Name: Hoàng Phúc

MSSV: 1752099

**Report 3rd week-Verilog**

Chapter 2:

1. Module
   1. To declare a new module:

module module\_name (port name, port name, ....);

<...>

Endmodule

Ex:

module data\_conv(a,b,..);

input[3:0] a;

inout b;

output [15:0] g;

endmodule

* 1. Data type:

Inputs have to be “wire” data type

1. Instances: a known module’s object

Chapter 3:

1. Lexical convention:

+ Number: <size>’<base\_format><number>

+ String: variables of ‘reg’ type

+ System task & function: $display, $monitor, $stop

+ Compiler directive: `define, `include, `timescale 1ns/100ps

+ Data types: ‘wire’ and ‘variable’

+ Parameters: 2types: module parameter (parameter/localparam) and specify parameter

\* ‘parameter’ : can be changed from outside of the module

\* ‘specify parameter’: is used in a specify block

Chapter 4: STRUCTURAL MODELING

=>Structural models are built from primitive gate, switches, and other modules

1> User-defined primitives = > using true table

2> Example

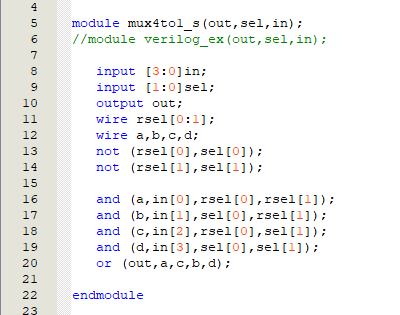


Figure 1: Mux 4 to 1 module

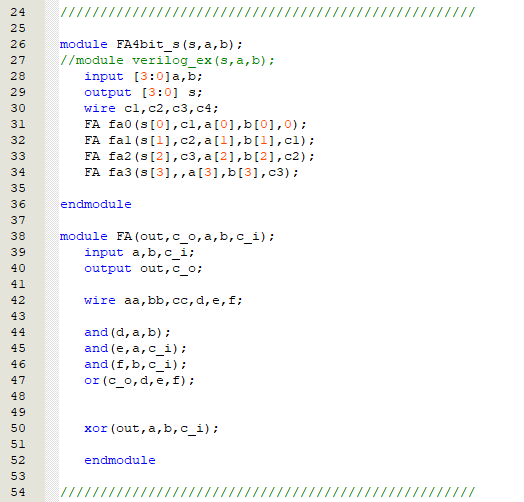


Figure 2: Full adder module

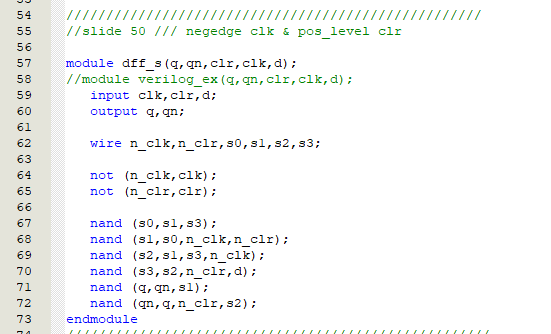


Figure 3: D flip-flop with negative edge clock & positive level clear

Chapter 5: DATAFLOW MODELING

=> Dataflow model: using expression instead of primitive gates

1> Continuous assignment:

+ assignment wire = wire/ reg/ function call (\*LHS NEVER is REG)

Example : assign out = i1 & i2;

+ common error:

Not assigning a wire to a value( value : variable, number, net,..)

Assigning a wire to a value more than 1

2> Operator:

1. Arithmetic op: +, -, \*, /, %, \*\*(any operand bit value = x => entire result value = x)
2. Logical op: ! , &&,|| (Return only 1 bit , treat all values that are nonzero as “1”)
3. Logical and case equality op:

* ===, !== (evaluating x and z);
* ==, != (evaluating only 1,0); result = 0 , 2 operands have the same pos is different in 0 and 1, if 0,1 compare to x, z => result = x

1. Relational op: <,>,<=,>= ( the result = x if the operands have x/z)
2. Bit-wise op: ~,&,|,^,^~:\

Special case: ~z=x , (0,1,x,z) & z = x , (0,1,x,z) | z = x

1. Unary reduction op: &,~&,|,~|,^,^~ : Example ^b ( b= 1010) => 1^0^1^0 = 0
2. Shift Op: >>,<< (unsigned data type - the vacated bit pos = 0),>>>,<<< (signed data type – >>> will fill with the value of the sign bit )
3. Concatenation op: {}
4. Replication op: {n{m}} (n must be const)
5. Conditional op: cond\_expr ? true\_expr : false\_expr

3> Example:

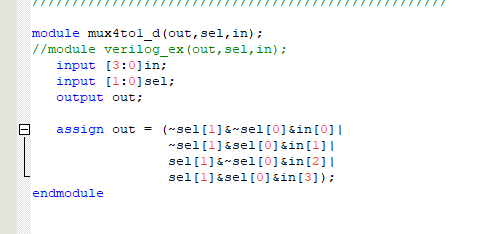


Figure 4: mux4to1 module

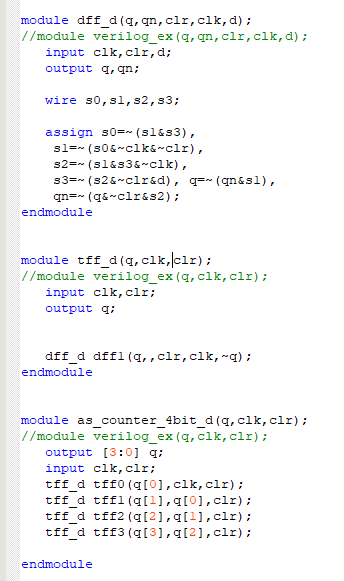


Figure 5: Asynchronous counter 4bit module

Chapter 6: BEHAVIORAL MODELING

1> initial & always construct

- initial & always can be nested

- initial executes ONCE,

- always executes REPETITIVELY

- they contain behavioral statements like: if..else,case,loop statements.

- Always construct: always may be followed by event control (@), delay control (#), wait statement (use $stop/$finish(system task) to halt the simulation)

2> Procedural assignment

* Difference bet CA and PA:

CA drive ‘wire’, are evaluated and update whenever an input operand changes value

Procedural assignment update the value of variable under the control of the procedural flow constructs that surround them.

* PA has 2 types : blocking ( = )& non-blocking ( <= )
* Blocking PA: used to describe the comb. logic
* Non-BPA: used to describe the seq. logic
* Begin…end: seq. block , Fork…join: parallel block
* Procedural continuous assignment: assign/deassign (LHS = variable), force/release (LHS = variable, wire)
* The assign PCA statement will override all PA to a variable
* The value of the var will remain the same until the var is assigned a new value through a PA or a PCA

3>Statement : If...else, case() , for , forever , while, repeat.

4>Procedural timing control : #,@(edge-sensitive) ,wait ( level sensitive)

5>Race condition => two signals racing each other to influence the output first

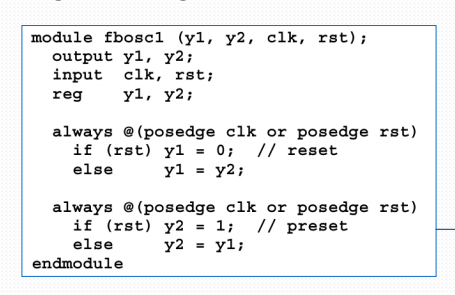


Figure 6: race condition example

6> Coding guidelines

* When modeling seq. logic 🡪 nonblocking assignment.
* When modeling comb. logic 🡪 blocking assignment.
* Modeling both seq. & comb. logic in the same block => nonblocking assignment.
* DO NOT mix blocking & nonblocking assignment in the same always block
* DO NOT make assignment to the same variable from more than 1 always block.

7>Modeling combinational logic tips

Every element of the input must be in the sensitivity list ( use @(\*)

The combinational output must be assigned in every control path

8>Modeling seq. logic tips

* Latch == comb circuit => blocking style
* If event-expression contains posedge or negedge, ff (register) will be synthesized.
* A variable assigned within an always @ block that is not fully specified will result in latches synthesized.
* In all other cases, combinational logic will be synthesized.

Chapter 7: Funcs & Tasks

1) Functions

-Functions cannot include timing delays

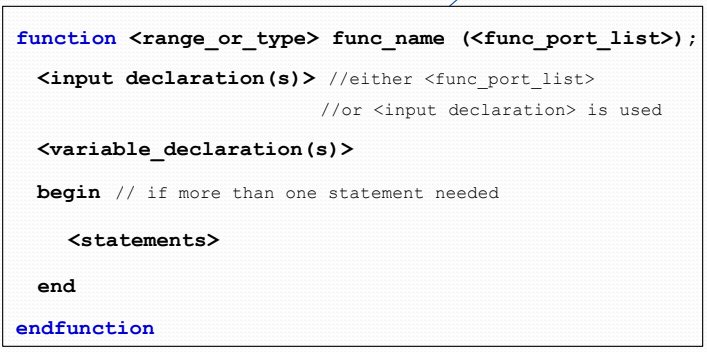
-Funcs have many input & 1 output

-Can be used for modeling combinatonal logic

-Functions can call other 'funcs' but can not call 'tasks'

\*\*internal implicit reg == the same name of the function =>return value == implicit reg

To declare a function:



\*\* <range of type> defines width ( default bit width =1 ) and type ( integer, real, time, realtime) of implicit reg

2) Function rules

-not contain delay

-not enable tasks

-not any nonblocking assignments/PCA

-not 'event triggers'

-has at least 1 argv

3) Tasks

-> tasks can be called many times ,reducing code repetition

-> can including timing delays

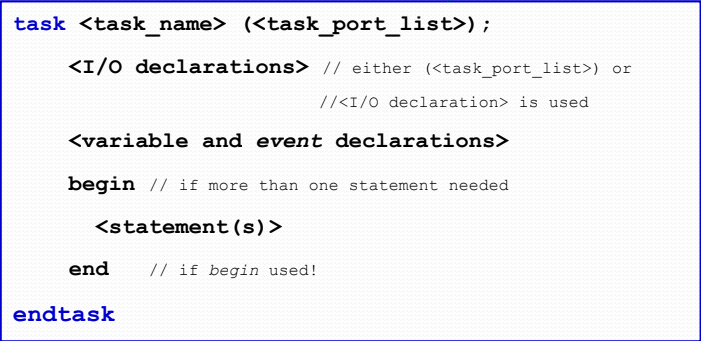
-> can call task or funtion

-> use for modeling both combinational and sequential logic

-> task must be specifically called with a STATEMENT, it cannot be used within an expression as a function can

-> task should be written in a separate file -> it can be used in multiple modules

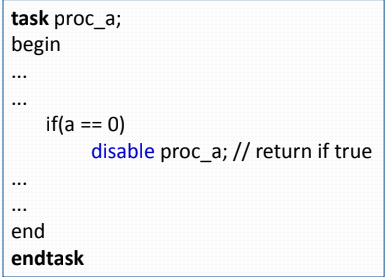
To declare a task:



\*\* input and inout argument are passed into the task

\*\* output and inout arguments are passed back to the invoking statement when task is completed

To disabling tasks



4> Difference between functions & tasks

a>Function:

* can call functions
* no delay
* no timing statement ( @,wait...)
* at least 1 input
* return only a single value

b>Tasks:

* call both func & task
* can include delay
* -allow timing statement
* -may have arbitrary I/O or inout
* -no return value

\*\*Task == any common verilog code

\*\*Functions are used when common code is purely combinational => are used for conversions and commonly used calculation

5>Sameness:

* Do not contain initial/always
* Contain only procedural assignment => called from initial or always statements
* Functions & Tasks can take, drive and source global variables, when no local variables are used.

6> System tasks & function: there are some categories as follows:

* display tasks
* file i/o tasks
* simulation control tasks
* math functions

Chapter 8: Finite state machines

1>Model FSM:

Decide how many states are needed, which transitions are possible from 1 state to another

Reset state

‘’always @ (posedge clock)” defines the state registers

Combinational logic dfines the next state and output logic

2>Tips on FSM:

Don’t forget to handle the ‘defualt’ for case statement

Should use 2 different ‘always blocks for next state and current state circuit

Outputs can be a mix of comb. and seq.

3>Example

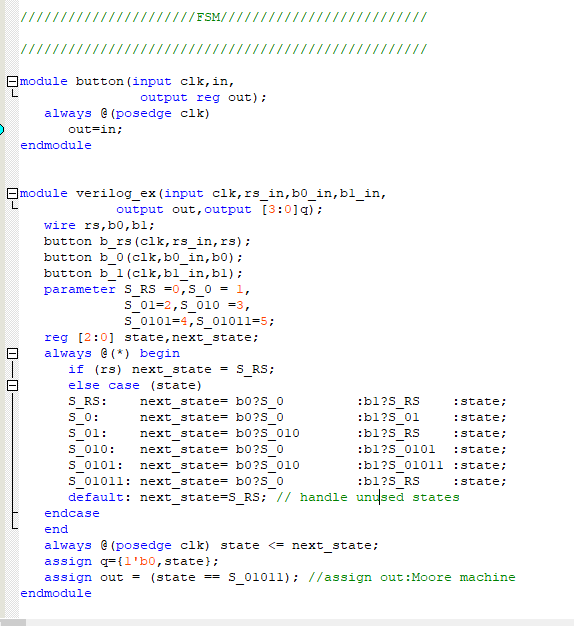


Figure 7: Example about FSM ( Moore machine)

Chapter 9: TESTBENCH & VERIFICATION

1>Pre-systhesis verification:

* Check design function flaws that may cause by ambiguous problems, designer errors, or incorrect use of parts in the design.
* Done by simulation, assertion verification with testbench definition or input waveform

Input waveform: is usually good for small design

Function verification with testbench:

=> Using Verilog language for testing design module -> testbench

=> testbench is a code for TEST, not a part of final design

=> timing & display procedures: important in designing testbench

2>Structure of testbench

In testbench, a dummy template which basically declares inputs/inouts to DUT ( design under test) as reg and outputs from DUT as wire ( \*\*\* NOTE: there is no port list for the testbench)

There are 2 styles of the testbench:

1. Style 1

* Develop your hierarchical system within a module that has input and output ports ( called “ design” here)
* Develop a separate module to generate tests for the module (“test”)
* Connect these together within another module (“testbench”)

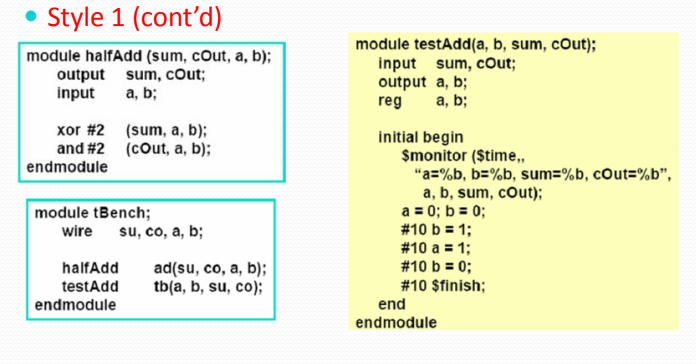


Figure 8: example about testbench’s style 1

1. Style 2

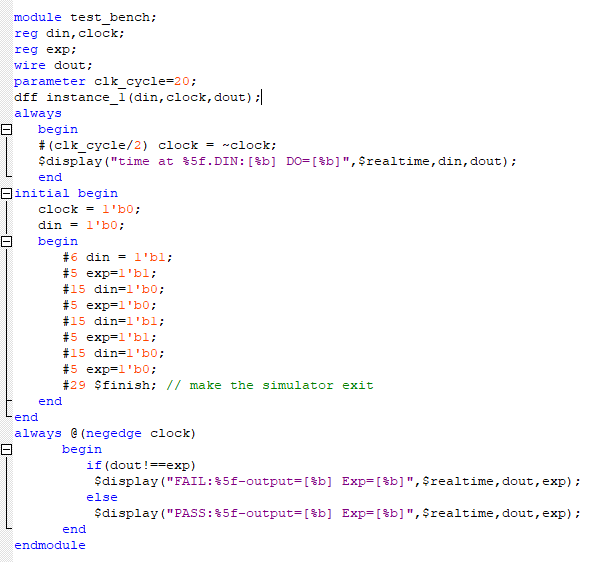


Figure 9: example about testbench’s style 2

3> Testbench techniques

1. Simulation control: $stop, $finish
2. Limiting data set: ( don’t understand)
3. Applying synchronized data:

Ex: initial forever @(posedge clock) #3 x = $random; ( what does difference to ‘always’ ?)

4> Test cases:

Must have the state transition matrix

Choose test data wisely: typical cases, all possible state change , corner cases

Many kind of bugs escape through RTL simulation test.

Should follow the flowchart below:

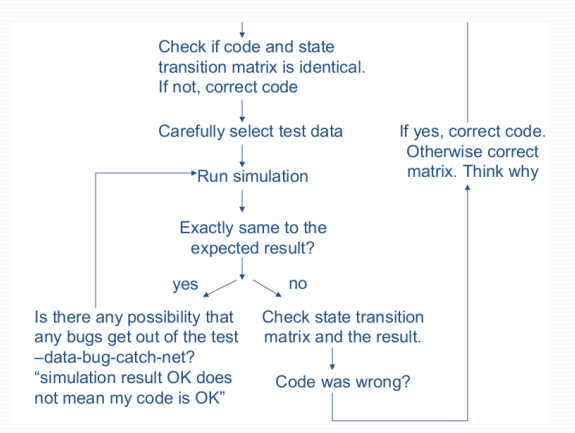


Figure 10: The suggestion flowchart for testing the design