

Test4 does not become ‘1’ at the beginning of clock cycle 3. It becomes ‘1’ at the beginning of clock cycle for because it takes time to travel through the flip-flop and to the location where it’s needed.

**Propagation Delay:**

Two ways to fix: decrease clock speed, break up logic into stages (pipeline)

**Blocking vs Nonblocking Statements:**

Verilog - “<=” Nonblocking, “=” blocking

Blocking means that this statement has to finish executing before any subsequent statements can be used.

**Verilog Procedures:** Code that exists in **always**, **initial**, **task**, or **function** blocks.

“**reg**” data types can only be used in these. For all other Continuous Assignments (called that because they’re always connected), we must use the “**wire**” data type

**Full-duplex vs. Half-duplex**

Full-duplex is when you have separate wires for data transmission and reception (UART). Half-duplex is when you only have one wire (I2C).

**Trio of Speed Metrics:**

High throughput (bits per second)

Low latency (time or clock cycles)

**Variable Types and Keywords**

|  |  |  |
| --- | --- | --- |
| Verilog | VHDL | Description |
| wire | signal | A register |
| assign | N/A | Continuous assignment - permanently assign output port to register |
| always | process | Used to create a block of code with a sensitivity list. Means that whenever the variable in the list changes, the block of code gets executed. |
| reg var\_name | signal var\_name : std\_logic | Register variable used to store data, can be driven |
| always @ (posedge clk) | if rising\_edge(clk) | Way to create a sequential logic block |
| 1'b0 | '0' | Syntax to assign bits |
| &, |, ^ | and, or, xor | bit-wise operator differences |
| reg [3:0] var\_name | signal var\_name : std\_logic\_vector(3 downto 0) | Register vectors |
|  |  |  |
| parameter | generic | A way to configure a module during initialization |
| {var1, var2, var3, ...} | var1 & var2 & var3 & ... | Concatenation of bits |
| module | entity |  |
| parameter | constant | Keyword for constants |
| Case Sensitive | Case Insensitive |  |
| input, output, inout | in, out, inout |  |

**Counters**

When making a counter, make it count down and have the count value be one bit longer than necessary. That way you can compare the MSB to determine whether the count has finished. This will reduce the logic during synthesis. In the example below, only one comparison is needed versus 5 (3 ='s and 2 and's) when using 3 bits. This benefit becomes even greater the larger the count value.

Ex.

Max Count - 7

0111

0110

0101

0100

0011

0010

0001

0000

1111

Normally: (bit 2 = 1) and (bit 1 = 1) and (bit 0 = 1)

Optimized: (bit 3 = 1)

**Finite State Machines**

Combinational Logic - Depends only on current inputs

Sequential Logic - Depends on current inputs and current state of the system (where it state could be a variable, state machine state, internal signals, etc)

Moore State Machine - The present state is the only thing that determines the output (combinational)

Mealy State Machine - Depends on present state as well as inputs (sequential)

**Coding Convention**

i\_ Input signal

o\_ Output signal

r\_ Register signal (has registered logic)

w\_ Wire signal (has no registered logic)

c\_ Constant

g\_ Generic (VHDL only)

t\_ User-Defined Type

**Verilog**

localparam vs parameter

- localparam cannot be redefined during the instantiation of a module

[size]'[base\_format][number]

Size: Number of Bits

Base\_format: d (decimal), h (hex), o (octal)

No size will default to size of variable being assigned this number.

Just a number will default to decimal.

Negative sign for negative numbers must go before <size>.

**Identifiers:**

Names of variables can only have alphanumeric characters, underscores, and dollar signs. Cannot start with $ or a number.

**Data Types**

**Nets:**

Connects hardware entities, basically a wire, so it stores no value. Most common type is **wire**.

**Variables:**

Like software variables, can store data. Represented by type **reg**.

**integer:**

Standard 32-bit wide variable, same as software

**time:**

Unsigned 64-bit variable that stores simulation time

**realtime:**

Stores time as a 64-bit floating point number instead of an integer

**real:**

Stores 64-bit floating point numbers

**Verilog Strings:**

Unlike SystemVerilog, they are stored in **reg** arrays, each character requires a byte of space. Upper bytes get truncated if not enough space, and extra space gets padded with zeros.

**Vectors:**

reg [MSB:LSB] - LSB and MSB must be constants, so you can't assign it to integers, only parameters

**Part-Select:**

For a vector, you can choose the range using [<start\_bit> +: <width>] or [<start\_bit> -: <width>].

The + version increments from the start bit, and the - version decrements from the start bit.

Ex. [8 +: 3] = [10:8]

[8 -: 3] = [8:6]

**Arrays:**

reg [MSB:LSB] var\_name [MSB:LSB]

Can only be used for **reg, wire, integer,** or **real**

Multidimensional arrays can be made by adding another vector: var\_name [MSB:LSB] [MSB:LSB]

**Modules:**

module <name> #([parameters\_list]) ([port\_list]);

end module

Ports by default are of type **wire**.

Ports by default are of type **unsigned** as well.

Unconnected ports are high Z in simulation. In synthesis, they are tied to ground.

Ports cannot be connected more than once.

Input and inout ports can only be of type **wire**.

Output ports can be wire, reg, wand, wor, or tri.

Connecting a smaller vector to a larger vector in a port will cause the larger bits of the vector to be ignored.

**Two Drivers, Same Wire**

**wire/tri Truth Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | 0 | 1 | x | z |
| 0 | 0 | x | x | 0 |
| 1 | x | 1 | x | 1 |
| x | x | x | x | x |
| z | 0 | 1 | x | z |

* **tri** and **wire** are essentially the same. All drivers connected to a **tri** **/wire** must be z, except one (which will determine the value of the **tri/wire**)

**wand/triand Truth Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | x | z |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | 1 |
| x | 0 | x | x | x |
| z | 0 | 1 | x | z |

* All assignments to these two will be anded together to produce the state of that wire.

**wor/trior Truth Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | x | z |
| 0 | 0 | 1 | x | 0 |
| 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x |
| z | 0 | 1 | x | z |

* All assignments to these two will be ored together to produce the state of that wire.

**Assign Statement**

**Rules**

* LHS must be a wire type
* RHS can have wire/register types or function calls.
* When partially assigning a wire, unassigned bits will be Z.
  + assign var[3:1] = 'b101, bit 0 is Z
  + var will equal 'b101Z
* When fully assigning a wire a smaller vector size, upper bits will be 0 padded.
  + assign var = 'b101, bit 3 is 0
  + var will equal 'b0101
* The LHS can be a concatenation of wires.
  + assign {var1, var2} = {var3, var4}

**Explicit Assignment**

* wire var;
* assign var = x;

**Implicit Assignment**

* wire var = x;

The above explicit and implicit assignments are equivalent.

**Basic Building Blocks**

**Half Adder**

* Sum = A xor B
* Cout = A and B

**Full Adder**

* Sum = A xor B xor Cin
* Cout = (A and B) or (Cin and (A xor B))

**Multiplexer**

* C = sel ? A : B

**Decoder**

* C = en ? 1 << in : 0
* Whatever number "in" is, a one will get shifted to that bit position and pushed out.

**Operators**

**Arithmetic Operators**: Plus (+), Minus (-), Multiply (\*), Divide (/), Modulus (%), Exponent (\*\*)

* If a divide or modulus by zero, the result is X.
* If either operator of the exponent is real, the result will be real

**Relational Operators**: Less Than (<), Greater Than (>), Less/Equal (<=), Greater/Equal (>=)

* If either side is X or Z, result is X.

**Equality Operators**: Logical Equality/Inequality (==)/(!=), Case Equality/Inequality (===)/(!==)

* Logical tests only 0s and 1s, Xs or Zs will cause result to be X
* Case tests will test for all for states 0s, 1s, Xs, and Zs, result will always be known

**Logical Operators**: And (&&), Or (||), Not (!)

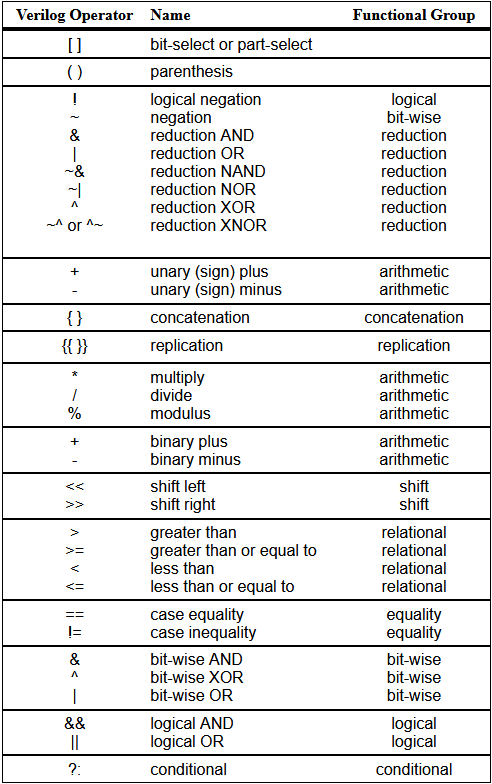
* Any Xs for the operands will result in an X

**Bitwise Operators**: And (&), Or (|), Xor (^)

* Result cannot have any Zs
* Xs and Zs count 0s or 1s in the calculation
* If a definitive result cannot be obtained when an operand is X or Z, then the output is X

**Shift Operators**: Logical Shift Left/Right (<</>>), Arithmetic Shift Left/Right (<<</>>>)

* Logical shift just pads bits with 0s
* Arithmetic shift will sign extend on the left side and pad with 0s on the right side

**Operator Table: Descending Precedence**

**­­­­­**

**Always Statement**

**always @ (event)**

* Whenever an event in the sensitivity list changes, the statements inside the always block will be executed.

**always #10 clk = ~clk;**

* Without a sensitivity list, always blocks will execute with 0 time delay.
* Thus, we need to add a delay ourselves.
* Delays are non-synthesizable and are only for testbench uses.

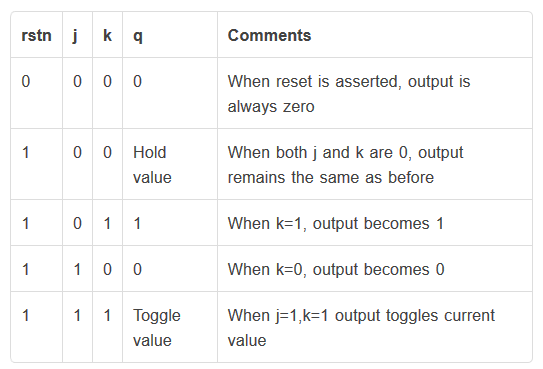
**always @ (\*)**

* Any register or wire that affects another register or wire will be included in the sensitivity list automatically.
* Ex. c = a + b, a and b will be placed in that list.
* Used mainly for combinational logic, so blocking statements (=) will be used most of the time in these blocks.

In general, always can be used for either sequential or combinational logic depending on how you set up the sensitivity list.

**Flip Flop**

**JK Flip Flop**



Toggle means the output value will switch between 0 and 1, unstable when it’s a JK Latch since there’s no time delay.

**Flip Flop vs. Latch**

* A latch is asynchronous while the flip flop is edge triggered.

**Initial Blocks**

* **initial** blocks are not synthesizable.
* Simulation only tool used for setup and driving design port
* $finish will kill all initial blocks that are not finished by ending the simulation.

**Module**

* Module names can contain alphanumeric characters and ‘\_’
* Reg can only be assigned in **initial** and **always** blocks
* Wire can only be assigned using the **assign** keyword
* Ports
  + <direction> <type> [3:0] <var\_name>
  + Direction: input, output, inout
  + Type: wire, reg (default wire)
  + All ports are unsigned by default

**Generate**

* Can have modules, continuous assignments, or always/initial blocks
* Can include if…else and case statements
* **generate… endgenerate**

**Generate For Loop**

* for (i = 0; i < 3; i++)
* Note: i++ is a blocking statement

**genvar**

* Can be used to declare variables to use in generate-for loops
* Stores positive integers and can be declared in or outside the generate block
* Can be assigned/changed during compilation and elaboration
* Ex. genvar i;

**Case Statement**

case (value)

val1 : begin…end

val2 : begin…end

val3, val4 : begin…end

…

default : begin…end

endcase

**Block Statement**

* Blocks are enclosed by **begin…end** keywords
* There are sequential and parallel blocks
* They can be named.
  + Ex. begin : <name>
  + Ex. fork : <name>

Sequential blocks are just standard, normal blocks. Parallel blocks can be formed by using **fork-join** keywords.

**Fork-Join**

* There are three type of fork-join statements: **fork-join, fork-join\_none, fork-join\_all**

**fork-join**

* All processes must finish before executing the statement after the block

**fork-join\_none**

* Kicks off the parallel processes without waiting for them to finish before executing the statement after the block

**fork-join\_any**

* Kicks of the parallel process. When one of them finishes, the statement after the block will be executed

**Assignment Types**

**Procedural Assignment**

* Occurs in procedures such as **always, initial, task,** and **functions**
* Holds value until next assignment
* Used for variables (**reg**)
* Values in an array cannot be initialized all during declaration at the same time, must be done individually. As such, array declaration assignments are not allowed. (Verilog only, Systemverilog allows this)

**Continuous Assignment**

* Used for nets (**wire**)
* LHS will change as soon as RHS changes
* Can be initialized in the same line as the declaration, must be done only once per net

**Procedural Continuous Assignment**

* **assign** and **deassign**
  + Continuous assignment for registers in a procedural block
  + Deassign will untie a register from an assigned value
* **force** and **release**
  + Continuous assignment for registers as well as nets, concatenations, bit-select
  + Release will untie any of the above from an assigned value
  + Simulation only use is recommended
  + Not for use on arrays
  + Overrides all other assignments to that net/register

**Note:** Any register that is deassigned/released will retain the previous value until a new one is assigned. Procedural continuous assignment is not synthesizable.

**Blocking and Non-Blocking Statements**

**Blocking Assignment**

* Prevents execution of next statement until current statement finishes
* Does not prevent other parallel blocks from executing
* Ex. Two initial blocks with blocking statements will not prevent each other from running

**Note:** $display is a blocking statement

**Non-blocking Assignment**

* Does not prevent execution of next statement until current one finishes
* Calculates the RHS of all statements for a particular time-step
* At the end of time-step, the RHS values will be assigned to LHS

**Control Blocks**

**If block**

* Pretty self-explanatory
* **if…else if…else**
* Must enclose in **begin…end** if multiple statements are in a block

**forever**

* As the name implies, statements inside a forever block will run forever

**repeat**

* **repeat (n) begin…end**
* Repeats the statements inside the block n times
* If expression evaluates to X or Z, it is treated as zero and the statement will not be executed

**while**

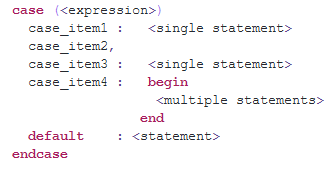
* **while (expression) begin…end**
* Pretty self-explanatory

**for**

* **for (i = 0; i < 5; i++) begin…end**
* Pretty self-explanatory
* Can be used to design a shift register

**case**

* Case statements synthesize into multiplexers
* If-else statements can synthesize into priority encoders if they have too many conditions (too many else-ifs)
* If none of the cases match and there is no default case, the block will exit without doing anything.
* Can nest case statements within case statements
* Case statements can involve Xs and Zs, and they need to match exactly (2’b10 != 2’b1x)
* Case statements provide definitive results when there are Xs and Zs

****

**Functions**

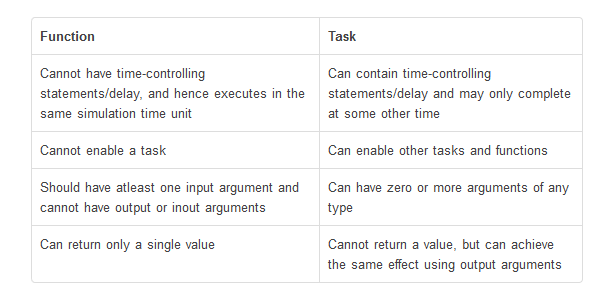
* **function [automatic] [return\_type] name (input [input\_list]);**
  + automatic makes the function **reentrant**
  + Normally, functions and tasks are static, so they share variables between different calls.
* Does not and cannot consume simulation time (#, @, wait, posedge, etc are illegal)
* Cannot call tasks
* Can only have blocking assignments
* Must have at least one input
* Cannot use **force-release** or **assign-deassign**
* Cannot have output, inout, or triggers (-> used in assertions)
* A function will automatically create an internal variable with the same name as the function. This variable, when assigned something, will store/output the return value.

**Reentrant Function**

* Cannot use static or global variables
* Function cannot modify its own code (pretty rare for my usage)
* Doesn’t call other functions that violate the above 2 rules
* Items declared in the function will be dynamically allocated every time the function is called
* Ex. Recursive functions

**Tasks**

* **task [automatic] name (input [input\_list], inout [inout\_list], output [output\_list]);**
  + Added inout and output lists
* Cannot return anything, but outputs serve the same purpose
* Doesn’t need input arguments



**Parameters**

* Parameters are the Verilog equivalent of generics. They are used to instantiate modules with different specifications.
* They are constants, so their value cannot be modified at runtime.
* If a parameter is assigned to another parameter, if the RHS parameter changes, the LHS will update accordingly.

**defparam**

* Modify module parameters after instantiation
* Common testbench strategy
* Ex. defparam [module\_instance\_name].FIFO\_DEPTH = 128

**Specify Parameters/Block**

* The specify block is used to define delays in a module.
* **specparam TODO**

**Compiler Directive**

**`ifdef <FLAG>… `endif**

* Can be nested in another `ifdef
* If <FLAG> is defined anywhere in all of the compiled files or given via the command line, then include statements within to be compiled

**`ifndef <FLAG>… `endif**

* Can be nested in another `ifndef
* If <FLAG> is not defined anywhere in all of the compiled files nor given via the command line, then include statements within to be compiled

**Note: `elsif** can be added to either of the above for more options.

**Timing Control**

**Delay Control**

* **#(delay\_value)**
  + Wait for delay\_value ns
* If delay\_value is U or Z, it will count as a 0, meaning no delay.
* If delay\_value is negative, it will be considered a 2’s complement unsigned integer, basically absolute value.

**Note: $realtime** is the system task for current simulation time.

**Event Control**

* Triggered by value changes on nets/variables

**posedge**

* **@(posedge var)**
* Posedge is defined as a change from (0 to X, Z, or 1) or (X or Z to 1)

**negedge**

* **@(negedge var)**
* Negedge is defined as a change from (1 to X, Z, or 0) or (X or Z to 0)

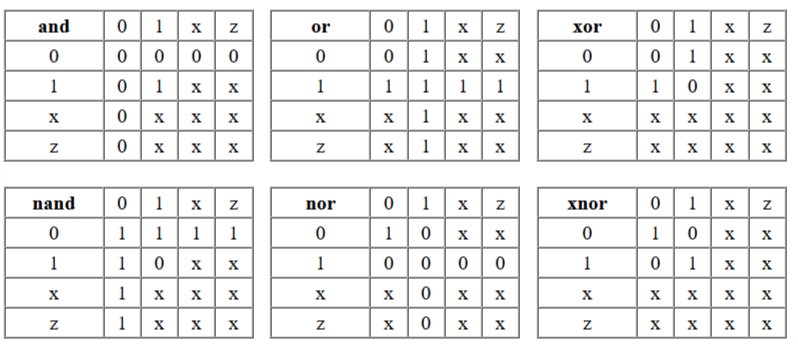
**Named Events**

* Using the **event** keyword, you can create your own event types.
* If the correct conditions are met, you can trigger the event using -> **<event\_name>**
* You should have another block looking for that event trigger and do something when that happens. Ex. always @ (<event\_name>)
* A delay before the arrow is allowed. This obviously makes this a testbench specific thing.
* Arrays of events are also possible: **event** b\_event[5];

**Event or operator**

* **always @ (posedge a or posedge b)**
* **always @ (posedge a, posedge b)**
* Both ways are acceptable of writing a list of events to trigger

**Gate Level Modeling**



Verilog has primitives that model gate level behavior.

**And**

* A

**TODO:**

* Look up specparam
* Look up specify block

**VHDL**

**Types of HDL Code**

Three types of HDL code exist: Behavioral, RTL, Gate-Level (structural)

Behavioral - Mimics the desired functionality of the hardware but not necessarily synthesizable. There is no strict rules as long as the code generates the desired behavior. Guideline is to keep it simple and readable. Behavioral are often used to represent analog block, place holder code (RTL/gates not ready), and testbench code. Example: clock generator, delay cells.

RTL - Register-Transfer-Level, an abstraction hardware functionality written with always blocks and assign statements that are synthesizable (can be translated into gate level). Pure RTL does not instantiate sub-modules. RTL could contain sub-modules to guide the synthesizer. Structural RTL (ofter still called RTL) is a module that contains other RTL modules. Example: FSM (Finite-State-Machine)

Gate-Level - Logic described by gates and modules only. No always blocks or assign statements. This is a representative of the real gates in the hardware.

**Alternative Definitions:**

Behavioral - the highest level of abstraction and when writing behavioural code you simply need to define the relationships between inputs and outputs without specifying anything about how those relationships will be implemented. Sometimes behavioural descriptions are too high level and cannot actually be synthesized into hardware. If you are doing a simulation and just need a block to behave in a certain way, then a behavioural model will be adequate.

RTL - fits in the middle. It specifically describes the relationships between inputs and outputs be describing how data moves between registers in the hardware. RTL descriptions are implementable in hardware. For this particular example, it is not very important to understand the nuances of the architecture type (behavioural, RTL, or structural), you just need to define it as something.

Structural - the lowest level of abstraction. When writing structural code, you describe how the low level structures (e.g., logic gates) connect together to give the system that you want. If you need precise control over the logic gates that will be created, a structural model is what you need.

**Gate-Level:**

Half Adder -

Sum = A xor B

Cout = A and B

**RTL Equivalent:**

process/always:

temp = A + B;

end process/always;

Sum = temp(0);

Cout = temp(1);

**Uncategoriable Tidbits**

* Zeros in FPGAs are faster to compare than ones.
* Static variables have a lifetime that is the length of the program
* Automatic variables are allocated and deallocated when the program enters and leaves that variable’s scope respectively.
* In C, all variables are automatic by default.