Simulation

Simulation is part of modelling; the accuracy of the model is refined as the design process progresses.

- Assume that high-level modelling has been done
  - Already know the architecture and the algorithms: models exist
  - Have block relationships but not exact (cycle accurate) implementation/timing
- In this module simulation is refined to verify:
  - The functionality of an RTL description
  - The correctness of a completed chip
  - The timing and electrical properties of the proposed product
  - Thermal too?

Caveat: The intention of this material is to give enough information to facilitate design. It is not intended to be a complete description of all the available facilities.

“Accuracy” vs. “Precision”

Scientifically ‘accuracy’ is usually used to indicate how representative a value is to physical reality. “Precision” is how repeatable a measurement is. If you measure something twice you may get different values; the question is how different? Repeated similar readings give a precise answer; however if the end was cut off the ruler they may not be very accurate!
Simulation detail

To finalise an ASIC design a number of different ‘levels’ of simulation must be performed.

- Functional
- Timing
- Electrical
- Physical (maybe?)

Each looks at different aspects of the design.

- Require increasingly detailed models to perform
  - Simulations take longer to run
  - Consequences of ‘respin’ increasingly expensive

This section concentrates on functional verification.

- Does the code (or schematics) perform the correct logical operations?

The other aspects will be revisited later.

---

**Functional tests**

A high-level model (e.g. TLM) can demonstrate the feasibility of an architecture. It may be refined down to a cycle-accurate model – i.e. one in which each operation can be timed by counting clock cycles – but not (sensibly) much further.

There is then a need for translation into a behavioural model which is the function of a Hardware Description Language (HDL). A behavioural model may be just that – a model – or it may be a synthesizable Register Transfer Level (RTL) description which will eventually be converted into hardware.

The translation process from TLM to RTL may be error-prone; there is also the possibility of discovering previously untested cases as the model is prepared in more detail. There is therefore a need for simulation tests at the behavioural level. These tests perform functional verification, i.e. they check that the logic does what it is intended to.

Functional tests do not guarantee that a resultant circuit will fulfil its requirements: for example the design may be too slow to be clocked sufficiently fast to meet real-time requirements. Neither do they guarantee that the result will be constructable.

**Timing verification**

Functional tests typically use assumptions of synchronous behaviour and do not give any absolute timing information; elapsed time is measured in clock cycles. The designer may have a target clock speed in mind at this point; whether the design will achieve that is still unknown. Thus a design may do what it is intended to do but may not meet real-time requirements.

It is possible to annotate the HDL description with timing estimators to gain some idea of timing behaviour but these are necessarily imprecise. Circuit speed is greatly influenced by its physical properties which are not yet known.

Timing verification involves estimating and summing the various delays in an implementation to identify and isolate the critical path. This can only be achieved once a design has been synthesized into the target technology so that a netlist of gates, flip-flops etc. is available.

There are typically two phases to timing verification: pre- and post-layout. Having obtained a netlist, CAD tools can produce an estimate of the critical path delay. Modern tools are smart enough to try to factor in wiring loads as well as gate delays; however the true wiring delays are not yet known. This will give a moderately accurate guide to the maximum clock frequency. If the circuit is apparently too slow at this point it’s ‘back to the drawing board’.

Post-layout synthesis depends on a netlist following the Place-And-Route (PAR) stage and has a better estimate of timing. As well as wiring delays, other factors such as the need to add buffers (electrical amplification) may have been introduced. The simulation model used may be more precise, too. All this means is that it takes longer and, if a problem is found here, it is more expensive to iterate the design again.

Another concern here are the ‘edge speeds’ – the time it takes to switch wires between digital states. An edge which is too slow harms circuit performance but is also more vulnerable to electrical noise inducing extra switching. Not all gates will have the same input threshold so a slow edge may apparently switch at different times when interpreted at different destinations.

**Electrical characteristics**

A real circuit needs power, something neglected up to this point. When a gate switches its output there is a surge of charge onto or off a power supply. All these cause a (varying) current in the power supplies. There are (at least!) two serious concerns:

- Are the wires big enough to handle the current? A too-small wire is a fuse and will blow! More likely, there may be a wire that’s a bit thinner than is desirable which will ‘age’ due to electromigration and shorten the lifetime of the device.
- The gate models assume a supply of a certain voltage. The power supplies carry current and have resistance, therefore will impose a voltage drop (Ohm’s law). Thus the supply voltage at the centre of a chip (furthest from the connections) will be lower than that at the edges. The power supply wiring must be adequate to keep this drop within bounds for previous assumptions to remain valid. In case of difficulty, the usual solution is to force an increase in the number/width of the supply wires.

Of course the power that goes in comes out as heat. Thus there may be thermal modelling … and so on …
Objective
To verify that the RTL description fulfils all the logical operations required of the system.

You develop (a piece of) a system. It has to perform a certain set of functions and fit the appropriate set of interfaces. To do this a set of tests are required.

Some tests may already exist:
- Example: checks that a bus interface obeys the prescribed protocol

When a module is complete and tested it can be used in higher level functional tests.

You provide:
- a design, comprising HDL modules, schematics etc.
- a stimulus sequence to stress the design
- (possibly) a set of expected results
  - maybe generated from a higher level model

You can collect:
- waveform traces of a set of signals
- internal states at various times/conditions

What you don’t get (yet)
- Proof that your logic is synthesizeable
  - on the target technology
    - or, indeed, at all
- Estimates of size
  - feasibility
  - economic viability
- Estimates of speed
  - cycle accuracy is provided for synchronous units
  - no estimates of critical path ⇒ cycle time
- Estimates of power requirements

If any of the above criteria fail later then the RTL will need redesigning. Any editing may change (break) the functionality, so retesting is important.

Intention: reduce the number of redesign cycles to a minimum. Experience helps with this – but is not a sure guide.
**Functional Simulation**

Objective is to verify the logic behaviour of the design. Try to exercise every function.

Can be assisted by **test-coverage tools**:

- Which HDL statements have (not) been executed
- Which branches have (not) been taken
- Which nodes have (not) adopted both binary states at least once
  - at behavioural HDL there are no ‘nodes’ so this is tricky!

Achieving ‘complete’ coverage can be quite challenging!

It is not possible to gain accurate **timing** models … yet.

- **Cycle accuracy** is inherent – can count clock pulses
  - may previously have been estimated
- Some **annotation** is possible with estimated delays

---

**Functional Simulation**

**What it does**

Functional simulation – which you should be familiar with – is a test of the logic operation of a design. Higher level models should have shown that the algorithm can work; now the implementation must be verified.

Functional simulation is (primarily) a digital, event-driven simulation. This allows the simulators to run quite quickly which, in turn, allows a lot of test patterns to be exercised in a sensible elapsed time.

Simulation (of a synchronous design) at this level will be cycle accurate so the number of clock pulses taken can be counted. This gives a better ‘feel’ for the timing of a function. This may allow some redundant cycles to be identified and eliminated. If the design has a constrained clock speed then accurate performance estimates are possible, assuming the target clock speed can be achieved.

**What it doesn’t do**

What this form of simulation doesn’t give directly is any assurance that a particular clock frequency can be achieved by the synthesized logic. Delays can be included in the simulation but, in the first instance, these are simple estimates. Later, when the design has been developed further, more realistic delays can be extracted from the synthesized circuit and back-annotated to improve the realism of the simulation.

However, typically, a few test patterns are able to show timing problems. The main value of functional simulation is to provide good test coverage of the logical design.

**Hint**

To avoid the simulation ‘running away’, a statement like:

```
initial #10000 $stop;
```

can halt a simulation after a time limit.
Functional Simulation

Verilog is a Hardware Description Language.

- It can do things that are not (easily) made into actual hardware.
- This is useful for test purposes.
- The test environment can respond to state evolution of the Device Under Test ... without writing an explicit machine

Example: a handshake signal:

- The `req` signal may be timed by reacting to the state of `ack`

(Note: the protocol shown here is not quite the same as in the lab.)

Test strategy

The test block needs to provide 'req' which interacts with 'ack'.

One way: work out in advance (by hand?) the expected response times of the 'ack' signal and drive 'req' accordingly.

- Should expose the first error
- May be unreliable thereafter

Another way:

use the language to respond to the test block.

```verilog
... while (ack == 1) #CLOCK; // AAA
req = 1;
while (ack == 0) #CLOCK; // BBB
req = 0;
...
```

This:

- tests to see if any previous handshake is complete (AAA)
- if not, wait for a clock cycle and test again
- asserts the request
- inserts clock cycles until the acknowledge rises (BBB)
- removes the request
- continues – the test for handshake completion would occur before req is raised next time.

This form of test would typically be a single 'thread' which would run in parallel with other statements.

The details of such a handshake can be 'hidden' in a task.

Example: FIFO test

```verilog
initial // Input handshake
begin
rin = 0;
#clock;
rin = 1;
while (ain == 0) #CLOCK;
rin = 0;
while (ain == 1) #CLOCK;
#(10*clock); // Pause before continuing
...
end

initial // Output handshake
begin
aout = 0;
while (rout == 0) #CLOCK;
#(2*clock); // Choose to insert extra delay
aout = 1;
while (rout == 1) #CLOCK;
aout = 0;
...
end
```

This is an illustrative example. A real test would continue, for example stalling in different phases, trying to overfill the FIFO etc.

Note: it is convenient (but not compulsory) to use independent processes at the input and output.
Handy constructs

if (Boolean_expression) statement_1 {else statement_2}

- Used for making decisions: multiplexers, enables, simulation control
- ‘if’ is also used for compile-time control

while (Boolean_expression) statement

- Used for simulation control: e.g. handshaking

for (addr = 0; addr < 1024; addr = addr + 1) statement

- Iterate over a number of items: e.g. memory test

forever statement

- Loop indefinitely
  - must include some delay!

Control constructs

‘if’ has several uses:

- Synthesized into a multiplexer
- Making a run-time decision in simulation
- Control in generate of circuits [See later]
- Conditional compilation (as ‘ifdef, etc.)

it may take an else clause.

Note: In synthesizable, combinatorial logic the else may be wanted to avoid creating a latch, inadvertently.

Example

repeat (100)
begin
  #PERIOD
  if (req) ack = 1;
  else ack = 0;
end

Loop statements

‘while’ functions much as you would expect. It is not synthesizeable.

'define PERIOD 100
...
...
req = 1;
#PERIOD;
while (ack != 1) #PERIOD;
req = 0;
...
// Probably don’t care about
...
// completing handshake immediately
while (ack != 0) #PERIOD;
...

Request is asserted and given time to be seen.
If there is no acknowledgement, insert another clock cycle (and try again).
Continue waiting until the acknowledgement appears.

Hint
To stop a simulation due to an error, try:

initial failed = 0;
...
always @ (posedge failed) #100 $stop;
...
if (<error condition>) failed = 1;

- The delay allows time to be able to see the final signal states.
- The error is easy to find – at the end of the trace
Parallelism

Hardware, and therefore a HDL, is highly parallel too.

- Each initial and always block is an independent, parallel thread. Within which there can be:
  - Sequential block begin ... end
  - Parallel block fork ... join

In a parallel block, all statements are executed ‘simultaneously’, as if they were in separate blocks. Sometimes this makes no difference. It is important when inserting delays, however.

```
begin
#10 a = 1;
#20 b = 0;
end
```

Elapsed time 30 units

```
fork
#10 a = 1;
#20 b = 0;
join
```

Elapsed time 20 units

These blocks can be nested.

- Non-blocking assignments scheduled at a particular time all assign simultaneously.

Blocks

Blocks compose much as you might expect. In terms of timing a sequential block will take the sum of its internal delays whereas a parallel block will take the maximum of its individual delays.

```
initial
begin
a = 0; b = 0; c = 0; d = 0; e = 0;
end
fork
begin
#10 a = 1;
#10 b = 2;
end
fork
#10 c = 3;
#30 d = 4;
join
join
e = 5;
#10 $stop;
end
```

Named blocks

Individual blocks can be given unique names after the begin or fork.

```
begin: my_block
```

This can sometimes aid in identification in traces
Simulation time

As far as physics currently understands time is a single continuous dimension.

Simulation time is discrete but multidimensional!

- There is a simulation time which represents real delays
  - This is available as $\text{time}$
  - Resolution is controllable; may be a fraction of ‘#1’
- There is a list of things which happen simultaneously (in a given simulation) timestep
  - Some of these are ordered
    - e.g. blocking assignments within the same block
  - Some are unpredictable
    - e.g. blocking assignments different blocks
- There are different phases
  - first: all blocking assignments
  - second: non-blocking assignments

Naturally, none of this relates to the actual time taken to run the simulation, which depends on how much switching activity takes place in the design.

Assignment timing

Assuming they are scheduled to happen at the same time:

- Blocking assignments occur in some order.
  - the order is only defined within a single statement
- Non-blocking assignments happen simultaneously
  - first all the results are calculated from the RHS expressions
  - then all the LHS variables are altered
- Non blocking assignments happen after blocking assignments

Example

```verilog
always @(posedge clk) a = b; // Blocking
always @(posedge clk) b = a; // Blocking
always @(posedge clk) x <= y; // Non-blocking
always @(posedge clk) y <= x; // Non-blocking
```

After a clock edge:

- $x$ and $y$ will have been swapped
- $a$ and $b$ will be the same
  - but which value depends on the implementation
    i.e. which of the blocking statements is scheduled first

However:

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

is entirely predictable – if rather pointless

Potential pitfall

```verilog
always @(posedge clk) x <= a; // Non-blocking
always @(posedge clk) a = b; // Blocking
```

In simulation $x$ will always take the value from $b$.

However a synthesizer will probably see this as two sequential flip-flops.

A synthesized circuit will behave differently from the simulation!

Okay, that’s the wrong thing to happen but complaining won’t change it.

Guidelines

- Use blocking assignments for combinatorial logic
  - this should evaluate every time its inputs change
- Use non-blocking assignments for D-type (edge triggered) registers
  - typically ‘posedge clk’
- Don’t mix blocking (=) and non-blocking (<=) assignments …
  - … to a particular variable
  - … within the same always block
- Keep all assignments to a particular variable in the same block

- Try to avoid transparent latches
  ```verilog
  always @ (D, En) if (En) Q = D;
  ```
  - only do this if you must
  - easily created accidentally: remember ‘else’s and ‘default’s
  - may be functionally harmless but introduce redundant logic
  - some tools may produce warnings if you do
Stylistic pitfall (?)

When simulating synchronous circuit models the most convenient thing to write is:

```verilog
class always @(posedge clk)
    if (count < 9) count <= count + 1; else count <= 0;
```

‘count’ then changes:

- as a result of the clock edge
- after the clock edge
- at the same time as the clock edge

The resulting trace may be slightly misleading although it is safe

But what if the inputs (RHS) are generated with a blocking assignment ...

- as combinatorial logic?  Okay – inputs settled in time
- `@ (posedge clk)`?  Bad news – inputs change before non-blocking statement
  (In simulation, maybe not in synthesized logic.)

Therefore keep all blocks and modules in the same style!

This might aid readability …

(Perhaps) the obvious way to stimulate a design may be something like this:

```verilog
initial clk = 1;
always #5 clk <= !clk;

begin
    data = 0;
    #10;
    data = 1;
    #10;
    data = 2;
    #10;
end
always @(posedge clk) value <= data;
always @(posedge clk) value2 <= value;
```

Unfortunately this can lead to the following timing relationship:

```
data   | 0 | 1 | 2
value  | X | 1 | X
value2 | X | X | 1
```

- The ‘pipeline’ behaves as one might expect
- The input traces are easier to spot

Offsetting input changes slightly from the clock may help

```verilog
parameter period = 10; // Makes changes easier
initial clk = 1;
always #(period/2) clk <= !clk;

begin
    #1;
    data = 0;
    #period;
    data = 1;
    #period;
    data = 2;
    #period;
end
always @(posedge clk) value <= data;
always @(posedge clk) value2 <= value;
```

```
data   | 0 | 1 | 2
value  | X | 0 | 1
value2 | X | X | 0
```

Suggest that this is clearer!
Delays

- Delays use a syntax ‘#<value>’
- Delays can be added to a model in various ways.
  - They will not be synthesized and cannot be relied upon for functionality.

Here are some convenient methods:

```verilog
#20 a = 1l; // Delay in execution of sequential block
wire #4 q; // Declare a ‘net delay’
...
assign q = a & b; // q changes 4 timesteps after an input change
register <= #10 input_value; // Propagation delay on signal
```

Uses include

- Sequencing I/O in a test run
- Modelling ‘real’ components
  - e.g. external memory read delay in lab. system (> clock period)
- ‘Cosmetic’ delays to make waveform traces more readable

Time is modelled in two ways:

- advancing to the next timestep
- iterating over things which happen at the same time

Examples:

```verilog
initial
begin
a = 0;
#10;
a = 1;
end

assign b = !a;
assign c = !b;
```

At any point there is only one thing which is scheduled to happen, but a change schedules an immediate change on b, etc.

<table>
<thead>
<tr>
<th>Time</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a = 0</td>
</tr>
<tr>
<td>0</td>
<td>b = 1</td>
</tr>
<tr>
<td>0</td>
<td>c = 0</td>
</tr>
<tr>
<td>10</td>
<td>a = 1</td>
</tr>
<tr>
<td>10</td>
<td>b = 0</td>
</tr>
<tr>
<td>10</td>
<td>c = 1</td>
</tr>
</tbody>
</table>

Delays can alter when switching occurs.

Delays can also be inserted within assignments to delay the assignment without retarding the flow of execution, e.g.:

```
a <= #10 b;
```

Different delays

For added veracity it is possible to specify different rising, falling and turn-off delays for a signal by listing these in brackets. E.g.

```
#(3,2)
```

means a rising delay of 3, a falling delay of 2. The turn-off delay has not been specified so defaults to the minimum of these (if appropriate).
Events

Digital simulation is usually ‘event-driven’.
Events are awaited using the ‘@’ keyword.

- e.g. as part of a loop always @ (...) 
- May be used anywhere in code

An ‘event-driven’ simulator keeps track of when (instantaneous) stage changes will occur.

- A signal changing is an event @ (enable)
- These can be further refined @ (posedge clk)
- Other events can be created to aid simulation event my_event;
  -> my_event; // signal something
  ...
  always @ (my_event)
  ...

Waits for the next time the event is detected in the future
Not a strict thread rendezvous

Events

Events are things that happen during execution; they can be used to control the progression of a test sequence. An event can be something like:

(posedge clk)

It is possible to wait for events within a test harness. The ‘@’ keyword really means ‘wait until the following condition is satisfied’. This could be as simple as synchronising with the next clock cycle:

initial
begin
  ...
  @ (posedge clk) // Sync. with clock transition
  #2; // Delay changes for clarity
  a = a + 1; // Change input variables
  b = 17;
  ...
end

Which will wait until the specified event occurs. These can be used in combination with other operators. For example:

repeat ($random & 'h3) @ (posedge clk);

Will wait for between zero and three (inclusive) rising edges of the clock.

User events

Events can also be declared and generated in a behavioural model. For example, here is a mechanism for generating reports when errors are detected:

event error; // Declare event
always @ (error) $display("Error at time %t", $time);

initial
begin
  ...
  // Simulation proceeds
  if (<error condition>) -> error; // Generate event
  ...
end

Such signalling can be used to allow different blocks to interact in a test file. For example:

initial
begin
  [Do reset sequencing]
  -> reset_done;
end

initial
begin
  @(reset_done)
  [Start test sequence]
  ...
end

For more complete examples, try:
Comparing results

Three options:

- **Stare at waveforms by hand**
  - useful in initial debugging
  - error prone and tedious for regression tests

- **Use Verilog to compare results against an expected set**
  - with a predictable progression can regenerate results in test harness
  - can import expected results from a preprepared file

- **Dump a trace from the simulator and compare off-line**
  - simple tests can alert user to anomalous conditions
  - data traces can be exported into files for analysis

---

**Importing test results**

Probably the easiest way to do this is to read a file of expected results into a memory residing in the test harness. Initialise a pointer to the start of the file and increment it every time a comparison is made.

```verilog
reg [7:0] results [0:1023];
reg [10:0] result_pointer;
reg [7:0] comp_value;
initial $readmemh(<filename>, results);
initial
begin
    result_pointer = 0;
    ...
    test_next(comp_value); // Fetch value
    ...
end

task test_next; // Use a task for convenience
output [7:0] value;
begin
    if (result_pointer > 1023)
        begin
            $display("Out of data!");
            $stop; // Oops! Error
        end
    else
        begin
            value = results[result_pointer];
            result_pointer = result_pointer + 1;
        end
end
endtask
```

**Exporting monitoring information**

There are several ways to do this.

**User prints**

```verilog
$display
fwrite
```

C-like print statements: identical except $display adds a LF to the string end.

Suggestions:

- Use to monitor progress through a stimulus file
- Use to pick up and diagnose errors

```verilog
e.g.
$display("Starting test");
...
if (q != 32'h0000_0000)
    $display("Unexpected value: q = %X", q);
```

Sometimes it is convenient to watch particular signals rather than all signals through time. This can be done with the $monitor task.

```verilog
initial
$monitor("this = %x that = %x at time %t", this, that, $time);
```

This is similar to:

```verilog
always @(this, that)
    $display("this = %x that = %x at time %t", this, that, $time);
```

**Writing to files**

```verilog
$fopen(), $fclose(), $fwrite()
```

allow you to create your own output files. ("fwrite" is the same as "fwrite" except for line-feeds.) Their use, including file handles, is similar that which should be familiar from C programming.
Initialisation

When a state-holding element is switched on it will settle into a stable state. It is not predictable what state this will be, so it is unknown.

Does this matter? In some cases it does, in others it doesn’t.

Example: ARM registers

- R0-R14 are undefined
- R15 (PC) is 00000000

I.e. only the essential values are cleared

Rule of thumb:
- Control registers should be initialised
- Data registers probably don’t need initialisation

Undefined (“unknown”) values tend to propagate through logic. This is usually a good thing as it acts as a warning that something is wrong. Learn to exploit them!

Reset signals

The question of whether to include a reset on a flip-flop has no simple answer. The inclusion of reset logic imposes a small power/area penalty on a flip-flop. The fan-out of the reset network is typically very large and consumes some resources. Failing to reset a flip-flop which needs to be defined can be catastrophic.

Case: consider a clock divider flip-flop:

As a circuit this will function because it is in some digital state and will toggle to the other when clocked.

In simulation it will start unknown and always remain that way.

In Verilog you could ‘cure’ this anomaly with an ‘initial’; but if that is accidentally done to something where the phase matters then it’s Doom again.

In general:
- data registers can tolerate starting undefined
- control registers should be reset
  - this would include ‘validity’ indicators on data etc.

Some designers prefer to reset every flip-flop as a matter of routine. (There is a small additional cost in size and speed.)

You have to decide which approach works for you.

Memories

RAM does not have reset. If you expect a RAM to contain some values (typically zeroes) you will have to write these in actively.

- This is easy in a RAM model with a ‘for’ loop.
- This is considerable effort in reality requiring an FSM … and time.

Resetting in FPGAs

The state of an FPGA is downloaded when it is configured. This means:

- flip-flops will be initialised
- memory contents will be defined

This means, for example:

- ‘initial’ will work and a register can be preset at \( t = 0 \)
- a memory (on chip) can be initialised with ‘$readmemh()’

The second of these allows the creation of on-chip ROMs, useful (for example) in bootstrapping a processor or providing look-up tables.

Note that these facilities are provided because of the particular FPGA characteristics. They would not be available on an ASIC.

The ‘unknown’ state in simulators

Unknown/undefined states are a characteristic of digital simulation. Verilog simulation is digital, with signals adopting states \( \{'0', '1', 'x', 'z'\} \) and these have well-defined operations. For example:

\[
0 \&\& x == 0 \quad 1 \&\& x == x
\]

Digital (functional) simulation does not represent intermediate states, transition (edge) speeds etc.

Not all simulators work this way. Circuit-level simulators represent the analogue voltages on the wires to give more accurate estimates of the behaviour of the implementation. These values are (in principle) continuous, therefore always ‘known’, therefore they have to be assumed at start-up.

Unknown values will not appear. However the assumed values may not be those which occur in a real circuit and therefore should not be relied on.

This is another good reason for running a functional simulation as part of the design flow.
iffy logic?

- In digital logic there are two possible logic states: \{0, 1\}

if (a == 1) <statement> // Outcome - obvious

- In digital simulation there are three possible logic states: \{0, 1, \{x, z\}\}

if (a == 1) <statement> // Outcome - less obvious

An if clause is taken if the predicate is ‘true’ (i.e. 1).
An else clause is taken if the predicate is not ‘true’ (i.e. 0 or \(x\) or \(z\)).

Verilog defines its operators as:

- “logical (in)equality”
  - ‘==’ and ‘!=’ may return \{0, 1, \(x\)\}

- “case (in)equality”
  - ‘===' and ‘!==’ only return \{0, 1\}, looking for an exact match
  - useful for verification (e.g. detecting unknowns) not for synthesis

Equality operators

When executing an ‘if’ in a Verilog simulation a binary decision is made using three possible input values.

The case equality operators can be used to eliminate one of these.

These are useful, for example, in ‘if’ tests.

\[
\begin{align*}
0 == 0 & \quad \text{// Result is true (1)} \\
0 == x & \quad \text{// Result is unknown (\(x\))} \\
0 !== x & \quad \text{// Result is false (0)} \\
0 != x & \quad \text{// Result is unknown (\(x\))} \\
x == x & \quad \text{// Result is true (1)}
\end{align*}
\]

Example

Imagine you are testing for a particular pattern.

```verilog
if (data != test_value) $display("Error");
```

Bad idea! If the data and test_value are both defined then it works but if either is ‘unknown’ (\(x\)) then the expression will return ‘unknown’. This isn’t ‘true’ so the $display statement is not reached.

This could be cured by:

```verilog
if (data == test_value) ; // do nothing
else $display("Error");
```

but this is better:

```verilog
if (data != test_value) $display("Error");
```

Example

Here is a ‘broken’ AND gate:

```verilog
module my_and (input wire a, b, output reg q);

always @(a, b)
if (a)
  if (b) q = 1;
else q = 0;
else q = 0;
endmodule
```

Synthesized, this should works fine. However in simulation it will always produce a logic value output, even if the inputs are undefined.

<table>
<thead>
<tr>
<th>my and</th>
<th>(X)</th>
<th>(Z)</th>
<th>AND</th>
<th>(X)</th>
<th>(Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(X)</td>
<td>0</td>
<td>0</td>
<td>(X)</td>
<td>0</td>
<td>(X)</td>
</tr>
<tr>
<td>(Z)</td>
<td>0</td>
<td>0</td>
<td>(Z)</td>
<td>0</td>
<td>(X)</td>
</tr>
</tbody>
</table>

This may be harmless in itself but has the potential to hide other faults.
### Into the ‘unknown’ ...

```verilog
case (abc)
  2'b00: result = 1;
  2'b01: result = 2;
  2'b10: result = 3;
  default: result = 0;
endcase
```

- Cases are compared top-to-bottom.
- Only **exact** matches are considered.
- What happens if the input variable is 2'b0x ?

### ‘unknown’ is not the same as ‘don’t care’

```verilog
casex (xyz)
  2'b00: result = 1;
  2'b01: result = 2;
  2'b1x: result = 3; // Taken for cases 2 and 3
  default: result = 0; // Redundant - but good (cautious) practice
endcase
```

---

### Unknowns and ‘don’t cares’

**A ‘case’ statement selects one entry from a list of options; the first one which matches the criteria.**

The following example illustrates how all cases of a single wire could be monitored.

```verilog
case (some_wire)
  1'b0, // Note list of cases
  1'b1: $display("Wire has value %d", some_wire);
  1'bX: $display("Wire is unknown");
endcase
```

This facility is useful in testing and, perhaps, modelling: it is not relevant to synthesis because a physical wire will always have a value which is interpreted in some way as a digital signal.

#### Don’t cares

Two additional keywords allow variations in a case construct:

- **casez**
- **casex**

These allow ‘don’t care’ inputs in a case statement. The difference is that ‘casez’ treats only ‘Z’ inputs as don’t cares whereas ‘casex’ accommodates both ‘Z’ and ‘X’ equally.

‘casex’ is possibly the more useful and serves as an example.

---

### casez

```verilog
casez (some_wire)
  1'b0,
  1'b1: $display("Wire has value k%d", some_wire);
  1'bX: $display("Wire is unknown");
endcase
```

### casex

```verilog
casex (instruction[27:24])
  4'b000x: $display("Data processing");
  4'b01xx: $display("Load/store");
  4'b100x: $display("Load/store multiple");
  4'b101x: $display("Branch");
  4'b110x,
  4'b1110: $display("Coprocessor");
  4'b1111: $display("SVC (aka SWI)");
endcase
```

Sometimes it is useful to be able to build a circuit where part of an input is not compared, i.e. it is a **don’t care** value. Not caring what a value is is not the same as not **knowing** what a value is; however (confusingly?) Verilog uses ‘X’ for both cases.

Imagine decoding a microprocessor instruction. Not all the bits are used to determine the instruction type. If different sets of bits need to be examined, ‘casez’ allows these to be specified in a single, simple statement.

In the following (simplified) ARM instruction decoder the ‘X’s are bits which are not considered by the case statement.

```verilog
if (instruction[27] == 0)
  if (instruction[26] == 0) $display("Data processing");
  else $display("Load/store");
else
  if (instruction[26] == 0) $display("Load/store multiple");
  else $display("Branch");
else
  if ((instruction[25] == 0) || (instruction[24] == 0)) $display("Coprocessor");
  else $display("SVC (aka SWI)");
```

Bases other than binary can be used. In hexadecimal an ‘X’ represents four don’t care bits, in octal, three. It is not meaningful in decimal.

Without cases the decoder would probably look something like:

- **If** (casez)
- **Else**

Without cases the decoder would probably look something like:

- **If** (casez)
- **Else**