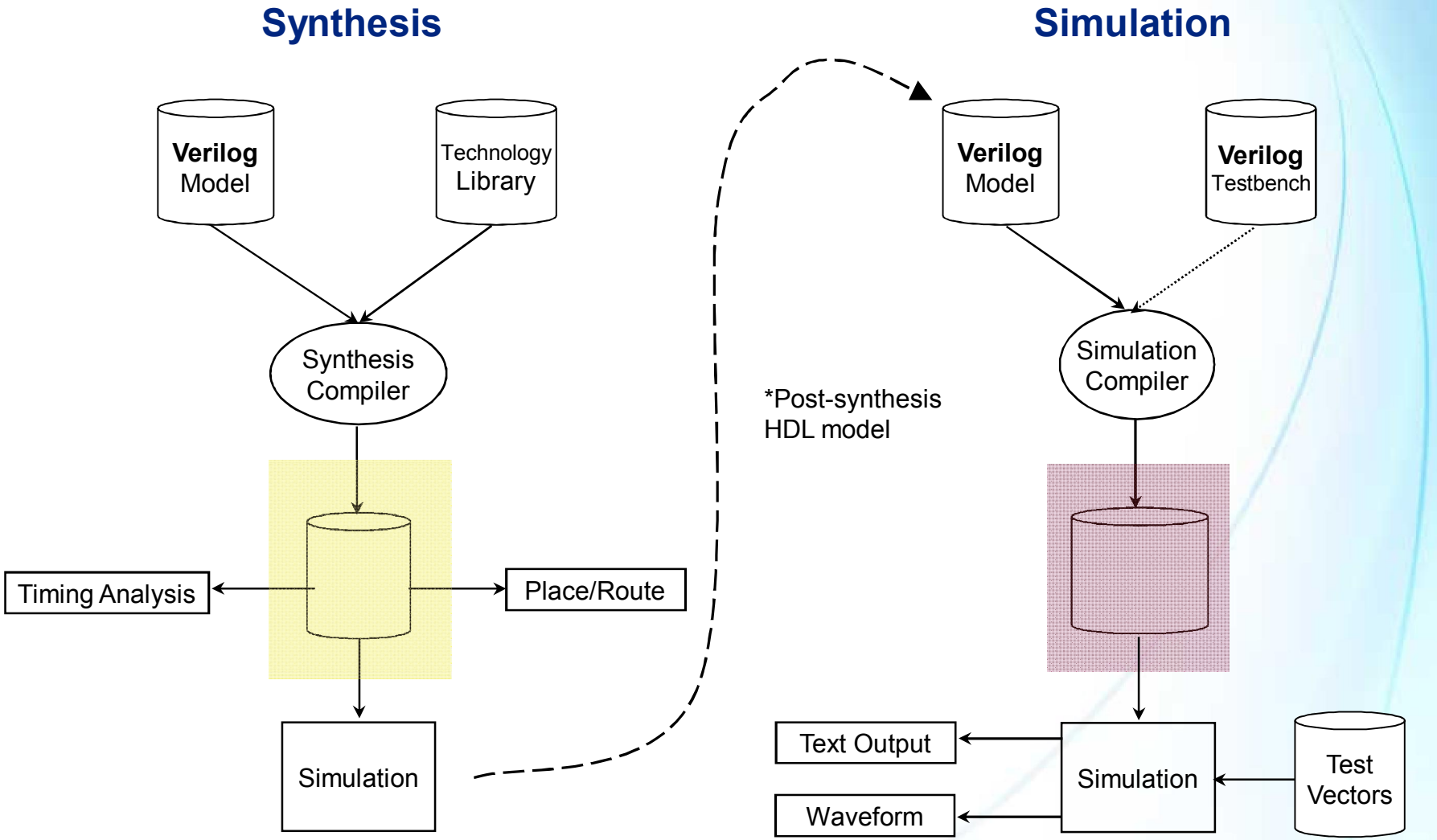
- Verilog
  - **always** @ (**posedge** clk)

    q<=d;
  - Result: Verilog Synthesis provides a positive edge-triggered flip-flop
- ABEL, PALASM, AHDL
  - **DFF**(d,q,clk)
  - Result: ABEL, PALASM, AHDL synthesis provides a D-type flip-flop. The sense of the clock depends on the synthesis tool

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



Introduction to Verilog

Verilog Modeling

Verilog - Basic Modeling Structure

module *module_name (port_list);*

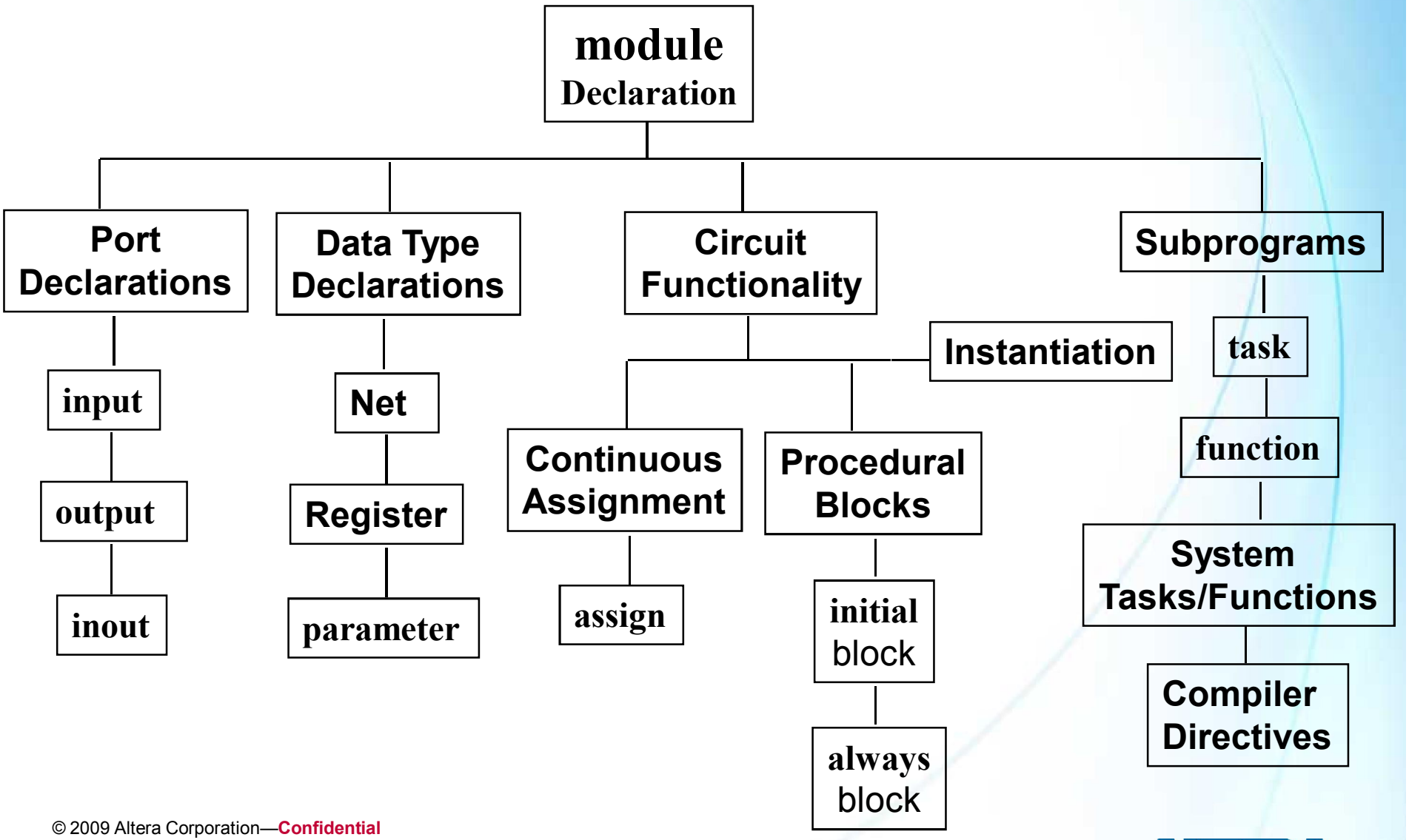
- *port declarations*
- *data type declarations*
- *circuit functionality*
- *timing specifications*

endmodule

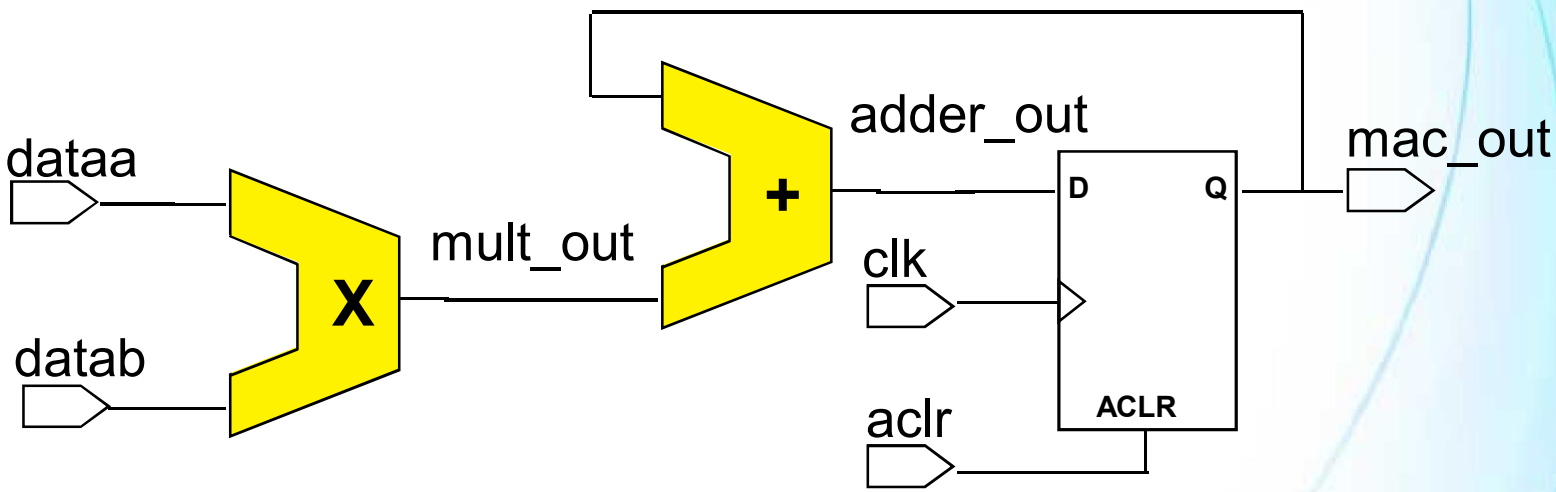
- Begins with keyword **module** & ends with keyword **endmodule**
- Case-sensitve
- All keywords are lowercase
- Whitespace is used for readability
- Semicolon is the statement terminator
- *//* : Single line comment
- */* */* : Multi-line comment
- Timing specification is for simulation (not discussed)

Example 1

Components of a Verilog Module



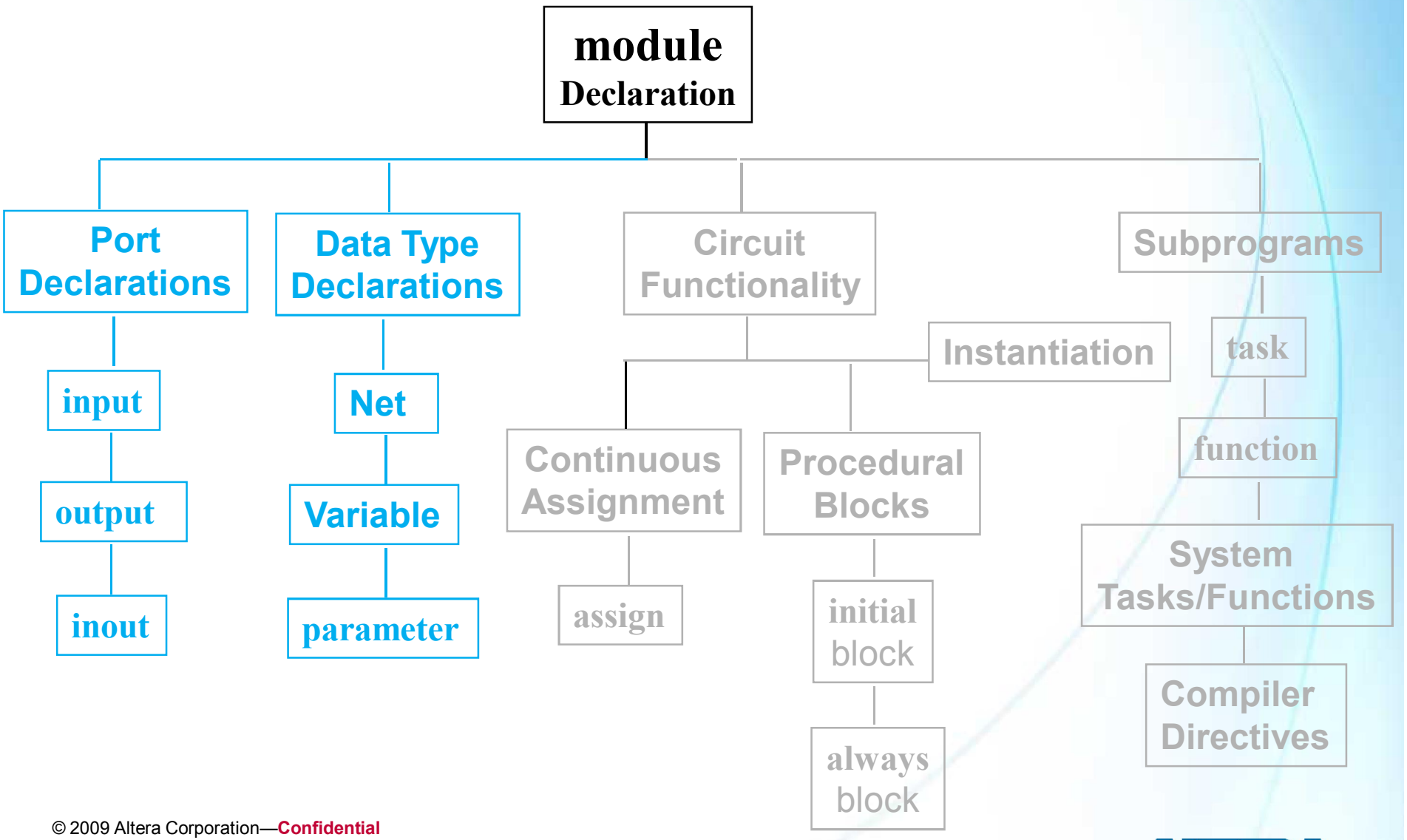
Schematic Representation - MAC



Verilog Model: Multiplier-Accumulator (MAC)

```
`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

      multa #(.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** ⇒ input port
  - **output** ⇒ output port
  - **inout** ⇒ bidirectional port

Verilog '2001 & 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

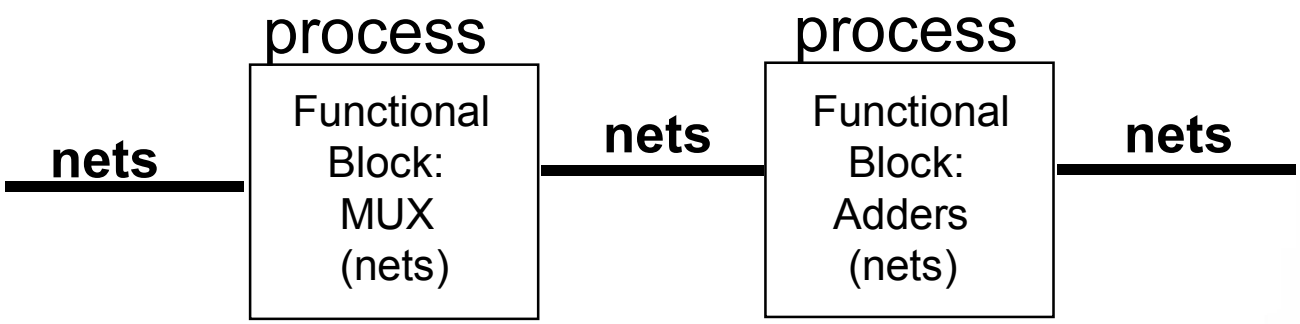
```
module mult_acc (
   input [7:0] dataa, datab,
   input clk, aclr,
   output [15:0] mac_out
);
```

MAC (Module & Port Declarations)

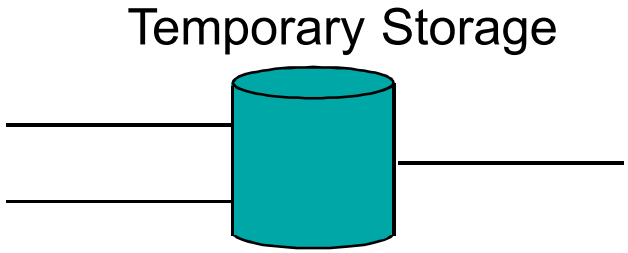
```
`timescale 1 ns/ 10 ps

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

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

endmodule
```

Data Types

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



- **Variable data type** - represents element to store data temporarily



Net Data Type & Net Arrays

- **wire** ⇒ represents a node
- **tri** ⇒ represents a tri-state node

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

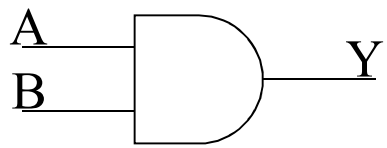
- Examples:
  - **wire** <*signal name*> ;
  - **wire** [15:0] mult_out, adder_out;

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
trireg	Stores last value when not driven	N

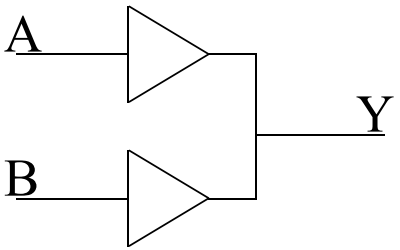
*Note: There is no functional difference between **wire** & **tri**; **wand** & **triand**; **wor** & **trior***

Net Data Types




```
wire Y;  // declaration
assign Y = A & B;
```

Y \ A	0	1
0	0	0
1	0	1

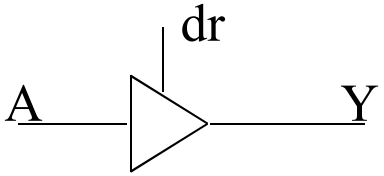



```
wand Y;  // declaration
assign Y = A;
assign Y = B;
```

Y \ A	0	1
0	0	0
1	0	1

```
wor Y;  // declaration
assign Y = A;
assign Y = B;
```

Y \ A	0	1
0	0	1
1	1	1




```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

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

Signal resolution

Table 3-2—Truth table for wire and tri nets

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Table 3-3—Truth tables for wand and triand nets

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Table 3-4—Truth tables for wor and trior nets

wor/ trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Variable Data Types & Variable Arrays

- **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*> ;

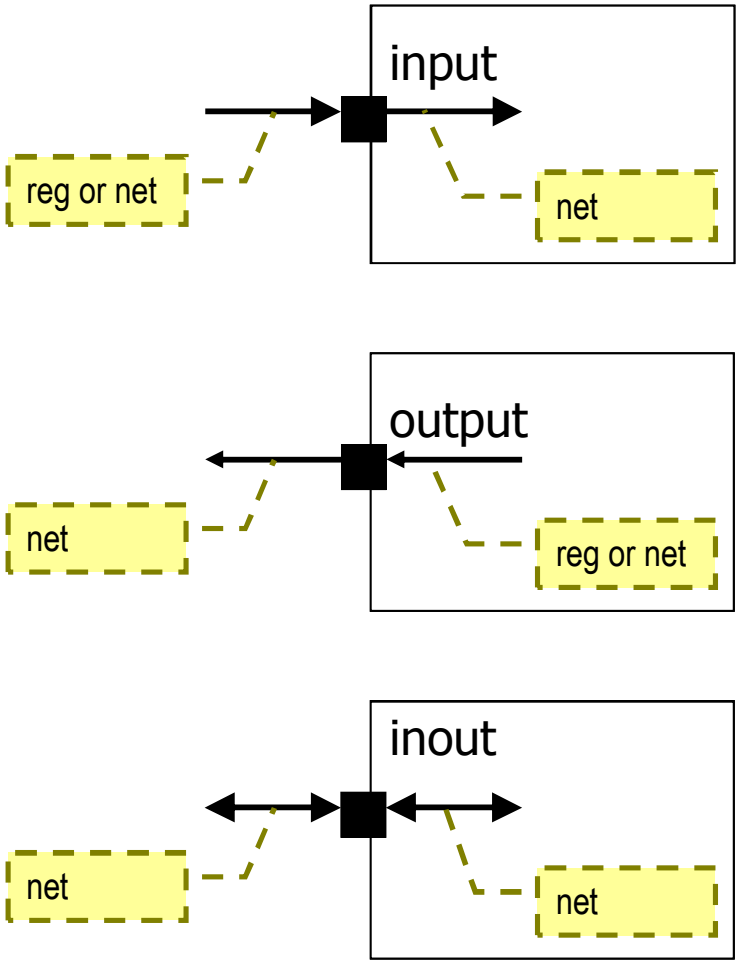
- Examples:
  - **reg** <*signal name*> ;
  - **reg** [7 : 0] out ;

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

Net 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

Connection through port



Port connection rules

- Input port must be net (i.e. wire).
- Output port can be either net or reg.
- Inout port must be net.

Memory

- Multi-dimensional variable array
  - Can not 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!!!

Array declaration and assignments

- Array declarations

```
reg [7:0] mema[0:255]; // declares a memory mema of 256 8-bit registers. The indices are 0 to 255
reg arrayb[7:0][0:255];  // declare a two-dimensional array of one bit registers
wire w_array[7:0][5:0]; // declare array of wires
integer inta[1:64];       // an array of 64 integer values
time chng_hist[1:1000] // an array of 1000 time values
integer t_index;
```

- Assignment to array elements

```
mema = 0;       // Illegal syntax- Attempt to write to entire array
arrayb[1] = 0;  // Illegal Syntax - Attempt to write to elements [1][0]..[1][255]
arrayb[1][12:31] = 0; // Illegal Syntax - Attempt to write to elements [1][12]..[1][31]
mema[1] = 0;      // Assigns 0 to the second element of mema
arrayb[1][0] = 0; // Assigns 0 to the bit referenced by indices [1][0]
inta[4] = 33559;  // Assign decimal number to integer in array
chng_hist[t_index] = $time; // Assign current simulation time to element addressed by integer index
```

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
);
```

Example 2

Data Type

- 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

MAC (Data Type Declarations)

```
`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

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

endmodule
```

Introduction to Verilog

Assigning Values – Numbers & Operators

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 numer
  - 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**

Short Quiz

- What is the actual value for 4’d017 in binary?

Short Quiz Answer

- What is the actual value for 4’d017 in binary?

  4’b0001, MSB is truncated (10001)

Example 3

Operators

- Arithmetic
- Bitwise
- Reduction
- Relational
- Equality
- Logical
- Shift
- Miscellaneous

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

Arithmetic Operators

Operator Symbol	Functionality	Examples ain = 5 ; bin = 10 ; cin = 2'b01 ; din = 2'b0z		
+	Add, Positive	bin + cin ⇒ 11	+bin ⇒ 10	ain + din ⇒ x
–	Subtract, Negate	bin – cin ⇒ 9	-bin ⇒ -10	ain – din ⇒ x
*	Multiply	ain * bin ⇒ 50		
/	Divide*	bin / ain ⇒ 2		
%	Modulus	bin % ain ⇒ 0		
**	Exponent*	ain ** 2 ⇒ 25		

- Treats vectors as a whole value
- Results unknown if any operand is Z or X
- Carry bit(s) handled automatically if result wider than operands
- Carry bit lost if operands and results are same size

**** Check synthesis tool for support***

Example 4

Bitwise Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
~	Invert each bit	~ain ⇒ 3b'010	~cin ⇒ 3'b10x
&	**AND** each bit	ain & bin ⇒ 3'b100	bin & cin ⇒ 3'b010
|	**OR** each bit	ain | bin ⇒ 3'b111	bin | cin ⇒ 3'b11x
^	**XOR** each bit	ain ^ bin ⇒ 3'b011	bin ^ cin ⇒ 3'b10x
^~ or ~^	**XNOR** each bit	ain ^~ bin ⇒ 3'b100	bin ~^ cin ⇒ 3'b01x

- Operates on each bit or bit pairing of the operand(s)
- Result is the size of the largest operand
- X or Z are both considered unknown in operands, but result maybe a known value
- Operands are left-extended if sizes are different

Reduction Operators

Operator Symbol	Functionality	Examples ain = 4'b1010 ; bin = 4'b10xz ; cin = 4'b111z		
&	**AND** all bits	&ain ⇒ 1'b0	&bin ⇒ 1'b0	&cin ⇒ 1'bx
~&	**NAND** all bits	~&ain ⇒ 1'b1	~&bin ⇒ 1'b1	~&cin ⇒ 1'bx
|	**OR** all bits	|ain ⇒ 1'b1	|bin ⇒ 1'b1	|cin ⇒ 1'b1
~|	**NOR** all bits	~|ain ⇒ 1'b0	~|bin ⇒ 1'b0	~|cin ⇒ 1'b0
^	**XOR** all bits	^ain ⇒ 1'b0	^bin ⇒ 1'bx	^cin ⇒ 1'bx
^~ or **~^**	**XNOR** all bits	~^ain ⇒ 1'b1	~^bin ⇒ 1'bx	~^cin ⇒ 1'bx

- Reduces a vector to a single bit value
- X or Z are both considered unknown in operands, but result maybe a known value

Relational Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
>	Greater than	ain > bin ⇒ 1'b0	bin > cin ⇒ 1'bx
<	Less than	ain < bin ⇒ 1'b1	bin < cin ⇒ 1'bx
>=	Greater than or equal to	ain >= bin ⇒ 1'b0	bin >= cin ⇒ 1'bx
<=	Less than or equal to	ain <= bin ⇒ 1'b1	bin <= cin ⇒ 1'bx

- Used to compare values
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- X or Z are both considered unknown in operands and result is always unknown

Equality Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b110 ; cin = 3'b01x	
==	Equality	ain == bin ⇒ 1'b0	cin == cin ⇒ 1'bx
!=	Inequality	ain != bin ⇒ 1'b1	cin != cin ⇒ 1'bx
===	Case equality	ain === bin ⇒ 1'b0	cin === cin ⇒ 1'b1
!==	Case inequality	ain !== bin ⇒ 1'b1	cin !== cin ⇒ 1'b0

- Used to compare values
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- For equality/inequality, X or Z are both considered unknown in operands and result is always unknown
- For case equality/case inequality, X or Z are both considered distinct values and operands must match completely

Example 5

Logical Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b000 ; cin = 3'b01x		
!	Expression not true	!ain ⇒ 1'b0	!bin ⇒ 1'b1	!cin ⇒ 1'bx
&&	**AND** of two expressions	ain && bin ⇒ 1'b0	bin && cin ⇒ 1'bx	
||	**OR** of two expressions	ain || bin ⇒ 1'b1	bin || cin ⇒ 1'bx	

- Used to evaluate single expression or compare multiple expressions
  - Each operand is considered a single expression
  - Expressions with a zero value are viewed as false (0)
  - Expressions with a non-zero value are viewed as true (1)
- Returns a 1 bit scalar value of Boolean true (1) / false (0)
- X or Z are both considered unknown in operands and result is always unknown

Shift Operators

Operator Symbol	Functionality	Examples ain = 3'b101 ; bin = 3'b01x	
<<	Logical shift left	ain << 2 ⇒ 3'b100	bin << 2 ⇒ 3'bx00
>>	Logical shift right	ain >> 2 ⇒ 3'b001	bin >> 2 ⇒ 3'b000
<<<	Arithmetic shift left	ain <<< 2 ⇒ 3'b100	bin <<< ⇒ 3'bx00
>>>	Arithmetic shift right	ain >>> 2 ⇒ 3'b111 (signed)	bin >>> 2 ⇒3'b000 (signed)

- Shifts a vector left or right some defined number of bits
- Left shifts (logical or arithmetic): Vacated positions always filled with zero
- Right shifts
  - Logical: Vacated positions always filled with zero
  - Arithmetic (unsigned): Vacated positions filled with zero
  - Arithmetic (signed): Vacated position filled with sign bit value (MSB value)
- Shifted bits are lost
- Shifts by values of X or Z (right operand) return unknown

Example 6

Miscellaneous Operators

Operator Symbol	Functionality	Format & Examples
?:	Conditional test	(condition) ? true_value : false_value sig_out = (sel == 2'b01) ? a : b
{ }	Concatenate	ain = 3'b010 ; bin = 3'110 {ain,bin} ⇒ 6'b010110
{ { } }	Replicate	{3 {3'b101} } ⇒ 9'b101101101

Example 7

Example 8

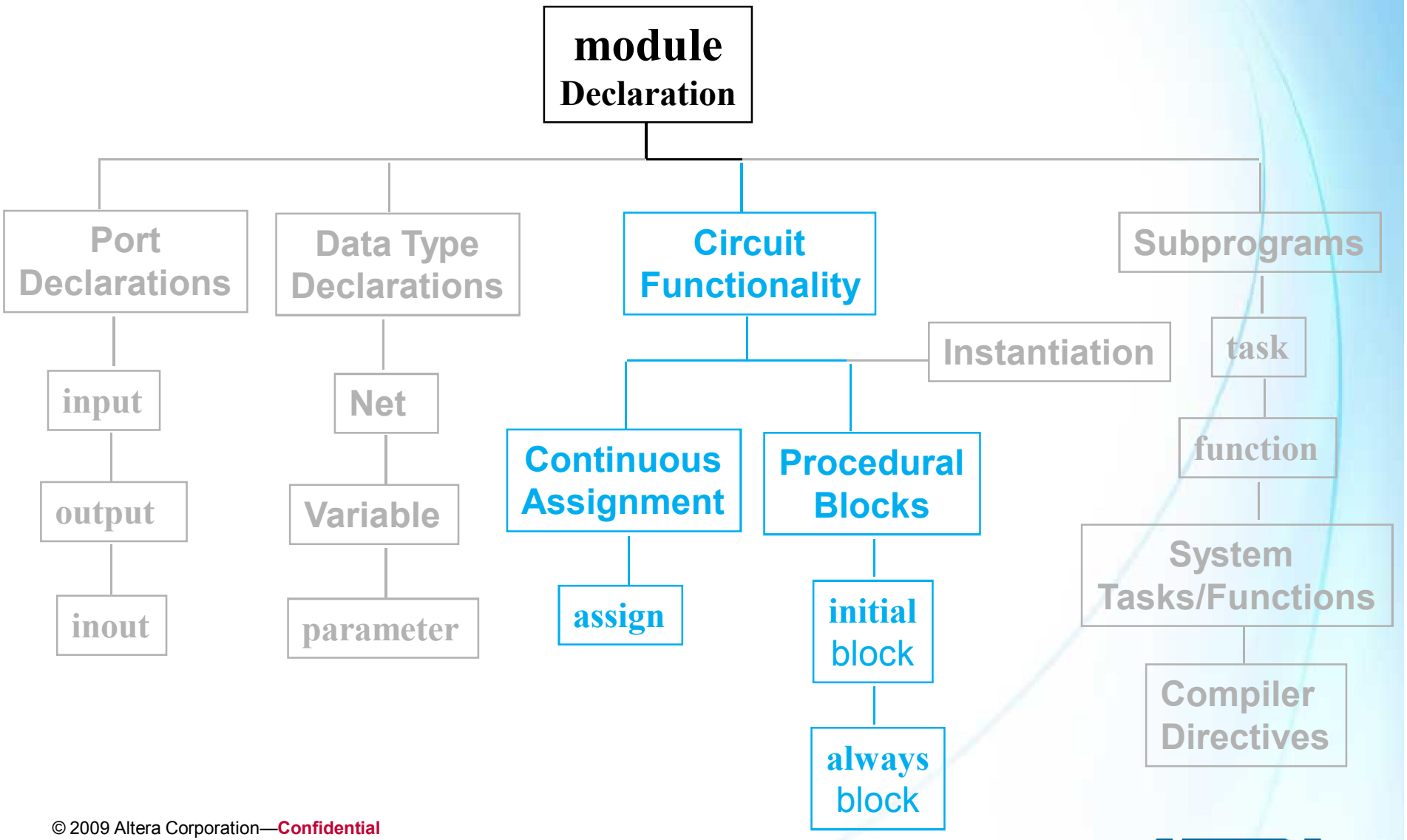
Operator Precedence

Operator(s)	Priority
+ - ! ~ & ~& etc. (unary* operators)	High
**	
* / %	
+ - (binary operators)	
<< >> <<< >>>	
< > <= >=	
== != === !==	
& (binary operator)	
^ ~^ ^~ (binary operators)	
| (binary operator)	
&&	
||	
?:	
{ } { { } }	Low

- ( ) used to override default and provide clarity

** Unary operators have only one operand*

Let's Take a Look at



Introduction to Verilog

Behavioral Modeling –
Continuous Assignments

Continuous Assignments

- 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 */
```

OR

is equivalent to

  - By using the **assign** statement

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

Continuous Assignments 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

```
module and_func (
    input [7:0] ina, inb,
    output [7:0] out
);

assign out = ina & inb;

endmodule
```

MAC (Continuous Assignment)

```
`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

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

endmodule
```

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 ≠ 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

Exercise 2

Please go to Exercise 2
in the Exercise Manual

Introduction to Verilog

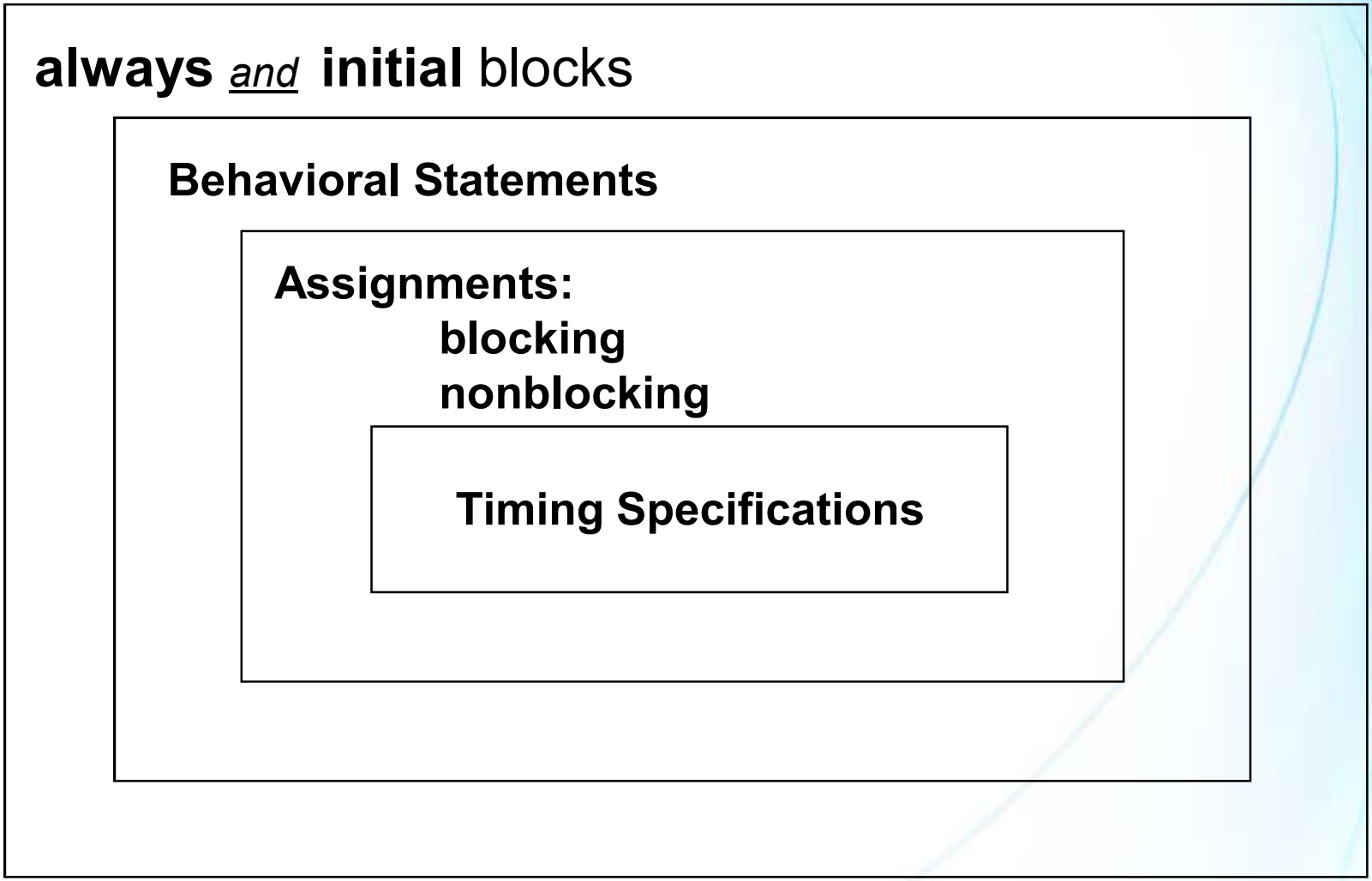
Behavioral Modeling –
Procedural Blocks

Two Procedural Blocks

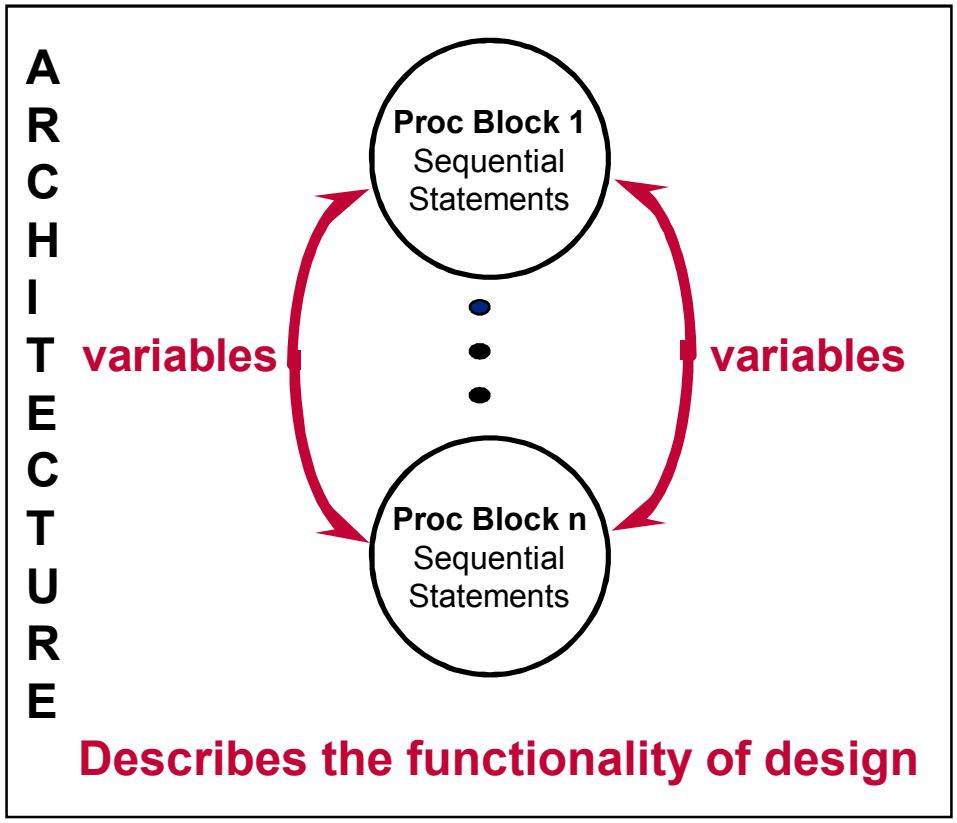
- **initial**
  - Used to initialize behavioral statements for simulation
- **always**
  - Used to describe the circuit functionality using behavioral statements

- ⇨ 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



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, the statements are executed sequentially
  - Order of statements within an **always/initial** block does matter

Procedural Block Characteristics

1) LHS must be a variable data type (e.g. reg, integer, real, time)
2) LHS can be a bit-select or part-select
3) LHS can be a concatenation of any of the above
4) RHS can be expression containing net data type, variable data type or function 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

Example 9

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
```

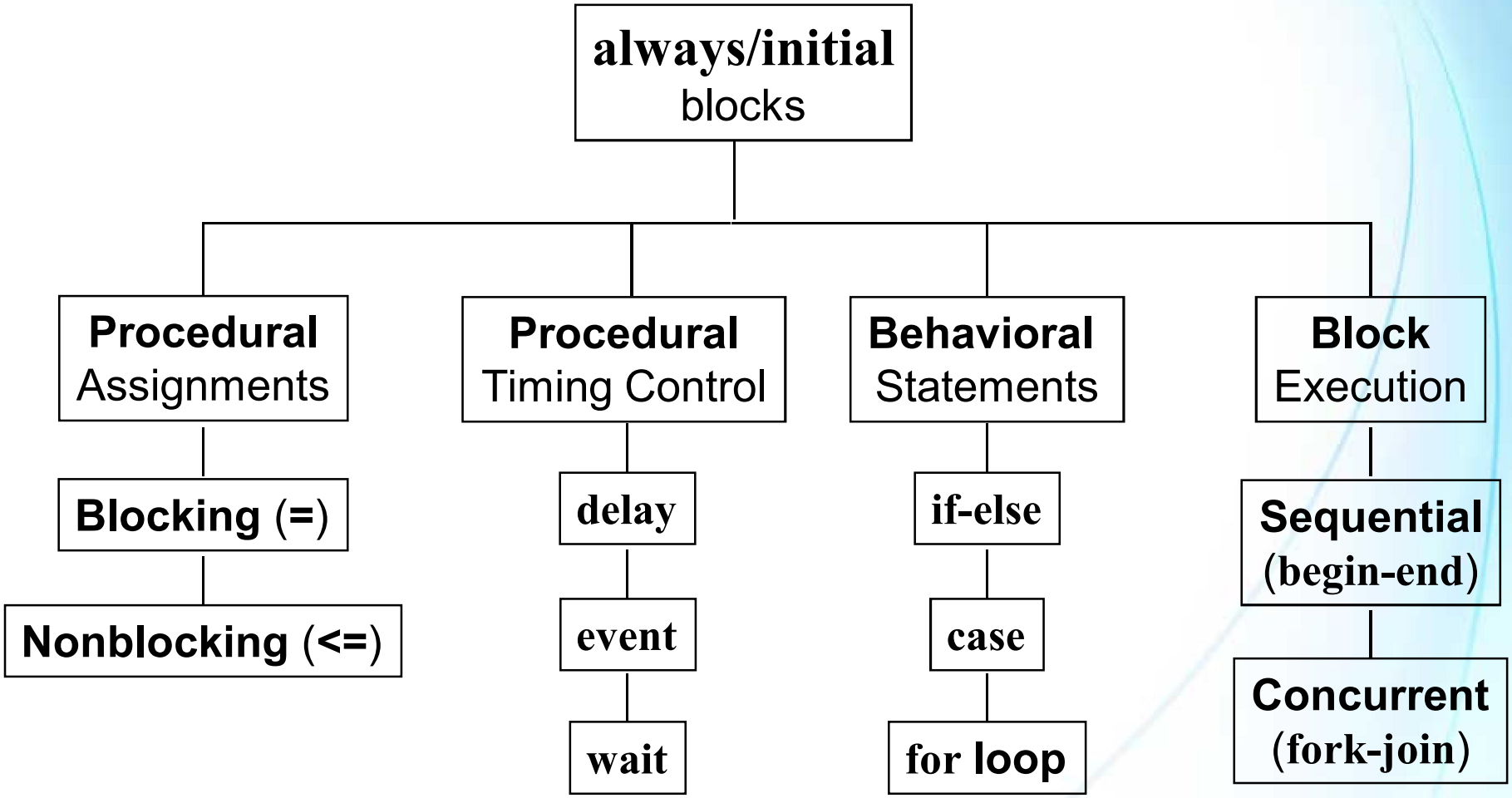
Time	Statement(s) Executed
0	clk = 1'b0;
25	clk = 1'b1;
50	clk = 1'b0;
75	clk = 1'b1;
100	$finish;

Naming Procedural Blocks

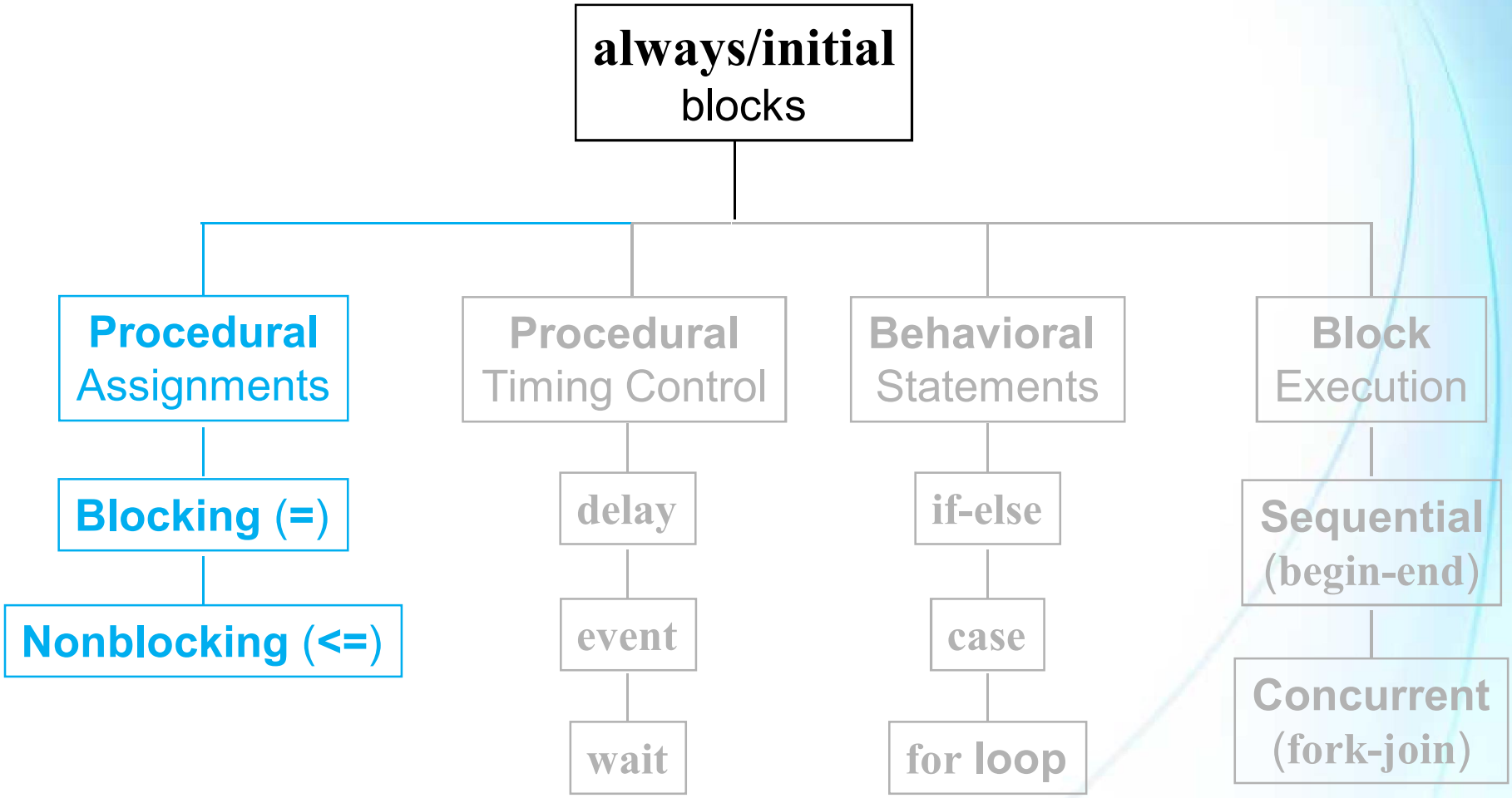
- Procedural blocks may be named by adding : <name> after **begin**
- Advantages
  - Allows the procedural block to be referenced in other places within the code by name
  - Allows declaration of objects local to the procedural block
  - Allows monitoring of procedural block by name in simulation tools

```
initial
begin : clock_init
    clk = 1'b0;
end

always
begin : clock_proc
    #(period/2) clk = ~clk;
end
```

always/initial Blocks



always/initial Blocks (Procedural Assignments)



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

Procedural Assignment Types

In Verilog, there are two types of procedural assignment statements

- Blocking
- Nonblocking

Procedural Assignment Types (cont.)

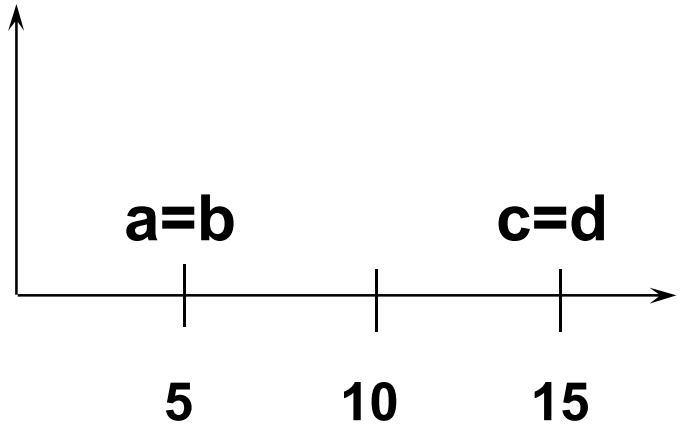
- Blocking (=) : Updates LHS assignments blocking execution of other assignments in the procedural block until finished
  - RHS (inputs) sampled when statement executed
  - LHS (outputs) updated immediately or after defined delay
  - Statements following must wait until blocking statement completely finished and LHS assignments are made (including delay) to begin execution

- Nonblocking (<=) : Schedules LHS assignments without blocking execution of the statements that follow in a sequential block
  - 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 nonblocking LHS assignments are made to begin execution

Blocking vs. Nonblocking Assignments

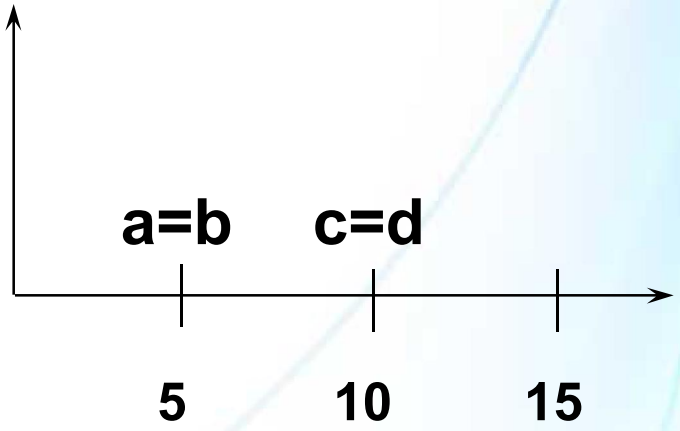
Blocking (=)

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



Nonblocking (<=)

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



Evaluating Blocking & Nonblocking

```
initial begin
    a = 1'b0;  //Assgnmnt0
    b = 1'b1;  //Assgnmnt1
    #5 a <= b; //Assgnmnt2
    #5 b <= a; //Assgnmnt3
end
```

This may seem confusing but it is perfectly valid Verilog and is here to illustrate the behavior of blocking and nonblocking statements

1. Assignment 0 executes and assigns **a** to 0 blocking remaining assignments
2. Assignment 1 executes and assigns **b** to 1 blocking remaining assignments
3. Assignment 2 executes and schedules **a** to take on the current value of **b** (1) after 5 time steps
4. Assignment 3 executes and schedules **b** to take the current value of **a** (0) after 5 time steps
5. Process ends and 5 time steps later **a** is updated to 1 and **b** is updated to 0

Notes on Blocking & Nonblocking Assignments

1) Avoid making multiple simultaneous (same time step) assignments to same variable in different processes
   - May causes indeterminate value (race condition)
2) For simultaneous (same time step) assignments to same variable in same process
   - Last blocking variable assignment overwrites previous assignments
   - Last scheduled non-blocking variable assignment overwrites previous assignments
3) During a time step, blocking assignments executed before nonblocking assignments
   - This is due to Verilog event queue (order in which events must be handled)
   - Exception: When non-blocking assignment triggers blocking assignment in another process
4) Not recommended to use both blocking & nonblocking assignments in same procedural block
   - Though legal Verilog, considered poor coding style
   - Requires very good understanding of Verilog event queue
   - Easy to produce incorrect or unexpected behavior

1)
```
initial #4 a = 1'b0;
initial #4 a = 1'b1;
```

2)
```
initial begin
    a = 1'b0;
    a = 1'b1; // Last
end
initial begin
    #4 a <= 1'b1;
    #4 a <= 1'b0; // Last
end
```

3)
```
initial #4 a = 1'b0; // 1st
initial #4 b <= 1'b1; // 2nd
```

4)
```
always begin
    a <= in * 2;
    b = a + 1'b1; // Old 'a'
    …
end
```

Notes Explanations

1) Procedural blocks assumed to be in parallel with other procedural blocks, thus no guarantee which assignment will be executed first; results in race condition
2) Since variable assignments are sequential, in both cases last assignment will overwrite the first whether blocking or nonblocking
3) Blocking assignment executed first, then nonblocking, so **a** will be equal to 1 at end of simulation cycle (this should avoided too!)
4) Very poor coding; **b** is actually using the previous value of **a** each time

1)
```
initial #4 a = 1'b0;
initial #4 a = 1'b1;
```

2)
```
initial begin
    a = 1'b0;
    a = 1'b1; // Last
end
initial begin
    #4 a <= 1'b1;
    #4 a <= 1'b0; // Last
end
```

3)
```
initial #4 a = 1'b0; // 1st
initial #4 a <= 1'b1; // 2nd
```

4)
```
always begin
    a <= in * 2;
    b = a + 1'b1; // Old 'a'
end
```

Last Note on Nonblocking

- Unlike with continuous assignments, non-blocking assignments allow scheduling of any number of events to the same variable at different times without cancelling or overwriting prior scheduled events

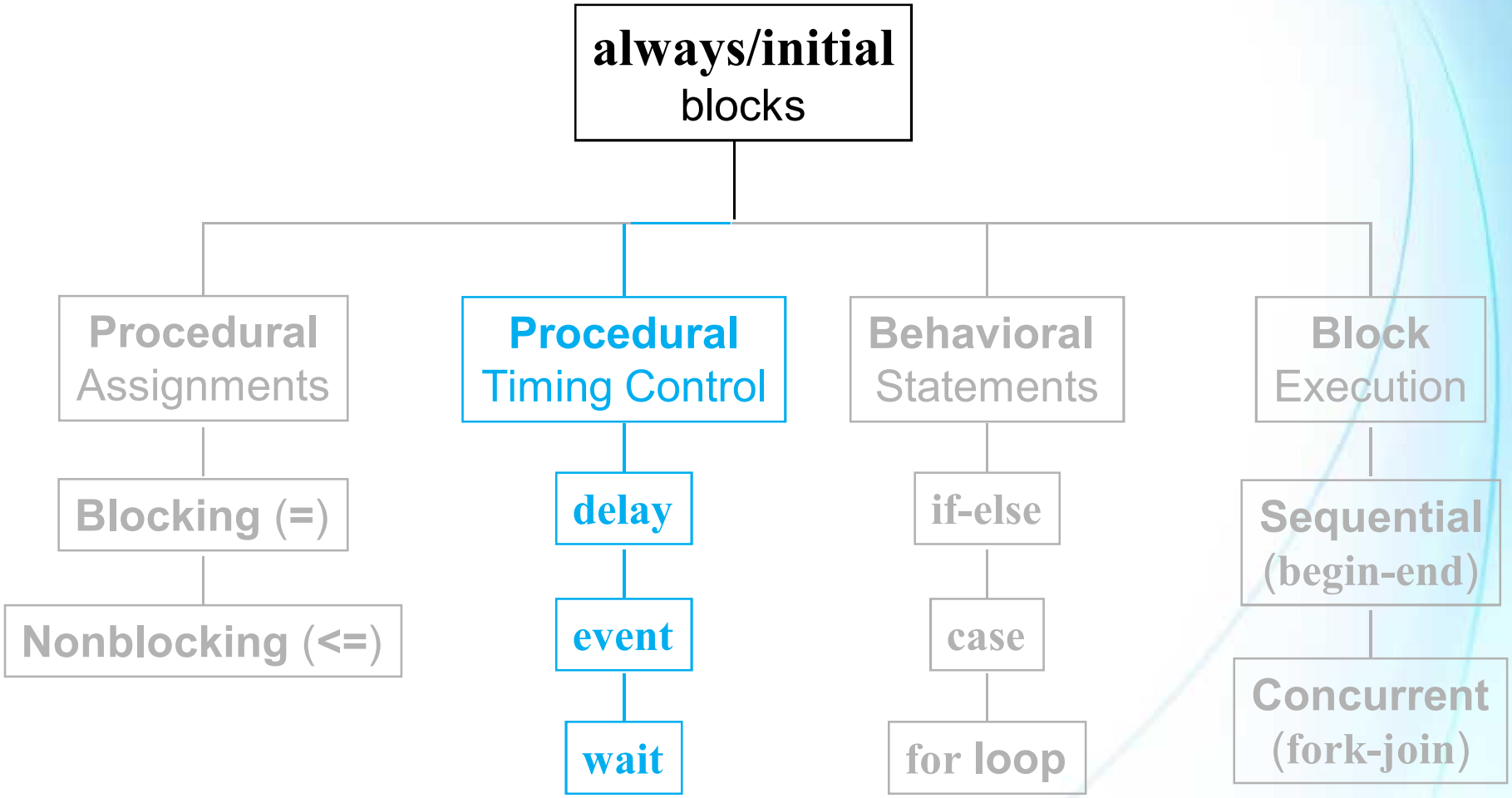
```
initial begin
    a <= 1'b0;
    #2 a <= 1'b1;
    #4 a <= 1'b0;
    #6 a <= 1'b1;
end
```

*Each nonblocking statement schedules **a** to a value at some time in the future without disturbing the scheduling of the other statements*

Process Execution Time

- All procedural blocks (processes) in all modules in a design execute together
  - Process execution time starts at time 0
  - Process execution time advances after all processes at current time step have completed their current execution cycle
  - Current execution cycle ends when procedural block reaches one of the following
    - End of initial block
    - Blocking assignment with delay
    - Event control is reached
    - Wait statement
- Process execution time is used to determine the behavior of a model
  - Simulation tool: Process execution time is the same as simulation time
  - Synthesis tool: Must generate functionally equivalent physical logic/hardware

always/initial Blocks (Procedural Timing Control)



Procedural Timing Control

- Delay control
- Event control
- Wait statement
- Named event
  - Not discusssed

Delay Controls for Procedural Assignment

- Use #<value> notation to delay executing procedural assignment

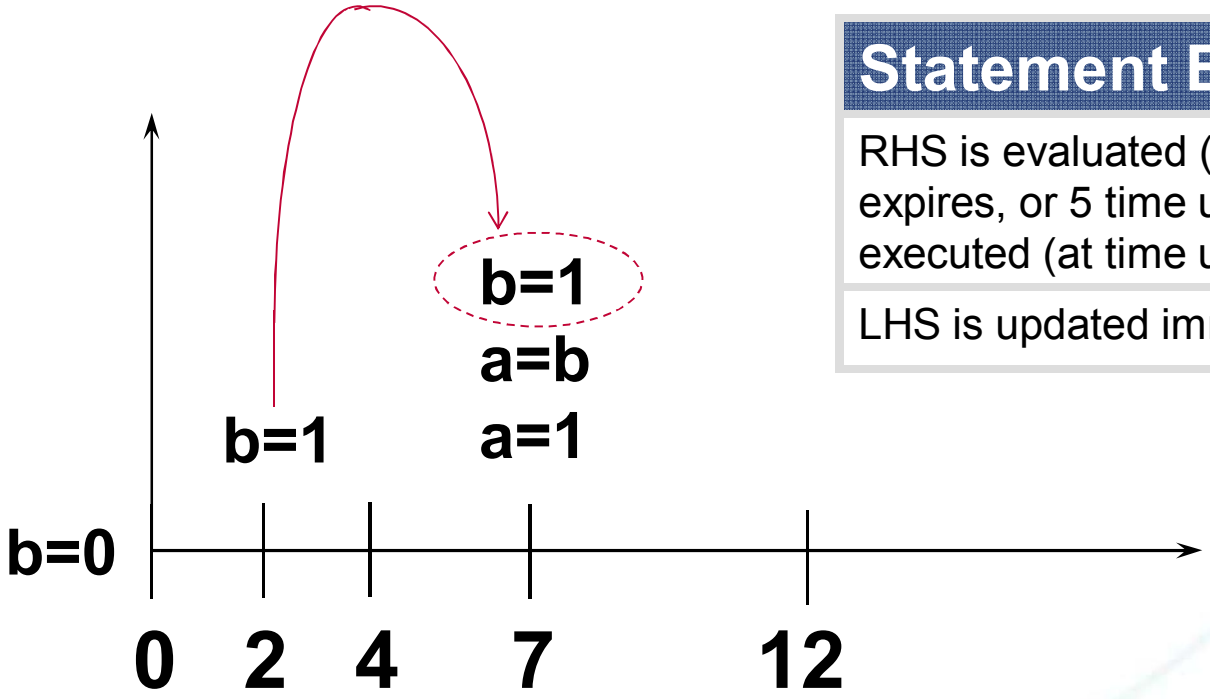
- Regular (Inter-Assignment) Delay Control
- Intra-assignment Delay Control
- Zero Delay Control

- Ignored for synthesis

Regular (Inter-Assignment) Delay Control

```
#5  a = b;
```

- Delays both read (RHS) and write (LHS) portions of statement execution

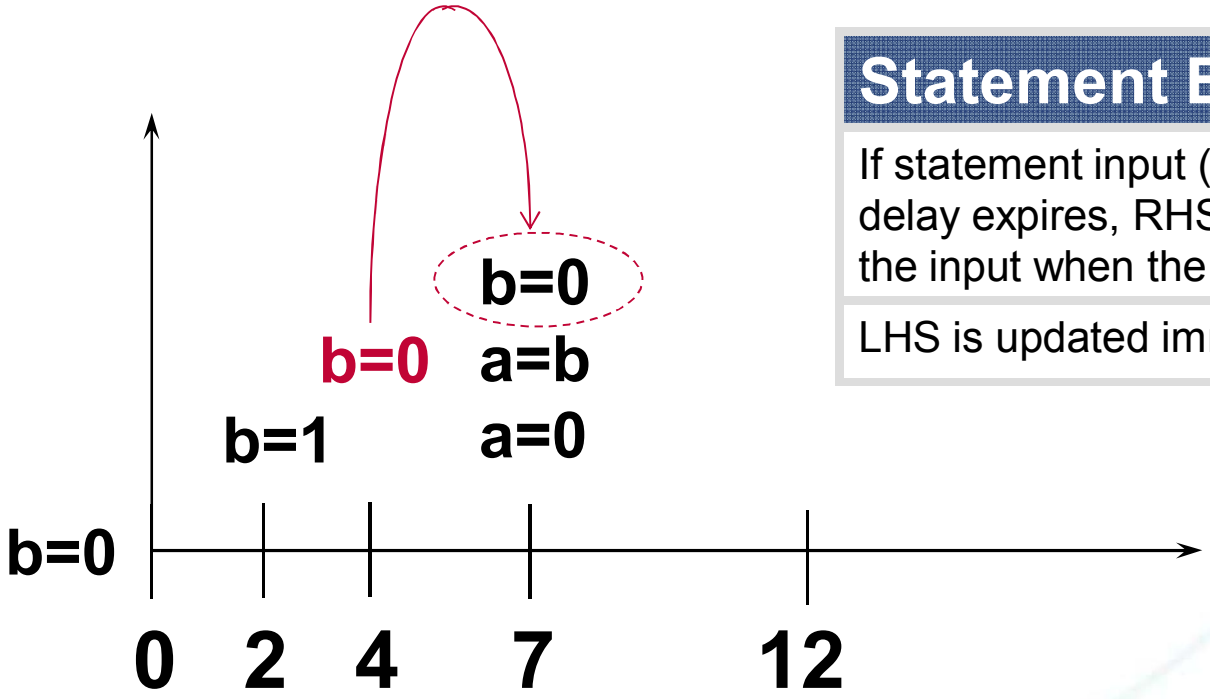


Statement Execution (1)
RHS is evaluated (i.e. **b** is read) after delay expires, or 5 time units after statement is executed (at time unit 7)
LHS is updated immediately

Regular (Inter-Assignment) Delay Control (cont.)

```
#5  a = b;
```

- Delays both read (RHS) and write (LHS) portions of statement execution



Statement Execution (2)

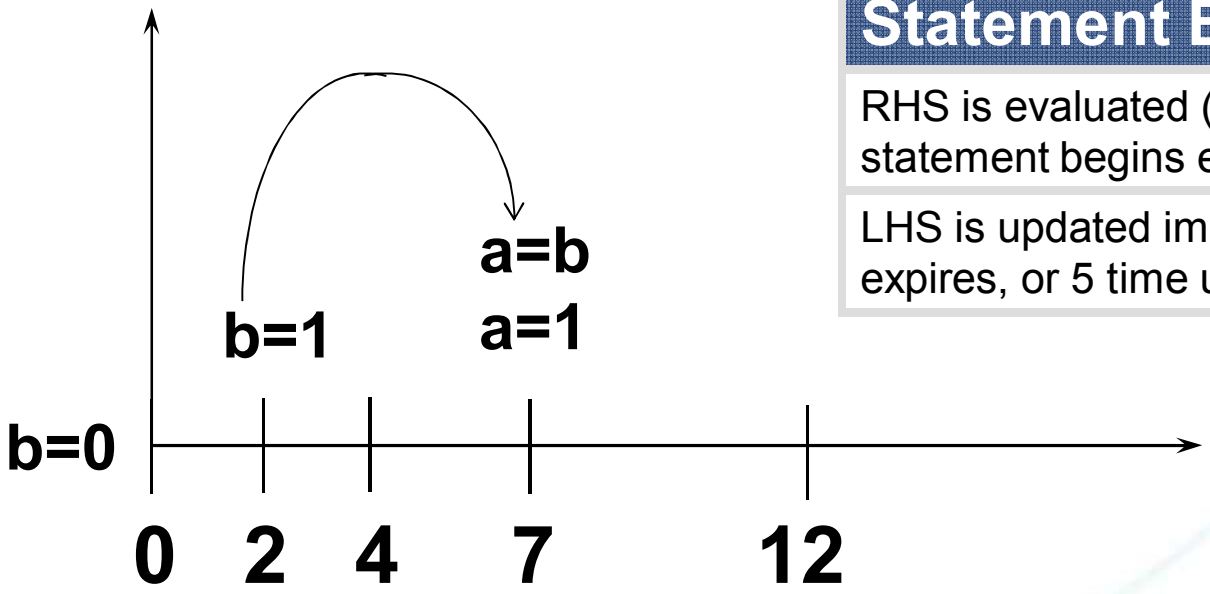
If statement input (RHS, or **b)** changes before delay expires, RHS picks up the new value of the input when the read actually occurs

LHS is updated immediately with new value

Intra-Assignment Delay Control

```
a = #5 b;
```

- Delays only write (LHS) portions of statement execution



Statement Execution (1)
RHS is evaluated (i.e. **b** is read) when statement begins execution
LHS is updated immediately after delay expires, or 5 time units later

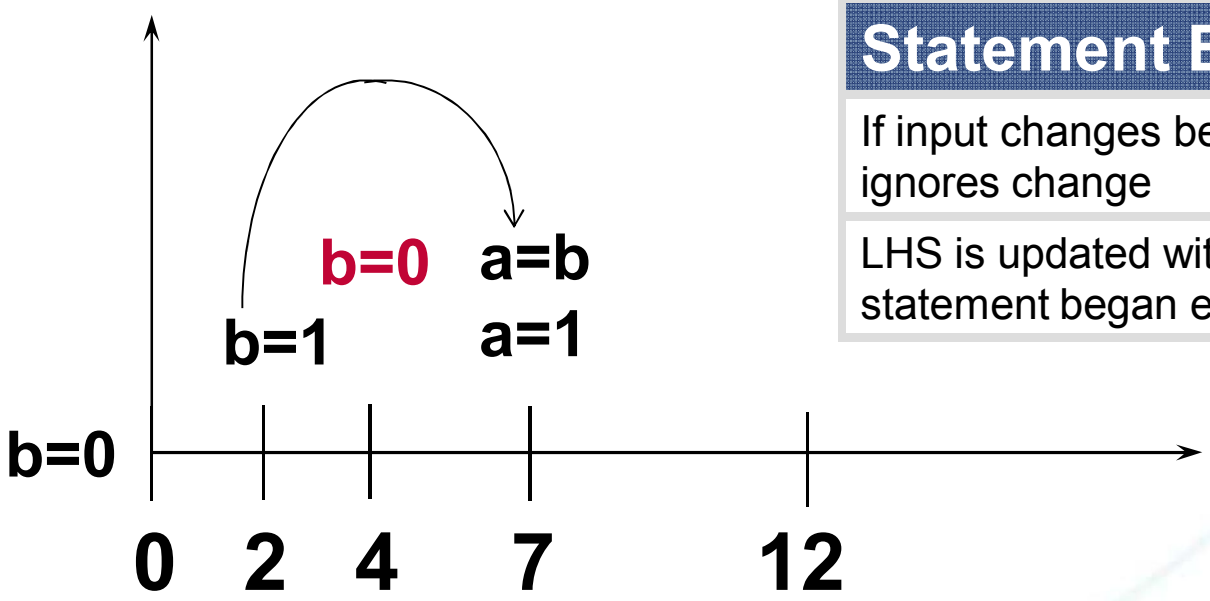
Intra-Assignment Delay Control (cont.)

```
a = #5 b;
```

<=>

```
temp = b;
#5 a = temp;
```

- Delays only write (LHS) portions of statement execution



Statement Execution (2)

If input changes before delay expires, RHS ignores change

LHS is updated with value of RHS when statement began execution

Example Uses of Procedural Delay

- Inter-Assignment Delay
  - Use with blocking assignment operator in testbenches to sequence test stimuli
- Intra-Assignment Delay
  - Use with nonblocking assignment operator to model transport delay (e.g. delay lines or transmission lines)

Example 10

Zero Delay Control

```
initial begin
    a = 0;
    b = 0;
end

initial begin
    #0 a = 1;
    #0 b = 1;
end
```

Statement Execution
All four statements will be executed at simulation time 0
Since #0 used for statements a = 1 and b = 1 have, they will be executed last.

- Provides a way of controlling the order of execution at 0 time
- Still not recommended to assign different values to a variable simultaneously or in different processes

`timescale Compiler Directive Preview

Time unit for delays in module

Precision for delay values used by simulator

```
`timescale 1 ns / 10 ps

module mult_acc (
    input [7:0] ina, inb,
    input clk, clr,
    output [15:0] out
);1
        …
    assign #5 …
    assign #25 …
```

Note: Other compiler directives discussed later.

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**

Examples*

```
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 events using
      //        "or" keyword
      r11 = @ (posedge clk or enable) r12;
end
```

**Note: This usage not supported by synthesis.*

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
```

wait Statement

- Provides level-sensitive event control
- Uses wait keyword

wait (*expression*)

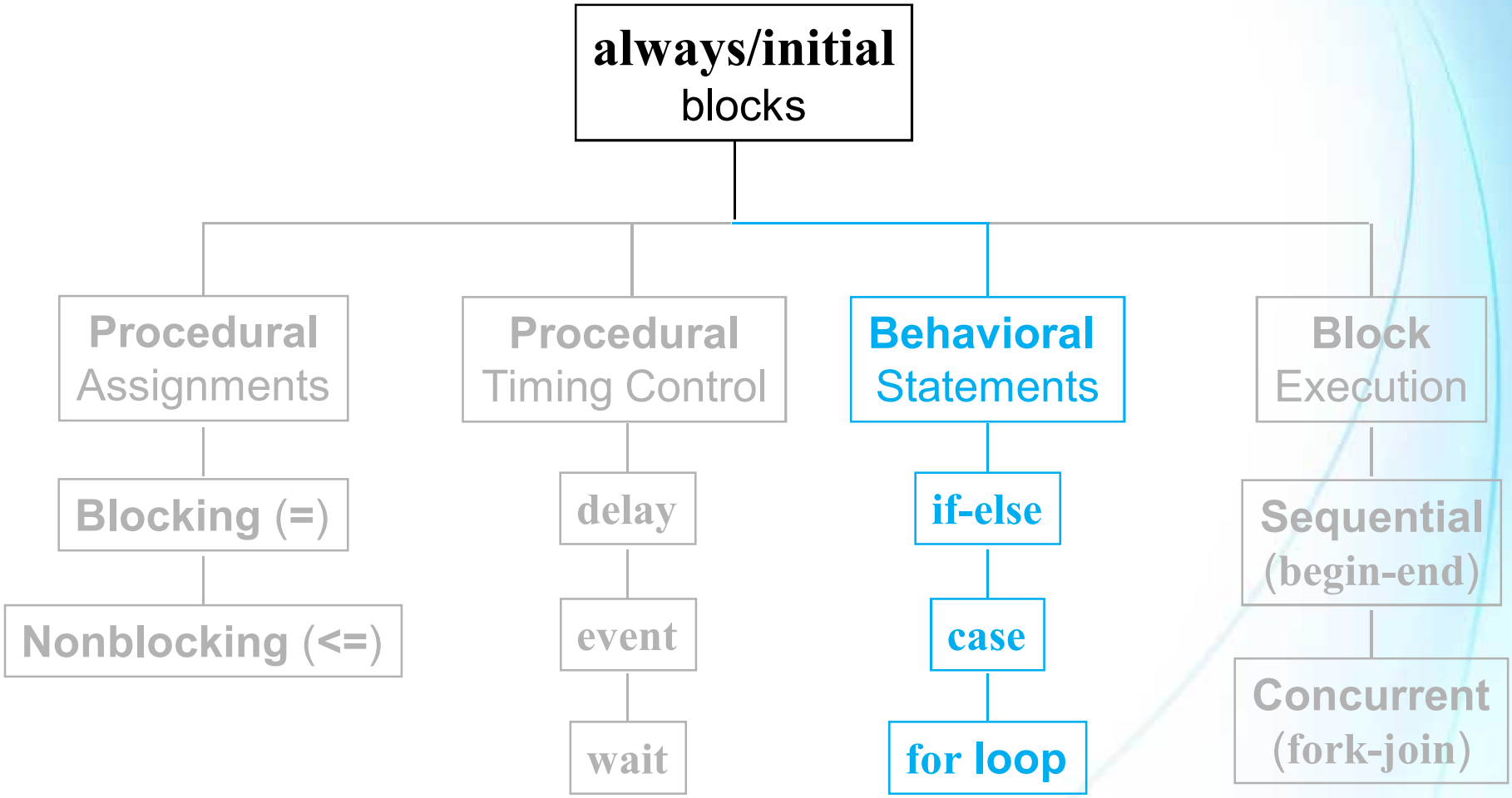
- Pauses execution of the procedural block until wait statement is satisfied
  - If level test not satisfied, procedural block pauses until it is
  - If level test already satisfied, procedural block continues execution without pausing
- Not supported by synthesis

Example

```
initial begin
    …
    wait (gate) r1 = r2;  //Assignment0
    wait (!gate) r3 = r4; //Assignment1
    …
end
```

- *Assignment0 must pause until gate is true (1) before r1 takes on value of data. Statement does not pause if gate already true*
- *Assignment1 must pause until gate is false (0) before r1 takes on value of data. Statement does not pause if gate already false*

always/initial Blocks (Behavioral Statements)



Behavioral Statements

- Describe behavior and express order
- Must be used inside procedural block

- Behavioral Statements
  - **if-else** statement
  - **case** statement
  - Loop statements

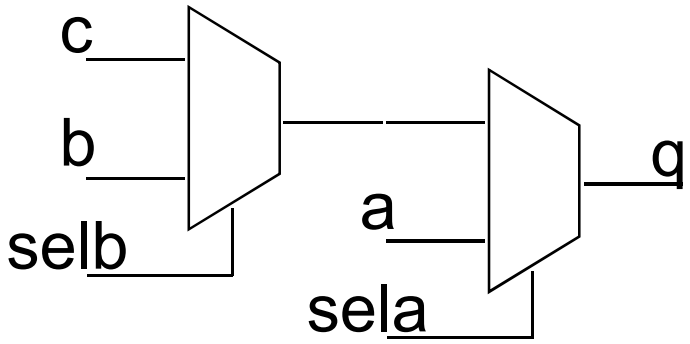
if-else Statements

- 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;
```



if-else Statements

- 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

Example 11

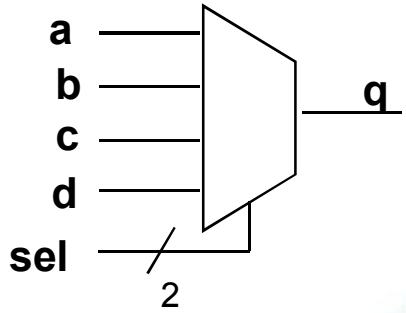
case Statement

- 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

Example 12

Two Other Forms of case Statements

- **casez**
  - Treats both **Z** and **?** in the case conditions 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 both **X** and **Z** 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**

Example 13

Exercise 3

Please go to Exercise 3
in the Exercise Manual

Loop Statements

- **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 Loops

- **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

Example 14

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!

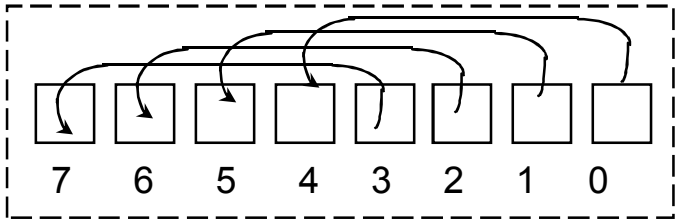
Example 15

for Loop

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

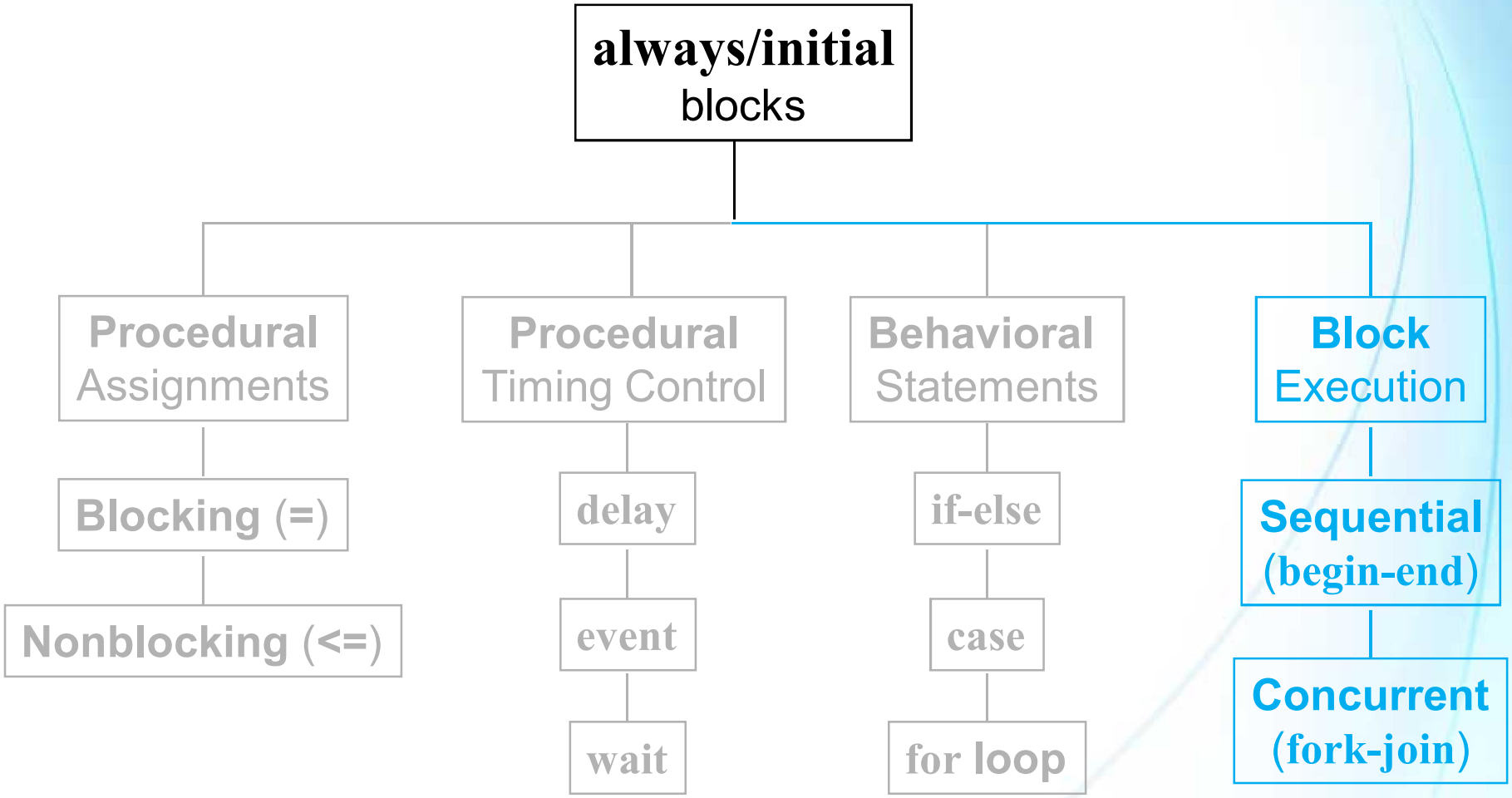
```
// declare the index for the FOR LOOP
integer  i;

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
```

4-bit Left Shifter



Example 16

always/initial Blocks (Block Execution)



Block Execution

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

- Two types of block execution
  - Sequential Blocks
  - Parallel Blocks

Block Execution 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

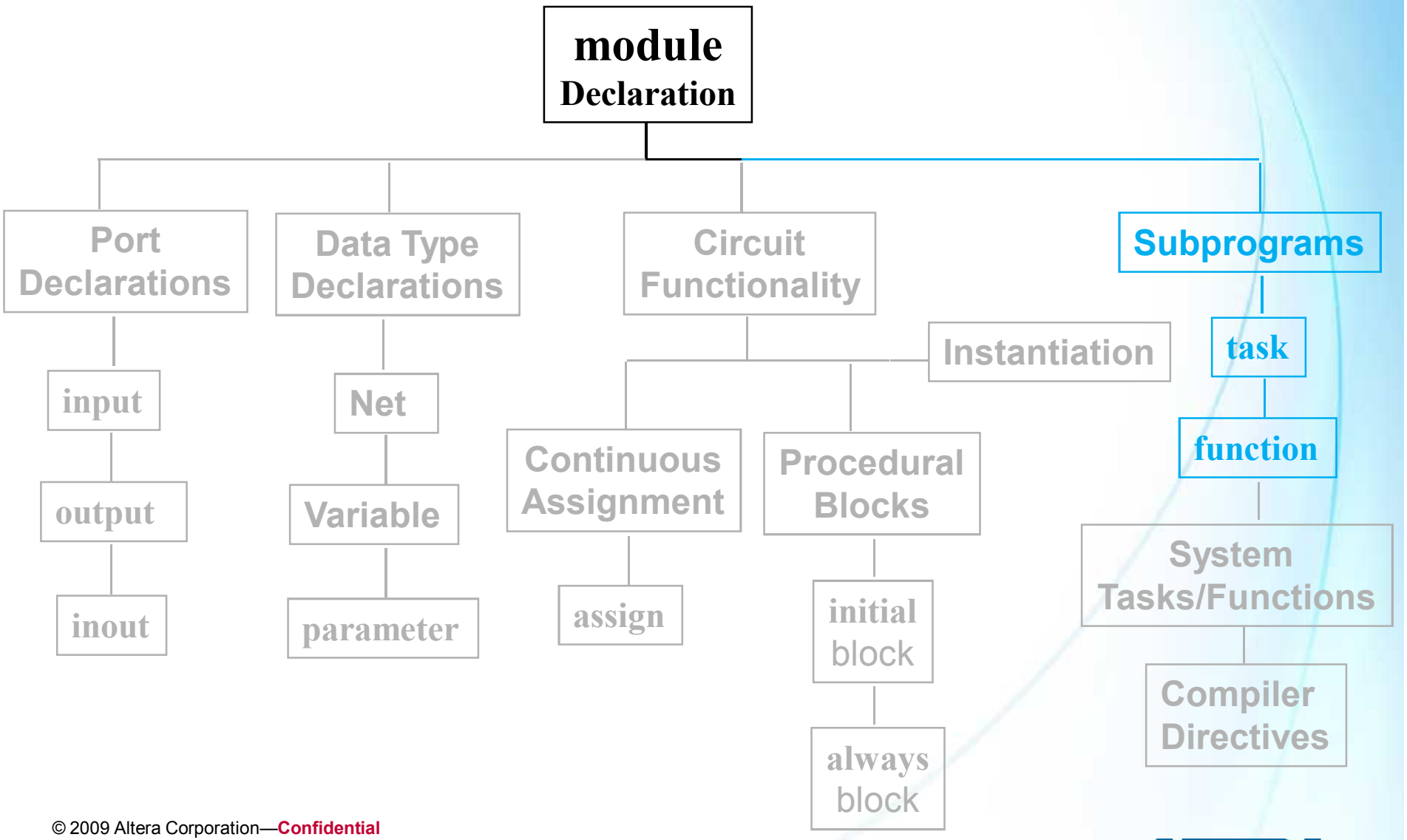
Sequential vs. Parallel Blocks

- 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



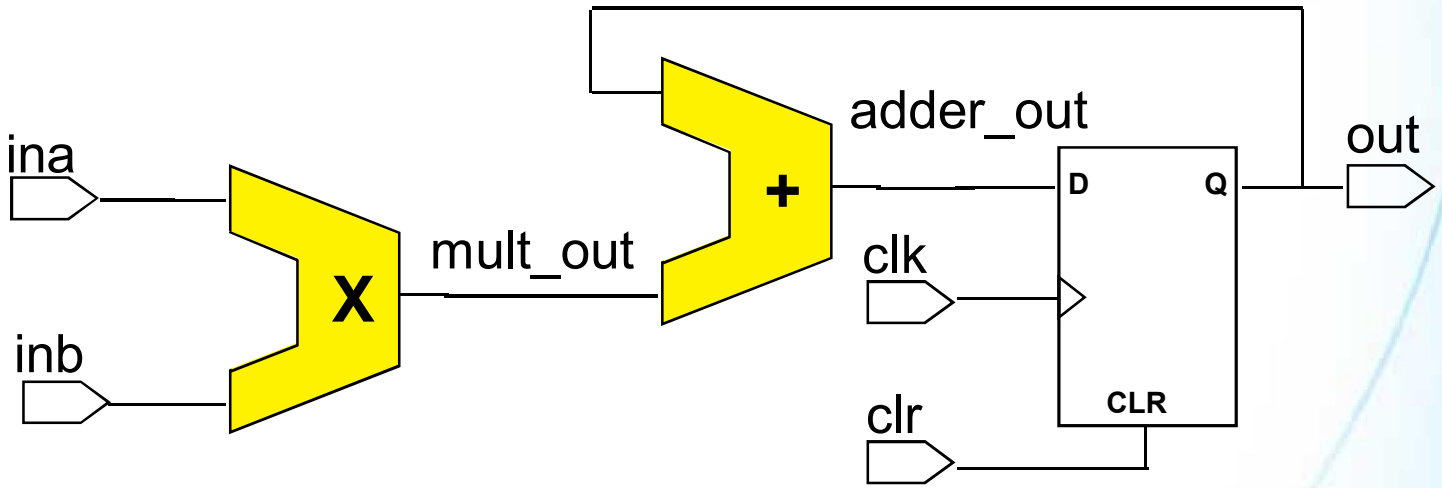
Introduction to Verilog

Behavioral Modeling –
Tasks & Functions

Verilog Functions and Tasks

- Function and Tasks are subprograms
- Consist of behavioral statements (like a procedural block)
- 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 Definition:

```
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
```

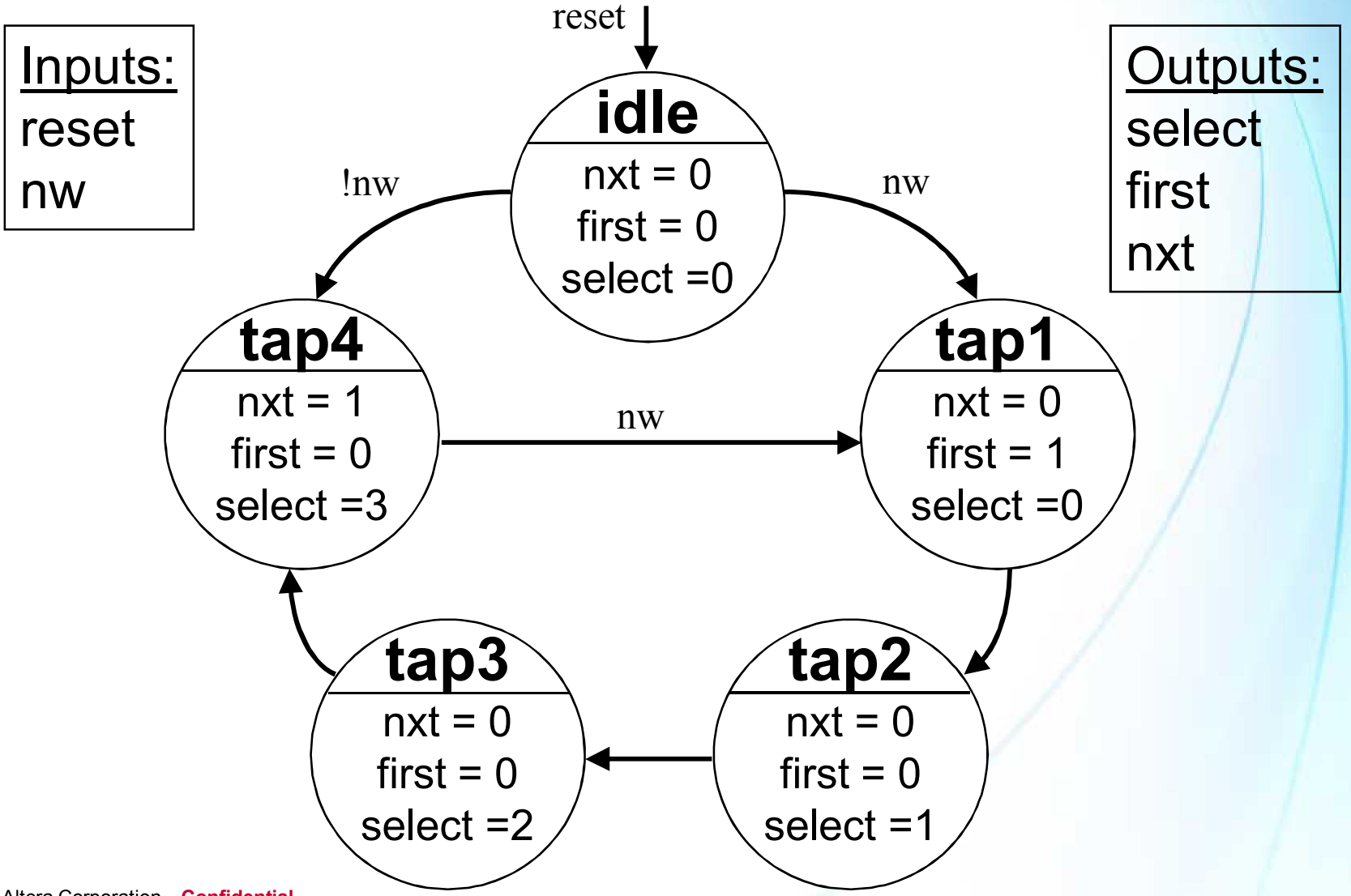
Example Function Invocation

```
`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)

    specify
        $setup (dataa, posedge clk, set);
        $hold (posedge clk, dataa, hld);
        $setup (datab, posedge clk, set);
        $hold (posedge clk, datab, hld);
    endspecify

endmodule
```

Example 17

Create Task for State Machine Output



Task Definition – State Machine Output

```
task stm_out;
    input [2:0] filter;
    output reg nxt, first;
    output reg [1:0] sel;
    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 – State Machine

```
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
```

Example 18

Functions vs. Tasks

Functions

- Always execute in zero time
  - Cannot pause their execution
  - Can not contain any delay, event, or timing control statements
- Must have at least one input argument
  - Inputs may not be affected by function
- Arguments may not be outputs and inouts
- Always return a single value
- May call another function but not a task

Tasks

- May execute in non-zero simulation time
  - May contain delay, event, or timing control statements
- May have zero or more input, output, or inout arguments
- Modify zero or more values
- May call functions or other tasks

Review - Behavioral Modeling

Continuous Assignment

```
module full_adder4 (
    output [3:0] fsum,
    output fco,
    input [3:0] a, b,
    input cin
);

assign {fco, fsum} = cin + a + b;

endmodule
```

Procedural Block

```
module fll_add4 (
    output reg [3:0] fsum,
    output reg fco,
    input [3:0] a, b,
    input cin
);

always @(cin or a or b)
    {fco, fsum} = cin + a + b;

endmodule
```

- Will produce the same logical model and functionality

Introduction to Verilog

Behavioral Modeling –
RTL Processes

RTL Processes

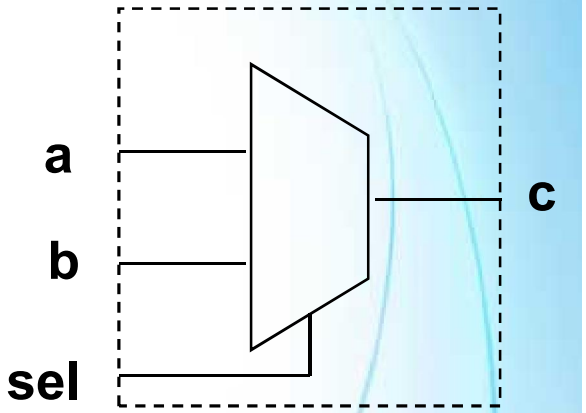
- If you remember, RTL is a synthesizable behavioral coding style
- RTL coding style involves two types of procedural blocks (processes)
  - Combinatorial Process
  - Clocked Process

Two Types of RTL Processes

- **Combinatorial Process**
  - Sensitive to all inputs used in the combinatorial logic

```
always @ (a, b, sel)
always @ *
```

Sensitivity list includes all inputs used In the combinatorial logic

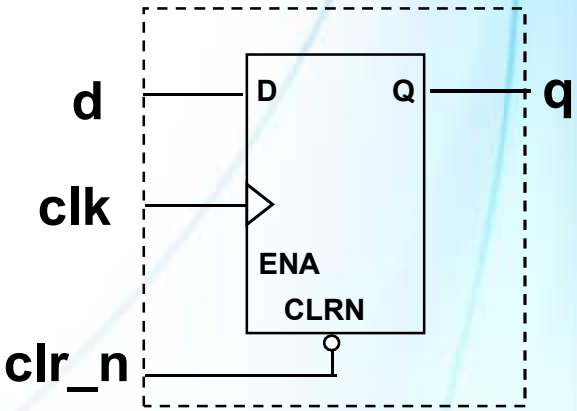


** is a Verilog shortcut to manually having to add all inputs*

- **Clocked Process**
  - Sensitive to a clock or/and control signals

```
always @(posedge clk, negedge clr_n)
```

Sensitivity list does not include the **d** *input, only the clock and asynchronous control signals*



Functional Latch vs. Functional Flipflop

Level-Sensitive Latch

```
module latch (
   input d, gate,
   output reg q
);

   always @(d, gate)
      if (gate)
         q = d ;

endmodule
```

Edge-Triggered Flipflop

```
module dff (
   input d, clk,
   output reg q
);

   always @(posedge clk)
      q <= d ;

endmodule
```

Example 19

Synchronous vs. Asynchronous

Synchronous Preset & Clear

```
module dff_sync (
    input d, clk, sclr, spre,
    output reg q
);

    always @(posedge clk) begin
        if (sclr)
            q <= 1'b0;
        else if (spre)
            q <= 1'b1;
        else
            q <= d;
    end

endmodule
```

Asynchronous Clear

```
module dff_async (
    input d, clk, aclr,
    output reg q
);

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

endmodule
```

Example 20

Clock Enable

Clock Enable

```
module dff_ena (
    input d, enable, clk;
    output reg q
);

/* If clock enable port does not exist in
    target technology, then a mux in
    front of the d input is generated */

    always @( posedge clk )
        if (enable)
            q <= d;

endmodule
```

Functional Counter

```
module cntr (
    input aclr, clk,
    input [7:0] d,
    input [1:0] func,  // Controls functionality
    output reg [7:0] q,
);

    always @ (posedge clk, posedge aclr) begin
        if (aclr)
            q <= 8'h00;
        else
            case (func)
                2'b00: q <= d;  // Loads counter
                2'b01: q <= q + 1;  // Counts up
                2'b10: q <= q - 1;  // Counts down
            endcase
    end

endmodule
```

Blocking/Nonblocking Rule of Thumb

- Use **blocking operator** (=) for **combinatorial logic**
- Use **nonblocking operator** (<=) for **sequential logic**

- This avoids confusion and unintended hardware implementations during RTL synthesis

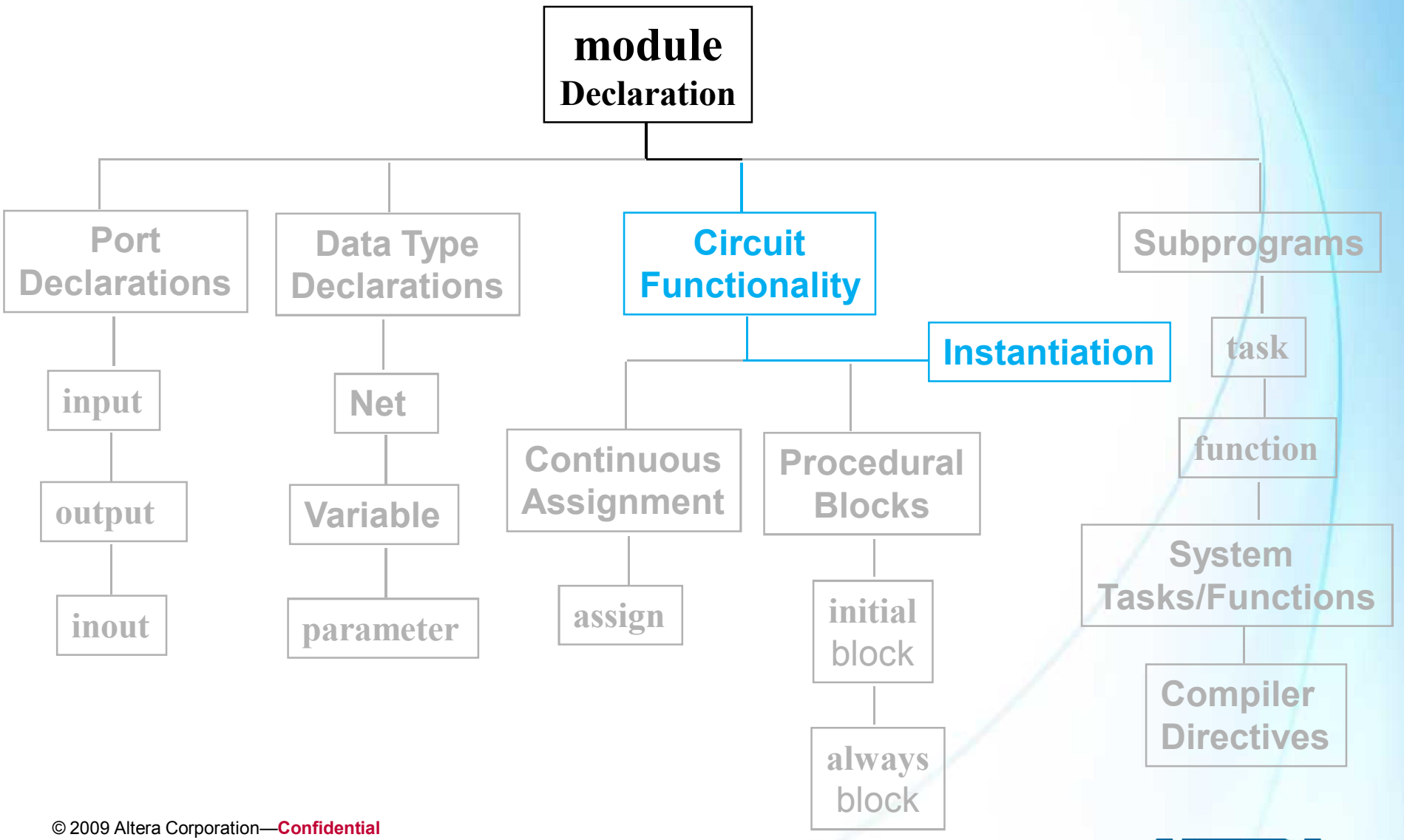
MAC (Behavioral Modeling)

```
`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

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

endmodule
```

Exercise 4

Please go to Exercise 4
in the Exercise Manual

Let's Take a Look at



Introduction to Verilog

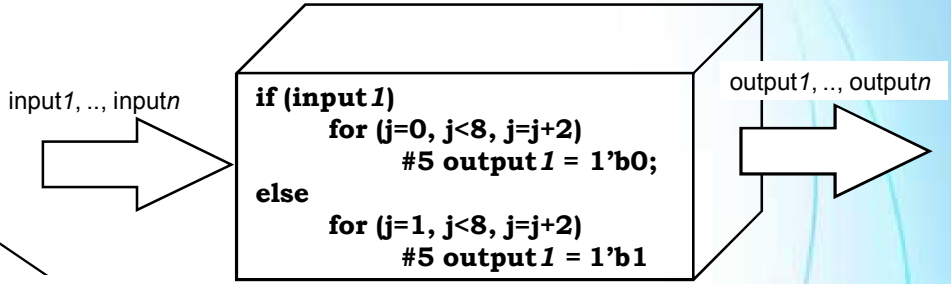
Structural Modeling

Levels of Abstraction

Behavioral Modeling

Behavioral Models (function only)

RTL Models (function and inferred structure)



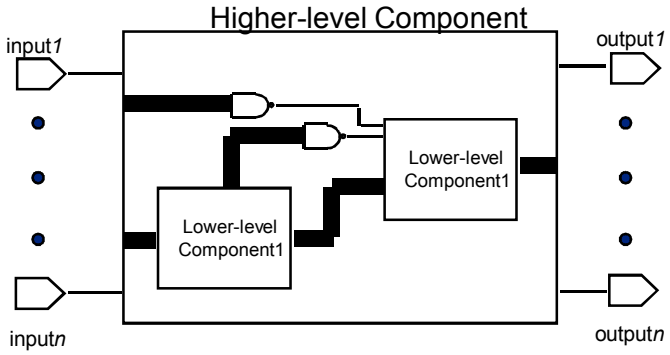
Abstract

Detailed

Structural Modeling

Gate Level Models (function and structure)

Switch Level Models (function and structure)



Structural Modeling

- Defines function and structure of a digital circuit
- Adds to Hierarchy

Verilog Structural Modeling

- ***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; see Appendix for examples
- ***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
  - ⇨ Switch level modeling will not be discussed

Gate-Level Modeling

- Verilog has predefined gate primitives

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

- Instantiation Format:

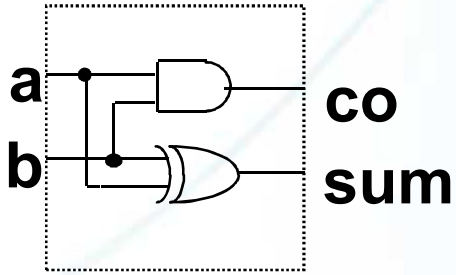
<gate_name> #<delay> <instance_name> (port_list);

  - ***<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
```



Example 21

Module Instantiation

- 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

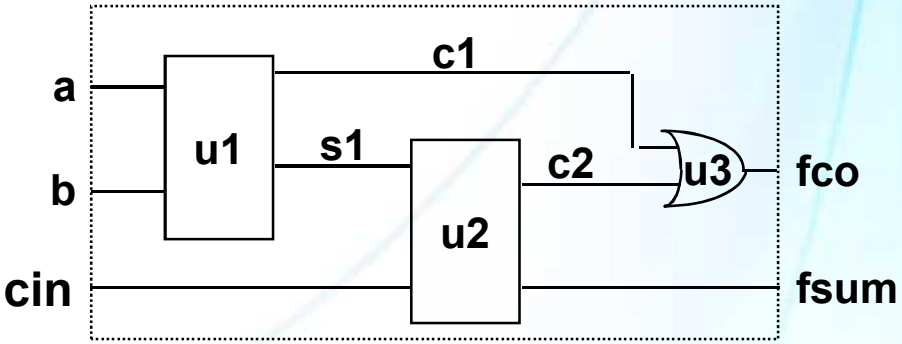
Connecting Module Instantiation Ports

- 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**

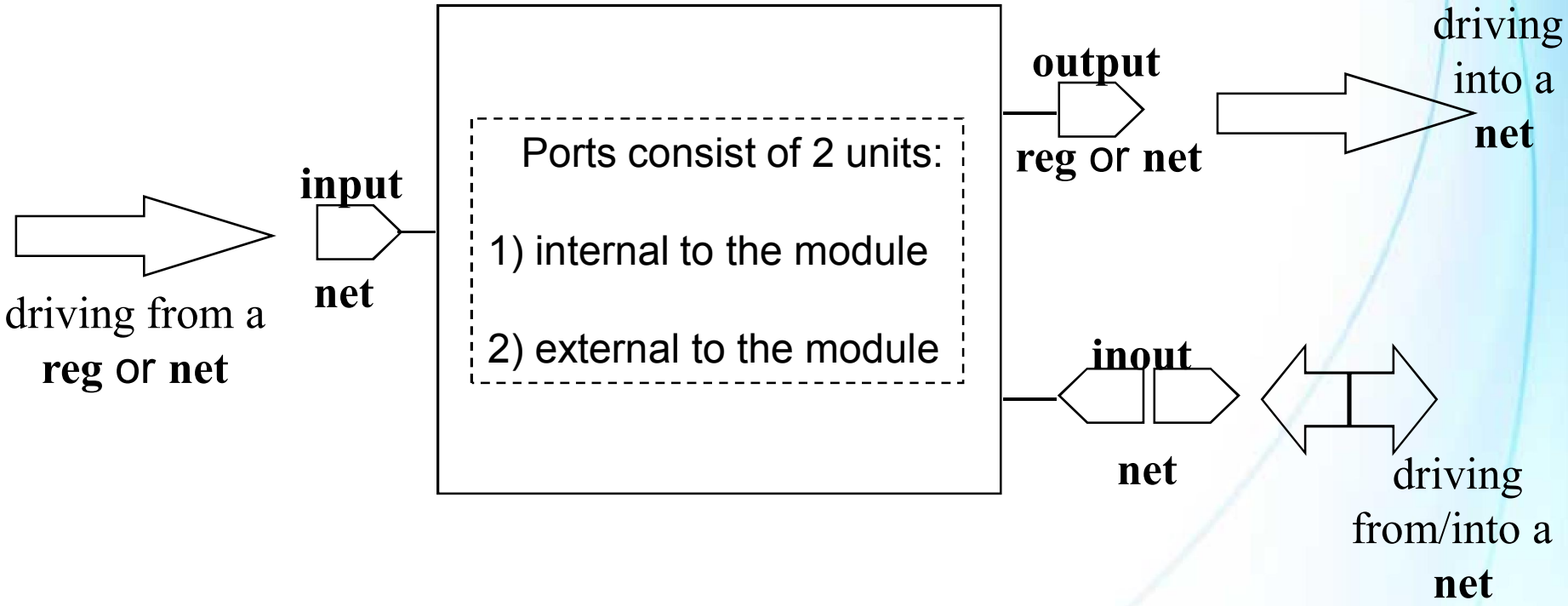
```
module full_adder (
    output fco, fsum,
    input cin, a, b
);
    wire c1, s1, c2;

    half_adder u1 (c1, s1, a, b);
    half_adder u2 (.a(s1),  .b(cin),
        .sum(fsum), .co(c2));
    or  u3(fco, c1, c2);

endmodule
```



**Note: This is the half-adder module from slide 142*

Port Connection Rules

These are the requirements for port connections when modules are instantiated within other modules



Example 22

Overwriting Parameters

- If a lower-level module contains parameters, there are two methods for overwriting its parameter values during instantiation
  - Done at compile time as parameters must resolve to constant values during compilation
- defparams
- Module instance parameter assignment

Defparam

- Use **defparam** statement and hierarchical name to overwrite parameters for module instantiations
- May be placed outside Verilog module (i.e. another file)

```
module full_adder (
    output fco, fsum,
    input cin, a, b
);

    wire c1, s1, c2;

    defparam  u1.and_delay = 4,  u1.xor_delay = 6;
    defparam  u2.and_delay = 3,  u2.xor_delay = 5;

    half_adder  u1 (c1, s1, a, b);
    half_adder  u2 (.a(s1),  .b(cin),
        .sum(fsum), .co(fco));
    or  u3(fco, c1, c2);

endmodule
```

Module Instance Parameter Assignment

- Parameter definition occurs during module instantiation
- Introduced in Verilog '2001
- Recommended method as it is easier to read and not prone to error as defparams

```
module full_adder (
     output fco, fsum,
     input cin, a, b
);

     wire c1, s1, c2;

     half_adder  #(4, 6)
          u1 (c1, s1, a, b);
     half_adder  #(.and_delay(3), .xor_delay(5))
          u2 (.a(s1),  .b(cin), .sum(fsum), .co(fco));
     or  u3(fco, c1, c2);

endmodule
```

Ordered list method

Name method (recommended)

To just use the (default) value defined in the lower-level module, in the name method, use empty value (e.g. .and_delay( )) or leave parameter assignment out entirely

MAC (Module Instantiation)

```
`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

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

endmodule
```

MAC (Module Instantiation & Local Parameter)

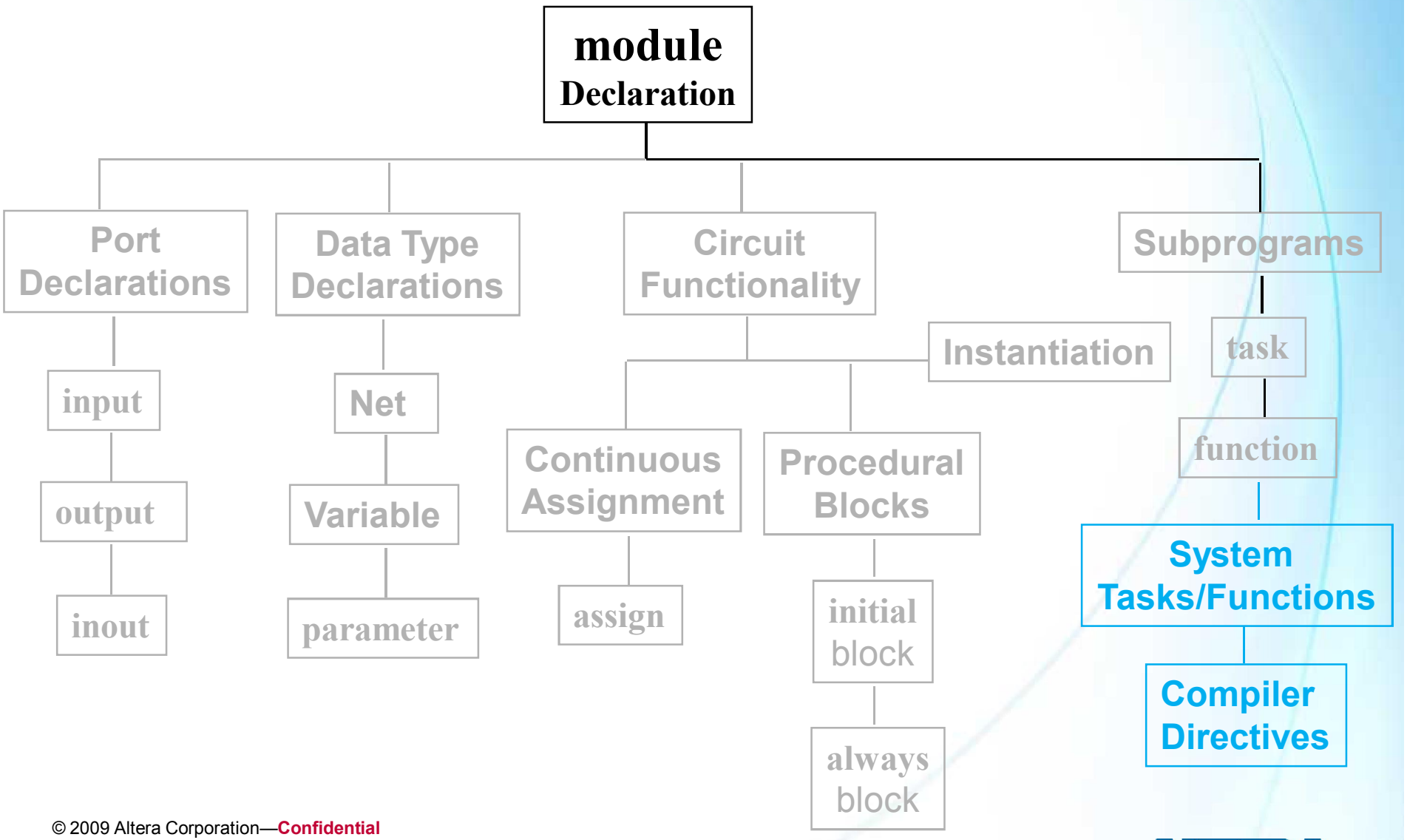
```
`timescale 1 ns/ 10 ps

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

    wire [15:0] mult_out, adder_out;

    localparam 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

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

endmodule
```

*Name **mult_size** defined as local parameter so it cannot accidentally be overwritten at compile time*

Exercise 5

Please go to Exercise 5
in the Exercise Manual

Let's Take a Look at



Introduction to Verilog

Compiler Directives & System Tasks

Compiler Directives & System Tasks

- Compiler directives
  - Commands issued to direct or control compiler behavior before or during compilation
  - Indicated by the ` (back tick) character
    - Key to the left of 1 key on keyboard, not the apostrophe!
  - Some may be placed within Verilog module, others must be placed outside Verilog module
- System tasks & functions
  - Built-in Verilog tasks and functions
  - Begin with **$** character
  - Provide a variety of useful capabilities
- We will cover a few of the compiler directives or system tasks and functions to give you an idea of the capabilities
  - Find many more with a quick web search

Compiler Directive Examples

- `timescale
- `include
- `define / `undef
- `ifdef / `ifndef / `elsif / `else / `endif

`timescale Compiler Directive

- Defines module timing using two values
  - <*reference_time_unit*>: specifies the unit of measurement for times and delays
  - <*time_precision*>: specifies the precision to which the delays are rounded off during simulation
- Only 1, 10, and 100 are supported integers
- Must be placed outside of module definition

- Ex: **`timescale** 1 ns / 10 ps

`include Compiler Directive

- Includes entire contents of another Verilog source file
- Use to incorporate commonly used library or definition files

```
`include half_adder.v    // Same as typing the entire
                         //    half_adder.v file here

module full_adder (
    output fco, fsum,
    input cin, a, b
);
    wire c1, s1, c2;
    …
```

`define and `undef Compiler Directives

- Define and undefine macros that perform text substitution
- Use macro by typing `macro_name
- Anything can be substituted
  - Numbers
  - Characters and strings
  - Comments
  - Keywords
  - Operators
- Can be placed outside or inside of module definition
- Can pass arguments
- Have global visibility
  - Once defined, can be used within any Verilog file read afterwards

```
`define HADD_DELAY_VAL1 #(4,6)
`define HADD_DELAY_VAL2 #(3,5)
`define MY_OR(or_out, or_ina, or_inb)  \
            or #5 (or_out, or_ina, or_inb)

module full_adder (
    output fco, fsum,
    input cin, a, b
);
    wire c1, s1, c2;

    half_adder `HADD_DELAY_VAL1
        u1 (c1, s1, a, b);
    half_adder `HADD_DELAY_VAL2
        u2 (.a(s1),  .b(cin), .sum(fsum), .co(c2));
    `MY_OR (fco, c1, c2);

endmodule

`undef MY_OR
```

`ifdef / `ifndef / `elsif / `else / `endif Compiler Directives

- Provide support for conditional compilation
  - Ex. Support different variations of a module
  - Ex. Choose different sets of stimulus
- Allow designer to compile Verilog statements based on whether macros have been defined
  - Any valid Verilog statements can be placed within
- Definitions
  - **`ifdef** <*macro_name*> : code compiled if macro defined
  - **`ifndef** <*macro_name*> : code compiled if macro not defined
  - **`elsif** <*macro_name*> : code compiled if macro defined and previous **`ifdef**/**`ifndef** not satisfied
  - **`else** : code compiled if previous **`ifdef**/**`ifndef** not satisfied
  - **`endif** : end code compilation region (one per **`ifdef**/**`ifndef**)
- Can be placed outside or inside of module definition
- Can be nested

Conditional Compilation Example

```
// Conditional Compilation

`ifdef TEST  // Compile counter_test only if macro TEST has been defined
    module counter_test;
    …
    endmodule

`else  // Compile the module counter as default
    module counter;
    …
    endmodule

`endif
```

System Tasks and Functions

- Simulation control & time
- Display control
- Math functions
  - Not discussed
- Conversion
  - Not discussed
- File I/O
  - Not discussed

Simulation Control & Time

- **$stop** task
  - Pauses simulation
  - Simulator still running
  - Add argument **1** or **2** to print message about simulator state
- **$finish** task
  - Stop simulation
  - Shut down simulator
  - Add argument **1** or **2** to print message about simulator state
- **$time** function
  - Returns current time in simulation

Display Control

- All display controls ignored for synthesis
- **$display**(…) task
  - Writes formatted message to simulator display whenever task is called
  - Example

```
$display("%0d : \n  Calculated %0d, Expected %0d", $time, sum, calc_cum);
```

- **$monitor**(…) task
  - Writes formatted message to simulator display whenever any monitor inputs change in value (except $time)

```
$monitor("%0d : \n  Calculated %0d, Expected %0d", $time, sum, calc_cum);
```

Example 23

Exercise 6

Please go to Exercise 6
in the Exercise Manual

Introduction to Verilog

Summary

Summary

- Verilog is a hardware description language used to model hardware
  - This was an introduction
    - The specification has 560 pages!
- Basic building block is the **module** statement
- All objects must have a defined data type whether net or variable
- Two types of assignments are the continuous and the procedural assignments
- RTL is synthesizable behavioral Verilog code
- When writing RTL code, the two types of procedural blocks are combinational and sequential

Introduction to Verilog

Appendix

User-Defined Primitives (UDP)

- Allows users to define custom Verilog primitives
- Defined using truth tables
- Supports both combinatorial and sequential logic
- UDPs instantiated exactly like built-in primitive
- Characteristics
  - Only 1 output
  - Must have at least 1 input but no more than 10
  - Must appear outside of module definition

UDP – Mux (Combinatorial)

```
primitive mux (mux_out, sel, ina, inb);  // combinatorial
   output mux_out;
   input sel, ina, inb;

   table
      // sel  ina  inb  mux_out
           0    0    ? :     0;  // '?' means don't care
           0    1    ? :     1;
           1    ?    0 :     0;
           1    ?    1 :     1;
           x    0    0 :     0;
           x    1    1 :     1;
   endtable
endprimitive
```

Note: Order of ports in table determined by declaration order, not from comment. Inputs are listed first followed by : and then output

UDP – Latch (Level-Sensitive)

```
primitive latch (q, gate_n, data); // level sensitive, active low
    output reg q;
    input gate_n, data;

    initial q = 1'b0; // Output is initialized to 1'b0.
                      // Change 1'b0 to 1'b1 for power up Preset
    table
       // gate_n data current_state next_state
             0      1          :?:          1;
             0      0          :?:          0;
             1      ?          :?:          -;      // '-' = no change
    endtable
endprimitive
```

UDP – Register (Edge-Triggered)

```
primitive d_edge_ff (q, clock,data); //edge triggered, active high
    output reg q;
    input clock, data;

    initial q = 1'b0;    //Output is initialized to 1'b0.
                         //Change 1'b0 to 1'b1 for power up Preset

    table
    // clk data state next
        (01)   0    :?:    0;
        (01)   1    :?:    1;
        (0x)   1    :1:    1;
        (0x)   0    :0:    0;
        (?0)   ?    :?:    -;  // ignore negative edge of the clock
          ?    (??) :?:    -;  // ignore data changes on clock levels
    endtable
endprimitive
```

Timing Specifications

Specparam

- Specparam - assigning a value to a symbolic name for a timing specification
  - Similar to parameter but used in specify blocks

```
specify

        specparam  a_to_b  =  5;

                a  =>  b  =  a_to_b;

end specify
```

Rise, Fall, Turn-off and Min/Typ/Max Values

```
specify

        specparam  rise  =  4:5:6;
        specparam  fall  = 6:7:8;
        specparam  turnoff  = 5:6:7;

                a  =>  b  = (rise, fall, turnoff);

end specify
```

Timing Checks

- **$setup** task - system task that checks for the setup time

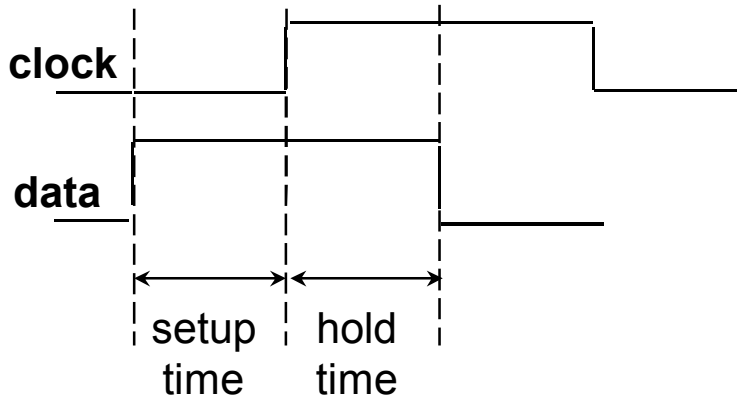
```
$setup(data_event, reference_event, limit);
```

  - data_event - monitored for violations
  - reference_event - establishes a reference for monitoring the dat_event signal
  - limit - minimum time for setup

- **$hold** task - system task that checks for the hold time

```
$hold(reference_event, data_event, limit);
```

  - reference_event - establishes a reference for monitoring the dat_event signal
  - data_event - monitored for violations
  - limit - minimum time for hold




```
specify
        $setup (ina, posedge clk, set);
        $hold (posedge clk, ina, hld);
        $setup (inb, posedge clk, set);
        $hold (posedge clk, inb, hld);
endspecify
```

Gate Delays

- Rise Delay: transition from 0, x, or z to a 1
- Fall Delay: transition from 1, x, or z to a 0
- Turn-off Delay: transition from 0, 1 or x to a z

```
<module_name> #(Rise, Fall, Turnoff) <instance_name> (port_list);
```

```
and   #(2)          u1 (co, a, b);          // Delay of 2 for all transitions
and   #(1, 3)       u2 (co, a, b);          //  Rise = 1, Fall = 3
bufif0 #(1, 2, 3) u3 (out, in, enable); // Rise = 1, Fall = 2,Turn-off = 3
```

Min/Typ/Max Values

- Min Value: the minimum delay that you expect the gate to have
- Typ Value: the typical delay that you expect the gate to have
- Max Value: the maximum delay that you expect the gate to have

#(Min:Typ:Max, Min:Typ:Max, Min:Typ:Max)

```
and   #(1:2:3)                    u1 (co, a, b);
and   #(1:2:3, 1:2:3)             u2 (co, a, b);
bufif0  #(2:3:4, 2:3:4, 3:4:5)    u3 (out, in, enable);
```