Chapter 4

Behavioral Hardware Description Languages

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟編製

Behavioral HDL

- To efficiently accomplish the verification task, you must well-versed in behavioral descriptions.
- To reliably and correctly use the behavioral constructs of Verilog, it is necessary to understand the side effects of the simulation algorithm and the limitations of the language – and ways to circumvent them.

RTL Coding Guidelines to Avoid Undesirable Hardware Structures

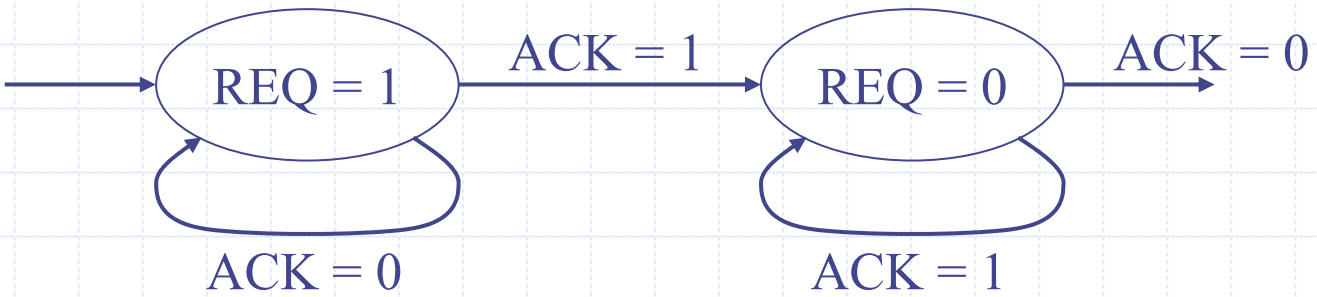
- To avoid latches, set all outputs of combinational blocks to default values at the beginning of the block.
- To avoid internal buses, do not assign *regs* from two separate *always* blocks.
- To avoid tri-state buffers, do not assign value 1'bz.

RTL Coding Guidelines to Maintain Simulation Behavior

- All inputs must be listed in the sensitivity list of a combinational block.
- The clock and asynchronous reset must be in the sensitivity list of a sequential block.
- Use a non-blocking assignment when assigning to a *reg* intended to be inferred as a flip-flop.

A Simple Handshaking Protocol Example

- It detects that an acknowledge signal (ACK) is asserted after a requesting signal (REQ) is asserted.
- Once ACK is detected, REQ is de-asserted, and it then waits for ACK to be de-asserted.



Advantages of Behavioral Models

- Faster to write
- Simpler, requiring less effort to ensure that it is correct
- Higher simulation performance

You Gotta Have Style!

- Write maintainable, robust code.
- Invest time now, save support time later.
  - Time invested in writing better code would be saved much time over in subsequent support efforts.
- Good comments improve maintainability.

Optimize the Right Thing

- Saving lines actually costs money.
  - There is no economic reason to reduce the number of lines of code. Unless, it also improves the maintainability.
- Optimizing performance costs money.
  - Performance optimization usually reduce maintainability and must be done only when absolutely required.

Structure of Behavioral Code

- Structuring code is the process of allocating portions of the functionality to different modules or entities.
- For maintainability reasons, behavioral code is structured according to functionality or need.

Available constructs for structuring code

VHDL	Verilog
Entity and architecture	Module
Function	Function
Procedure	Task
Package	Module

Encapsulation of Behavioral Code

- Encapsulation is an application of the structuring principle.
- The idea behind encapsulation is to hide implementation details and decouple the usage of a function from its implementation.
- The simplest encapsulation technique is to keep declarations as local as possible.
  - Avoid accidental interactions with another portion of the code where the declaration is also visible.

An Encapsulation Example

```
always
begin: block_1
      integer i;
      for (i=0; i<32; i=i+1) begin
                  …
      end
end

always
begin: block_2
      integer i;
      for (i=15; i>=0; i=i-1) begin
                  …
      end
end
```

```
task send;
      input [7:0] data;
      reg parity;
begin
      …
end
endtask

function [31:0] average;
      input [31:0] val1;
      input [31:0] val2;
      reg [32:0] sum;
begin
      sum=val1+val2;
      average=sum/2;
end
endfunction
```

Encapsulating Useful Subprograms

- Example: error reporting routines
  - To have a consistent error reporting format, a set of standard routines are used to issue messages during simulation.
  - They are implemented as tasks, with two packaging alternatives.

Packaging Technique I

- Put the tasks in a file to be included via a compiler directive within the module where they are used.
- Advantage
  - It can be used in synthesizable code whereas the other cannot.
- Disadvantages
  - The package cannot be compiled on its own since the tasks are not contained within a module.
  - Since the tasks are compiled within each module where it is included, it is not possible to include global variables, such as an error counter.

Packaging Technique II

- Put the tasks in a module to be included in the simulation, but never instantiated within any of the modules where they are used. Instead, an absolute hierarchical name is used to access the task in this global module.
- It can be compiled on its own since the tasks are now contained within a module.
- It is also possible to include global variables, such as an error counter.

Data Abstraction

- Synthesizable models are limited to bits and vectors.
- Work at the same level as the design under verification.

Real Values

- Constant could be defined using `*define* symbols.
  - `*define* symbols are global to the compilation and violate the data encapsulation principle.
- Defining them as parameters is better.
  - They would be local to the module.

An Example – Real Values

- Using `*define* symbols

```
`define a0      0.500000
`define a1      1.125987
`define a2      -0.097743
`define b1      -1.009373
`define b2      0.009672
```

- Using parameters

```
parameter       a0=0.500000,
                a1=1.125987,
                a2=-0.097743,
                b1=-1.009373,
                b1=0.009672;
```

Limitation

- Real values cannot be passed across interfaces.
  - Tasks, functions, and modules cannot accept a real value as one of its input arguments.
- Verilog provides a build-in system task to translate a real value to and from a 64-bit vector: *$realtobits* and *$bitstoreal*, respectively.

Records

- A module can emulate a record by containing only register declarations.
- When instantiated, the module instance emulates a record register, with each register in the module becoming a field of the record instance.

An Example – Records

```
module atm_cell_type;

reg [11:0] vpi;
reg [15:0] vci;
…
reg [7:0] payload [0:47];

endmodule
```

```
module testcase;

atm_cell_type cell();

initial
begin
    integer i;
    cell.vci=0;
    …
    for (i=0; i<48; i=i+1) begin
        cell.payload[i]=8'hFF;
    end
end
endmodule
```

Limitation

- The record is not a single variable such as a register.
  - It cannot be assigned as a single unit or aggregate, nor in an expression.
- By using a technique similar to the one build-in for the real numbers: conversion functions between records and equivalent vectors.

Records – An Alternative Implementation Technique

- If records are not nested.
- By using a vector composed of the concatenated fields
- The fields are declared and accessed using compiler symbols.
- Advantage
  - The records are true objects, thus can be passed through interfaces, or used in expressions.

An Example – Records

In file "atm_cell_type.vh"

```
`define ATM_CELL_TYPE [53*8:1]
`define VPI [12:1]
`define VCI [28:13]
…
`define PAYLOAD_0 [47:40]
…
`define PAYLOAD_47 [423:416]
```

In file "testcase.v"

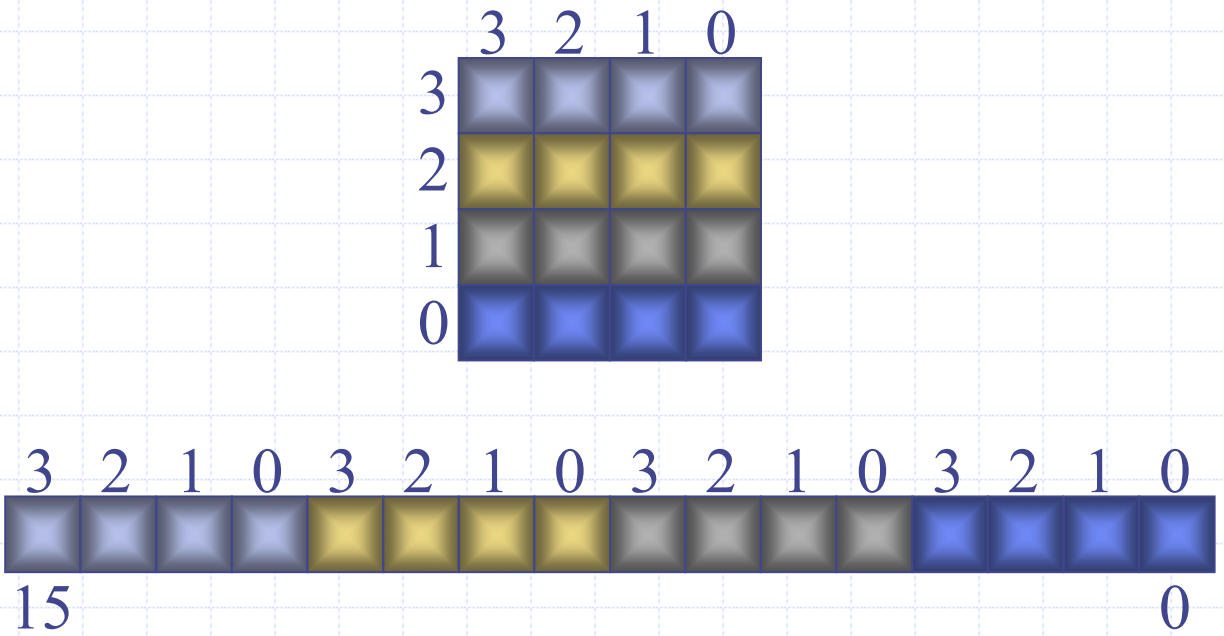
```
module testcase;

`include "atm_cell_type.vh"

reg `ATM_CELL_TYPE actual_cell;
reg `ATM_CELL_TYPE expect_cell;

initial
begin
        …
        // Receive the next ATM cell
        receive_cell(actual_cell);
        // Compare against expected one
        if (actual_cell != expect_cell) …
        // Increment the VPI field
        actual_cell`VPI=actual_cell`VPI+1;
        …
end
endmodule
```

Mapping a Array to a Linear Memory



location = 4Y + X

Multi-Dimensional Arrays

- Two-dimensional arrays
  - Use a memory of vectors.
  - The memory implements the first dimension.
  - The vectors in the memory implement the second one.
- Multi-dimensional arrays can be mapped onto a single dimensional structure.

Multi-Dimensional Arrays (cont.)

- A function and a task can be used to look-up and assign the array, respectively.
- A function and a task should be contained to convert the memory to and from a vector if the array needs to be passed through interfaces.

Issues

- How large should the memory be?
- What if you need two different array sizes in the same simulation?
- What about being able to use it in a subsequent project?

Lists

- Lists use memory more efficiently than arrays.
- Lists can be used to model large memories.
- Only the sections of the memory currently in use to be modeled.
- A linked list can be used to model a sparse memory.
- Lists can be implemented using an array.

Lists (cont.)

- Verilog does not directly support dynamic memory allocation and pointers or access values.
- There is a dynamic memory model PLI package provided by Cadence.
- This PLI package provides PLI routines that implement a sparse memory model using hashed linked lists.

Files

- External input files complicate configuration management.
- Many use files to initialize Verilog memories.

```
reg [7:0] pattern [0:55];
initial $readmemh(pattern, “pattern.memh”);
```

Files (cont.)

- If the file always contains the same data for the same testcase, it can be replaced with an explicit initialization of the memory in the Verilog code.

```
reg [7:0] pattern [0:55];
initial
begin
        pattern[0]=8'h00;
        pattern[1]=8'hFF;
        …
        pattern[55]=8'hC0;
end
```

Advantages and Disadvantages

- Verilog can only read binary and hexadecimal values.
- External files can eliminate recompilation.
- Files can program bus-functional models.

Concepts in HDLs

- Connectivity
  - Construct a design by connecting simpler blocks.
- Time
  - Represent how the internal states evolves over time.
- Concurrency
  - Describe actions occurring simultaneously.

Examples of HDLs

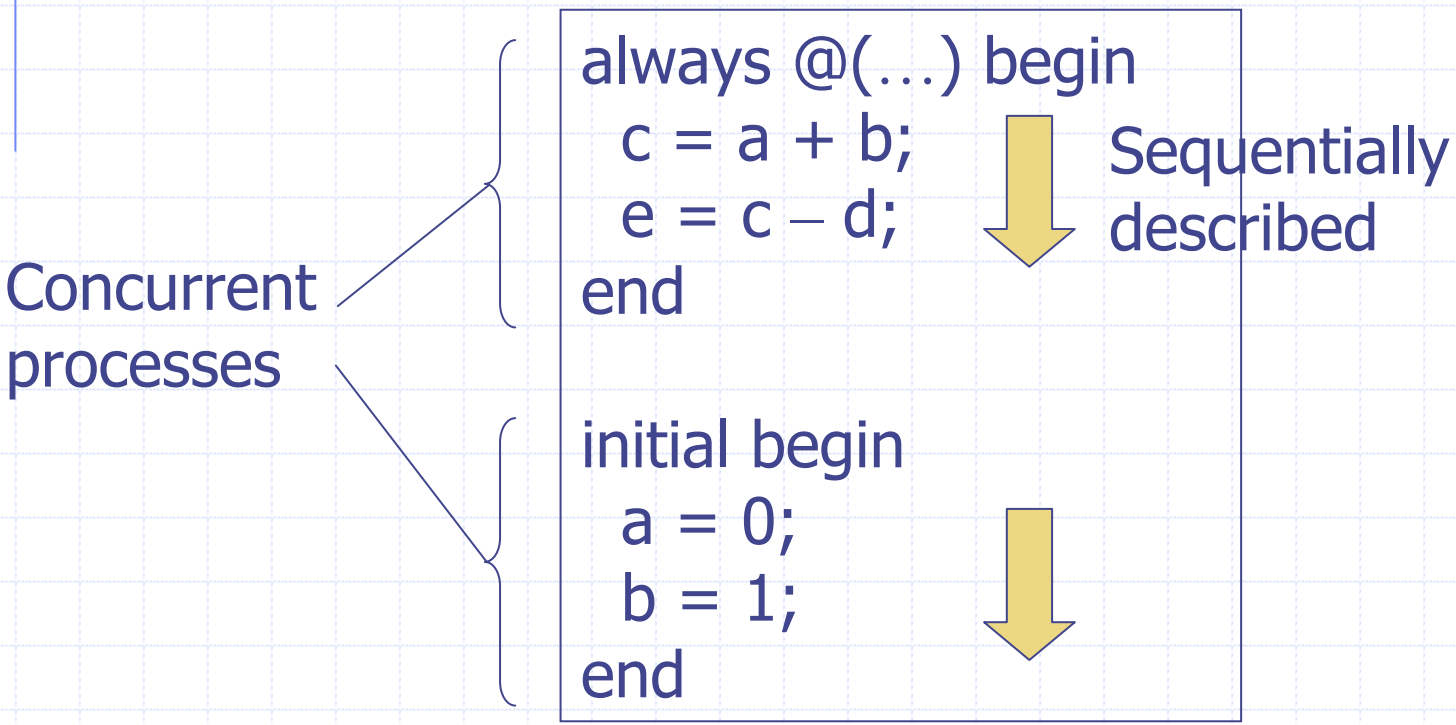
- Verilog
- VHDL
- SystemC: extend C/C++ to include connectivity, time and concurrency.

Problems with Concurrency

- You write better testbenches when understanding concurrency.
- Two problems:
  - Describe concurrent systems
  - Execute them

Describing Concurrent Systems

- Hybrid approach: concurrent processes (*always* and *initial* blocks in Verilog) described sequentially.

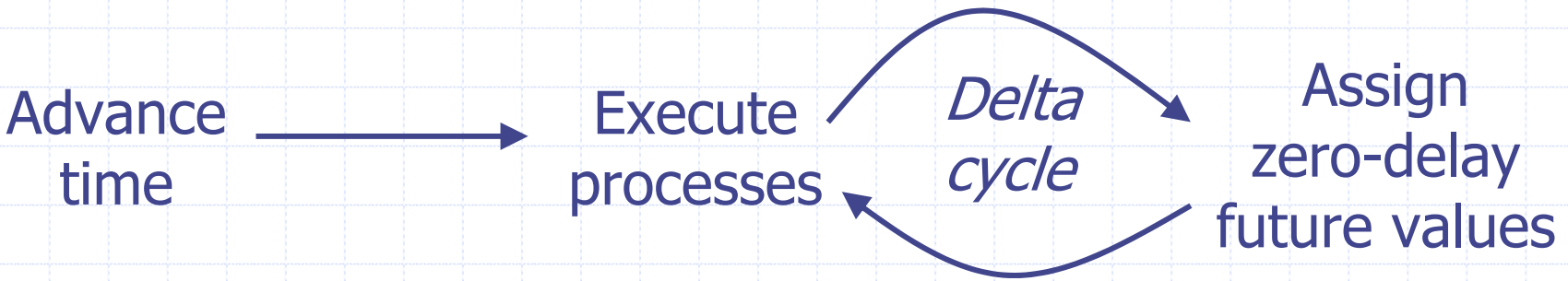


Executing Concurrent Systems

- Emulating parallelism on a sequential processor
  - Like a time-sharing OS
  - But has no time slice limit; process executes until a wait

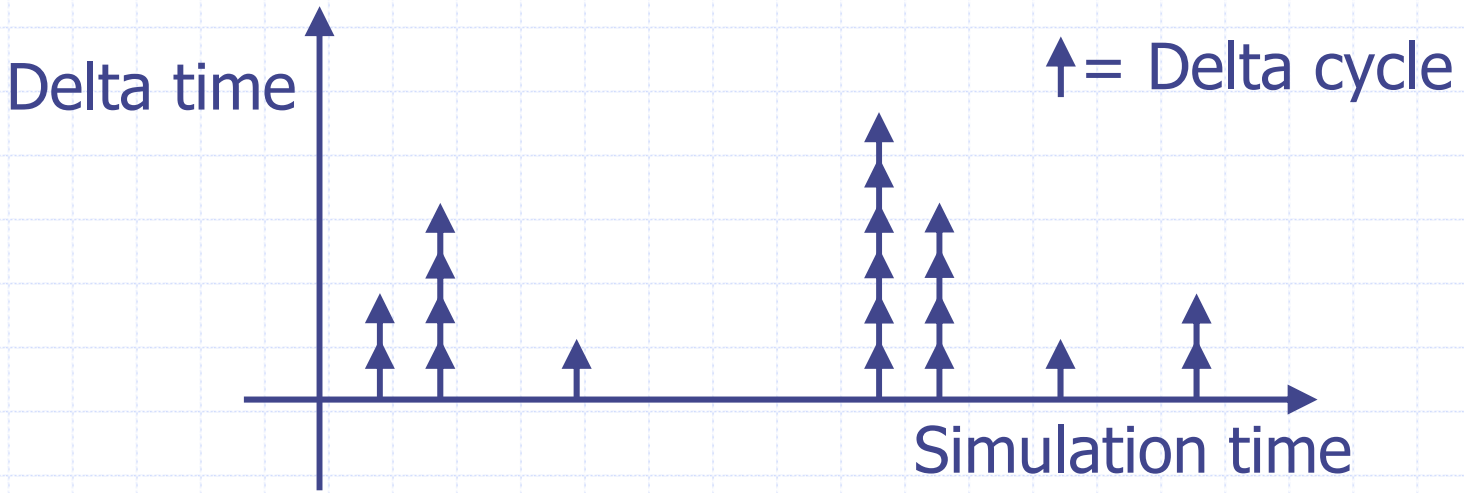
Simulation Cycle

- For each time instant
  1. Execute all processes scheduled at current time.
  2. Assign zero-delay future values (from non-blocking assignments).
  3. Schedule processes that are sensitive to the new values and return to step 1.



Time Progression

- Zero-delay cycles are called delta cycles.
- Simulators do not increment time step by step.



An Example About Concurrency in Verilog

```
reg R;
initial begin
  R = 1'b0;
  R <= #10 1'b1;
  #10;
  if (R == 1'b0) $wrirte("R is 0");
end
```

Will "R is 0" be printed?

Parallel vs. Sequential Description

- Use sequential description as much as possible.
  - Easier to understand and maintain
- Misuse of parallel description

```
reg clk;
initial clk = 1'b0;
always #50 clk = ~clk;
```

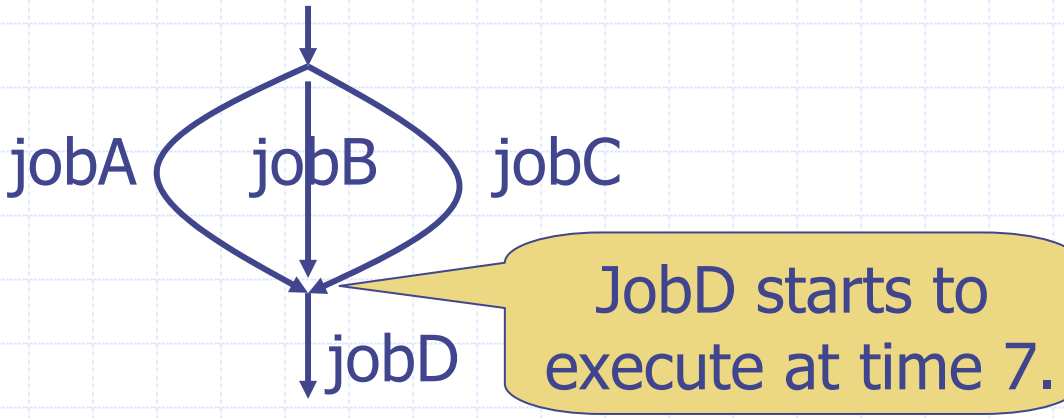
Misuse of concurrency

```
reg clk;
initial begin
   clk = 1'b0;
   forever #50 clk = ~clk;
end
```

Fork/Join Statement

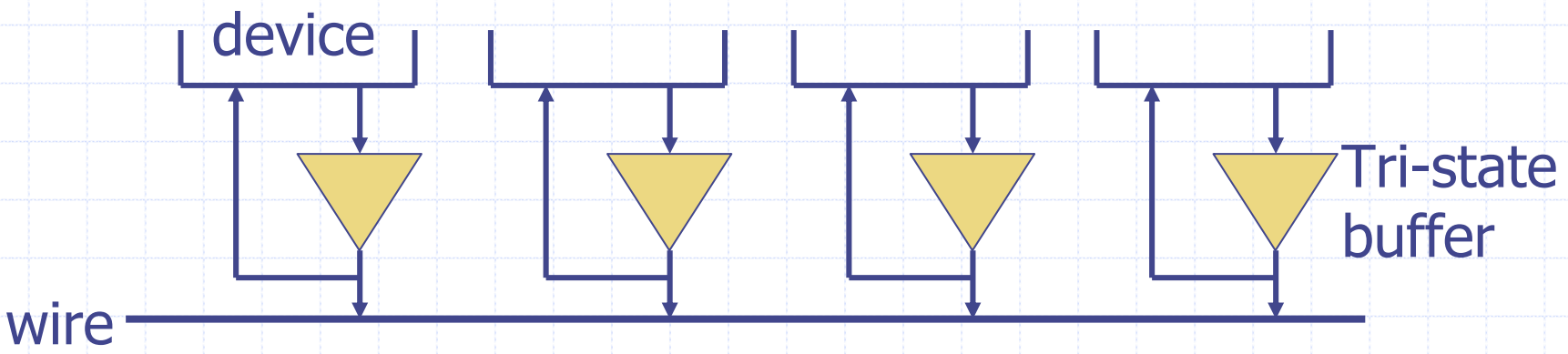
- Series of sequential and concurrent control flows.
- The sequential execution resumes after *join*, once all the concurrent regions are complete.

```
initial begin
  fork
    #5 jobA;
    #7 jobB;
    #3 jobC;
  join
  jobD;
end
```



Assigning vs. Driving

- Assign values to a register variable.
  - The last assignment determines the value.
- Each device drives a value onto a wire.
  - All the individual values being driven on the wire determines the final logic.



Verilog Portability Issues

- Two compliant simulators can produce different results
  - Because of unspecified situations in the IEEE standard

Read/Write Race Conditions

- Two concurrent blocks attempt to read and write the same register in the same timestep.
- However, the execution order among blocks is non-deterministic.

```
always @(posedge clk)
  count = count + 1;

always @(posedge clk)
  $write ("count = %d", count);
```

Assume *count* currently is 3. What value will be displayed once *clk* incurs a positive edge event?

To Solve Read/Write Races

- Use non-blocking assignment.

```
always @(posedge clk)
  count <= count + 1;

always @(posedge clk)
  $write ("count = %d", count);
```

Assume *count* currently is 3. What value will be displayed once *clk* incurs a positive edge event?

Write/Write Race Conditions

- Two concurrent blocks attempt to write to the same register at the same timestep.
- Non-blocking assignment does NOT solve this problem.

```
always @(posedge clk)
  count <= count + 1;

always @(posedge clk)
  count <= count - 1;
```

Initialization Races

- Initial blocks are NOT necessarily executed first
  - When the simulation is started, the *initial* and *always* blocks are executed one after another, in any order.

```
always @(posedge clk)
  $write("block #1 at %t", $time);

initial clk = 1'b1;

always @(posedge clk)
  $write("block #3 at %t", $time);
```

Guidelines for Avoiding Race Conditions (4 rules)

- If a register is declared outside of the *always* or *initial* block, assign to it using a non-blocking assignment. Reserve the blocking assignment for registers local to the block.

Guidelines for Avoiding Race Conditions

- Assign to a register from a single *always* or *initial* block.
- Use continuous assignments to drive inout pins only. Do not use them to model internal combinational functions. Prefer sequential code instead.
- Do not assign any value at time 0.

Events from Overwritten Scheduled Values

- Avoid overwriting previously scheduled values using non-blocking assignments.

```
always @(strobe) begin
  $write("strobe = %b", strobe);
end

initial begin
  strobe = 1'b0;
  strobe <= #10 1'b1;
  strobe <= #10 1'b0;
end
```

Schedule a 1 for *strobe* at time 10 and then a 0.
Does *strobe* receive an event?

Disabled Scheduled Values

- Avoid disabling a block where non-blocking assignments are performed.

```
always begin: if_logic
  …
  data <= #(10) read_val;
  …
end

always wait (reset == 1'b1) begin
  disable if_logic;
  wait (reset == 1'b0);
end
```

What will happen if *if_logic* is disabled when this line has been executed but *read_val* has not been assigned to register *data*?

Output Arguments on Disabled Tasks

- Disable the internal *begin/end* block inside the task instead of the task itself.

```
task read;
  output out;
begin
  …
  out = data;
  disable read;
end

always begin
  …
  Read(actual);
  …
```

Does *out* will be copied to *actual* when task *read* is aborted by disable?

```
task read;
  output out;
Begin: read_cycle
  …
  out = data;
  disable read_cycle;
end
```

New version

Non-Reentrant Tasks

- The same memory space is used for all invocations of a task.

```
task write;
input [7:0] wadd;
input [7:0] wdat;
begin
  addr <= wadd;
  @ (posedge rdy);
  data <= wdat;
end
```

The memory space is allocated at compile time.

Guarding Non-Reentrant Task Using a Semaphore

- A semaphore can give a message when we misuse a non-reentrant task.

```
task write;
input [7:0] wadd;
input [7:0] wdat;
reg in_use;
begin
  if (in_use === 1'b1) $stop;
  in_use = 1'b1;

  addr <= wadd;
  @ (posedge rdy);
  data <= wdat;
end
```