Simulation

COMP32211

Implementing System on Chip

Simulation

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

- Assume that high-level modelling has been done
  - Already know the architecture and the algorithms: models exist
  - Know the 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 (manufactured) 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.

"A little inaccuracy sometimes saves tons of explanation."
Saki; *The Square Egg* (1924)

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

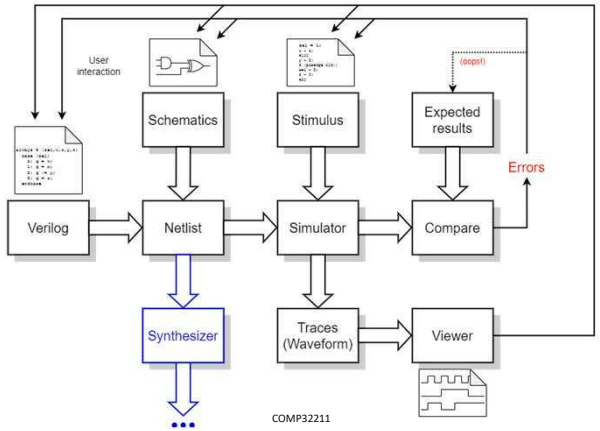
- Require increasingly detailed models to perform
  - Simulations take longer to run
  - Consequences of 'respin' increasingly expensive

This section concentrates on **functional verification**.

- Does the 'code' perform the desired **logical** operations?

Other aspects will be revisited later

Functional simulation flow



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' (wires) so this is tricky!

Achieving complete coverage an be quite challenging (even *justifiably* impossible)

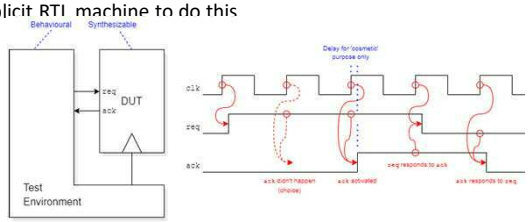
It is not possible to gain complete **timing** models … yet

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

Functional simulation

Verilog is a Hardware Description (HDL)

- 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 (DUT)
  - … without building an explicit RTL machine to do this
- Example: a handshake signal



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

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

Handy constructs

```
if (Boolean_expression) statement_1 <else statement_2>
```

- Used for making decisions: multiplexers, enabling, 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** ('`@`' or '`#`' in testbench)

Parallelism

Hardware – and therefore a HDL – is highly parallel.

- Each '`initial`' and '`always`' block is an independent, parallel 'thread' within which there can be a:
  - 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 managing delays, however.

`begin`	`fork`
`#10 a = 1;`	`#10 a = 1;`
`#20 b = 0;`	`#20 b = 0;`
`end`	`join`
Elapsed time 30 units	Elapsed time 20 units

These structures can (of course) be nested

- Non-blocking assignments scheduled at the same time are simultaneous.
- Coincident blocking assignments (in simulation) may be arbitrarily ordered.

Simulation time

As far as physics currently understands, time is a single, continuous dimension.
Verilog simulation time is discrete but multidimensional!

- There is a simulation time which represents real delays
  - This is available as `$time`
  - Resolution is controllable: may be a fraction of '`#1`'
- There is a list of things which happen simultaneously (in a given simulation) time-step
  - Some of these are ordered
    - E.g. blocking assignments within the same block
  - Some are unpredictable
    - E.g. blocking assignments in different blocks
  - There are different phases
    - First: all blocking assignments
    - Second: non-blocking assignments

Naturally, none of this refers to the actual time taken to run the simulation, which depends on the implementation and how much switching activity occurs within it.

Stylistic pitfall (?)

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

```
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 correct and safe
but what if the inputs (RHS) are generated with a blocking assignment …

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

Therefore keep all blocks and modules in the same style!

Delays

- Delays use a '`#<value>`' syntax.
- 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 methods of expressing delays:

```
#20 a = 11;          // Delay in execution of a sequential block

wire #4 q;           // Declare a 'net delay'
...
assign q = a & b;  // q changes 4 time-steps after an input change

register <= #10 input_value;   // Propagation delay on signal change
```

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

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) state changes will occur.

- A signal *changing* is an event ..... `@ (enable)`
- This can be further refined ..... `@ (posedge clk)`
- Other events can be *created* to aid simulation

```
event my_event;    // Declaration
-> my_event;  // Signal something
...
always @ (my_event)
...
```

Wait for the next time the event(s) occur(s) in the future.

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 ('golden') set
  - Can be regenerated in test harness – provided code appropriately written (esp. timing)
  - Can import expected results from a preprepared file (e.g. from external modelling)
- Dump a trace from the simulator and compare off-line
  - Simple tests can alert a user to anomalous conditions
  - Data traces can be exported into files for comparison/analysis

Initialisation

When a state-holding element is switched on it will settle into a stable (binary) state.
It is not predictable what state this will be (on an ASIC) 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; CPSR holds supervisor mode & interrupts disabled

i.e. only essential values are (guaranteed) cleared
Rule of thumb:

- Control registers should be initialised
- Data registers *probably* don't *need* initialisation.

Undefined (unknown) values tend to propagate and spread in logic-level simulations.
This is usually a Good Thing since it acts as a warning if something is wrong. Learn to exploit them!

iffy logic?

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

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

- In digital simulation there are (arguably) *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')
A (matching) '`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 '`=== 'hx`') – but not for synthesis!

Into the 'unknown' …

```
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 (abc) is 2'b0x ?

'unknown' is not the same as 'don't care'

```
casex (xyz)
  2'b00:   result = 1;
  2'b01:   result = 2;
  2'b1x:   result = 3;    // Taken for cases 2'b10 and 2'b11
  default: result = 0;
endcase
```