Chapter 2
Ensuring RTL Intent

A user starts the design of his block, by describing the functionality of the block in the form of RTL. The RTL code is then synthesized – to realize the gate level connectivity that would provide the same functionality. Before synthesizing, the designer needs to ensure that the RTL actually implements the functionality that is desired. For this purpose, the designer runs a lot of simulations. The process of simulation shows that for a given set of input vectors, what would be the response. Only when the designer is sure of the RTL achieving the desired functionality, the RTL is sent for synthesis and subsequent steps. The main advantages of describing the design in RTL rather than logic gates are mainly two fold:

(i) Higher level of abstraction makes it easier to describe the design, compared to providing the gate level connectivity.
(ii) RTL simulations are much faster compared to the corresponding gate level representation. Hence, validation of the functionality can be much faster.

2.1 Need for Unambiguous Simulation

Since the RTL code has to ensure that it has achieved the desired functionality, the RTL code is subjected to rigorous simulations with many different vector-sets to cover all the functionalities of the device. These vectors are supposed to cover various situations – including:

- Normal mode of operation
- Specific corner-case situations
- Error/recovery handling etc.

Simulation is unambiguous till the time the RTL has only one possible interpretation. However some pieces of RTL code can have multiple interpretations. The simulator being used by the RTL designer will exhibit any one of the possible multiple interpretations. The designer is finally satisfied with the exhibited functionality. However, it is possible that during simulations, the interpretation that has been used

S. Churiwala, S. Garg, *Principles of VLSI RTL Design*,
DOI 10.1007/978-1-4419-9296-3_2, © Springer Science+Business Media, LLC 2011

by the simulator is different from what gets finally realized (synthesized). If this happens, even though, the design functionality is verified through simulation, the actual operation of the realized device does not exhibit the same behavior.

Functional verification of an RTL code (through simulation) is one of the most basic and very fundamental aspect of a chip design and therefore, it is imperative that any RTL code that is written is un-ambiguous in terms of functionality.

2.2 Simulation Race

The word *race* could have different meanings in different context. In the context of HDL simulation, the word *race* means same HDL code exhibiting two or more possible behaviors – both of which are correct as per the language interpretation. So, a *race* implies ambiguity. By its definition itself, *race* says that there are multiple interpretations, and, all of them are correct. A simulator is free to choose any one of these multiple interpretations. However, it is not necessary that the final realization will match the interpretation that your simulator chose.

Simulation race is usually encountered in *Verilog* code. *VHDL* does not have any simulation race but it has some other nuances – which we will see in Section 2.3.3. Also, *SystemVerilog* has provided additional constructs and semantics which provide additional information to the simulators so that their response is consistent. Thus, *SystemVerilog* has provided ways to do away with a lot of *race* situations. However, it is still possible to have a *SystemVerilog* code, where a portion still has a race. This can happen, if the designer has used *SystemVerilog* only at some portions of his code and not really exploited the full prowess of capabilities provided by *SystemVerilog* to avoid a race situation.

In case of a design having a race:

- Different simulators can give different results
- Different versions of the same simulator can give different results
- Same version of the same simulator can give different results – based on switches chosen or changes made to the code, which have seemingly no relation with the change in behavior being exhibited. For example, including some debug level switches or some debug type statements could change the result of the simulation

2.2.1 Read-Write Race

Read-Write race occurs when a signal is being read at the same time as being written-into.

2.2.1.1 Combinational Read-Write Race

Consider the following code-excerpt:

```
assign a = b & c;

always @ (b or d)
if (a)
        o = b ^ d;
```

In the above code-segment, *a* is written into through the ***assign*** statement – while being read through the ***always*** block. When *b* gets updated, it will trigger both the ***assign*** statements as well as the ***always*** block. The ***if*** condition in the ***always*** block will see the updated value of *a,* if the ***assign*** statement gets executed first. On the other hand, the ***if*** condition will see the old value of *a*, if the ***always*** block gets executed first. Depending upon which of the two behaviors is chosen by your simulator, you might get a different behavior. The above code segment is an example of a race.

Similar race-condition is also exhibited by the following code segment:

```
always @ (b or c)
a = b & c;

always @ (b or d)
if (b) o = a ^ d;
```

In the above code segment, *a* is being written into (through the first ***always*** block) – while it's also being read (through the second ***always*** block). Depending upon which ***always*** block gets triggered first, the second ***always*** block may see the updated value or the old value of *a*.

The solution to remove this ambiguity is very simple. In both the examples shown, the sensitivity list of the second ***always*** block should have *a* in it. This will ensure, as soon as *a* gets updated – the second ***always*** block gets triggered and values are re-evaluated with the updated value of *a*. In the worst case, the second ***always*** block might get triggered twice, but, the final values would be unambiguous. (Theoretically, this could still be a race, but, for all practical purposes – there is no ambiguity. See Appendix A for more details)

SystemVerilog provides a much more elegant solution to this kind of ambiguity. Instead of using the keyword, ***always***, you should use ***always_comb***. When you use ***always_comb***, there is no need to explicitly specify the sensitivity list. All the signals that are being read in this ***always_comb*** block will automatically be included in the sensitivity list.

2.2.1.2 Sequential Read-Write Race

Consider the following code-excerpt:

```
always @ (posedge clk)
b = c;

always @ (posedge clk)
a = b;
```

In the above code-segment, at the positive edge of *clk*, both the ***always*** blocks get triggered. So, *b* is being written into (through the first ***always*** block), while it is being read also (through the second ***always*** block). Assume that the first ***always*** block gets triggered before the second one. *b* gets updated. Then, the second ***always*** block gets triggered; *a* sees the updated value of *b*. Thus, in effect, the value of *c* percolates all the way to *a* through *b*. Consider another scenario. The second ***always*** block gets triggered before the first one. In this case, *a* sees the old value of *b*, and, then, *b* gets updated. So, depending on the sequence chosen by the simulator, the value on *a* could be either *c* (i.e. the new value of *b*), or, the old value of *b*.

The solution to deal with this kind of ambiguity is also very simple. For a sequential element, we should use a *Non-Blocking Assignment* (*NBA*). With an *NBA*, the Right-Hand-Side is read immediately, but, the updating of the Left-Hand-Side happens after all the reads scheduled for the current time have already taken place. Think of this as an infinitesimally small delay!!! We are considering this to be infinitesimally small, because in reality the simulation time does not move. It is just that all updates to Left-Hand-Side (of an *NBA*) happen after all the corresponding Right-Hand-Sides have been read. So, irrespective of what sequence is used for triggering the blocks, the events will occur in the following sequence:

(i) *b* and *a* will decide what values they should go to, but, they will not actually get updated. The relative sequence of the two ***always*** blocks is still undeterminable, but, that does not make any difference.
(ii) Subsequently, *b* and *a* will get updated. This updating happens after all read of Right-Hand-Side have already taken place. Since, *a* has already decided what value to go to (in the first step itself), so, any change in value of *b* is not going to impact the value of *a*.

Thus, the results can be made unambiguous through the use of NBA.

For the sake of correctness and completeness, it should be mentioned here, that with an NBA, after all the read has happened, and, then the left hand side gets updated; an updated LHS at this stage can trigger another sequence of reads. So, the updates are in effect taking place before some of the reads. However, from race perspective, this fact does not make any difference. The sequence of read and then update (which can then trigger more reads) remains unchanged.

2.2.2 Write-Write Race

Write-Write race occurs when multiple values are being written into a signal – at the same time.

Consider the following code-segment:

```
always @ (b or c)
if (b != c)        err_flag = 1;
else               err_flag = 0;
```

```
always @ (b or d)
if (b == d)        err_flag = 1;
else               err_flag = 0;
```

When *b* changes, both the ***always*** blocks get triggered. It is possible that in one of the ***always*** blocks, *err_flag* is supposed to take the value of *1*, while, in the other, it's supposed to take the value of *0*. The value that *err_flag* finally takes will depend on the sequence in which these ***always*** blocks get triggered. The one that triggers later – will determine the final value. Though *b* is the common signal in the two sensitivity lists, it has no role to play in creating the race. It is possible to have a similar race, with no signal being common in the two sensitivity lists. The race is being created because of the assignments in the two ***always*** blocks being made to the same signal.

always_comb (instead of the ***always***) as explained in Section 2.2.1.1 cannot be used here, because, same variable, *err_flag* cannot be assigned a value in two different ***always_comb*** blocks. Use of *NBA* as explained in Section 2.2.1.2 will also not solve this problem. The simplest solution to this ambiguity is to avoid updating a signal in more than one concurrent block. A signal should be updated in only one concurrent statement. In the following excerpt ***always_comb*** has been used just for convenience. For a combinatorial block, it is anyways a good practice to use ***always_comb***, rather than just ***always***.

Modifying the above code-segment – in line with the above guidelines, you get:

```
alway_comb
begin
if (b != c)        err_flag = 1;
else               err_flag = 0;
if (b == d)        err_flag = 1;
else               err_flag = 0;
end
```

The above code-excerpt is still wrong. This is not probably what one had intended. But, it does not have a race. It will behave in the same (incorrect manner) with all simulators!!! The following code-segment avoids the race as well as achieves the intended behavior:

```
alway_comb
begin
err_flag = 0;
if (b != c)        err_flag = 1;
if (b ==d)         err_flag = 1;

end
```

A Write-Write race can occur through combinations of:

- ***assign/assign*** statements – if two ***assign*** statements try to update the same variable. However, typically, these are caught easily in simulations – as the assigned variable might tend to go to an *X*.
- ***always/always*** statements – if two ***always*** blocks try to update the same variable. These situations might be encountered only in a testbench. When used in a design this will result in a Multiple Driver scenario after synthesis – which is a violation of basic electrical requirement. The simulator might not necessarily see the Multiple Driver scenario!!!

2.2.3 Always-Initial Race

Always-Initial race occurs when an ***initial*** block is updating a signal which also appears in the sensitivity list of an ***always*** block.

Consider the following code-segment:

```
initial
rst_n = 1'b0;

always @(posedge clk or negedge rst_n)
if (! rst_n)     q <= 1'b0;
else             q <= d;
```

Scenario 1

At the start of the simulation, the ***always*** block is triggered first. It now has to wait on a positive edge of *clk* or negative edge of *rst_n*. So, it now waits for the triggering event to happen. Then, the ***initial*** block is triggered. This causes *rst_n* to go to *1'b0* thereby causing a negative edge on *rst_n*. Since the ***always*** block was waiting for a negative edge of *rst_n*, this ***always*** block starts executing, taking *q* to *1'b0*.

Scenario 2

At the start of the simulation, the ***initial*** block is triggered first. It takes the *rst_n* signal to value *1'b0*. The negative edge on *rst_n* (in the ***initial*** block) has already happened, even before the ***always*** block could wake up. So, the negative edge on *rst_n* has been missed by the ***always*** block. So, *q* does not get initialized at the start of the simulation. The ***always*** block would wait for the next positive edge of *clk* or negative edge of *rst_n*.

Both the scenarios are valid as per *Verilog* language definition. Thus, a simulator might exhibit any of the above behaviors. These kinds of races are resolved through one of the following methods:

Before the advent of *SystemVerilog*, one method was to initialize using inactive values, and then, initialize after a while. So, the above code would be modified as:

```
initial
begin
rst_n = 1'b1;
#5 rst_n = 1'b0;
end

always @(posedge clk or negedge rst_n)
if (! rst_n)     q <= 1'b0;
else             q <= d;
```

The use of *#5* in the above example is indicative only. Important point is to put some delay – however small it may be. In the above example, at time *0*, the ***always*** block is not supposed to be executed. It would have just woken up (or, get armed – as some people say). So, irrespective of the order of execution, at time *0*, the ***always*** block gets armed, but, not executed. And, at time *5*, there is a negative edge on *rst_n*, which will cause the ***always*** block to be executed. The ***initial*** block in the example mentioned above is typically found in a testbench, rather than in a design. Synthesis ignores ***initial*** block as well as explicit delay specifications. Hence, such constructs are not used in design which will be later realized in terms of logic gates.

SystemVerilog provides a much more elegant method of removing the non-determinism (i.e. race). It says that all variables initialized at the time of declaration are initialized just before time *0*. *rst_n* can be initialized to *0* at the time of declaration itself (in the testbench). This means, at time *0*, there is no initialization, and hence, there will be no event. Thus, *a* deterministic behavior is forced by *SystemVerilog*.

2.2.4 Race Due to Inter-leaving of Assign with Procedural Block

Section 5.5 of the *IEEE Standard 1364–1995* (popularly called as *Verilog*) provides the following example code-excerpt:

```
assign p = q;

initial begin
q = 1;
#1 q = 0;
$display(p);
end
```

For this specific situation, different simulators are known to exhibit different behavior. When *q* is updated to *0*, the ***assign*** statement is supposed to reevaluate

the value of *p*. Some simulators continue on with the current ***initial*** block to display *p*, before moving onto the ***assign*** statement. These simulators display a *1*. On the other hand some simulators suspend the current ***initial*** block, and, execute the ***assign*** statement. Then, they come back to the suspended ***initial*** block, and, display the new value of *p*, thus displaying a *0*.

The way to resolve this ambiguity would be to put a delay, before the ***$display***. This delay specification will force the execution to move to the ***assign*** statement, before executing the ***$display***. Alternately, use of ***$monitor*** instead of ***$display*** should show the final value of *p*. Appendix A discusses race implications due to interleaving of concurrent processes.

2.2.5 Avoiding Simulation Race

In his paper, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!" Cliff Cummings of Sunburst Design recommends the following guidelines (reproduced verbatim) to avoid simulation race conditions in your Verilog RTL design:

(1) When modeling sequential logic, use nonblocking assignments.
(2) When modeling latches, use nonblocking assignments.
(3) When modeling combinational logic with an always block, use blocking assignments.
(4) When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
(5) Do not mix blocking and nonblocking assignments in the same always block.
(6) Do not make assignments to the same variable from more than one always block.
(7) Use $strobe to display values that been assigned using nonblocking assignments.
(8) Do not make assignments using #0 delays.

2.3 Feedthroughs

Feedthrough refers to a situation, where a signal, instead of just getting captured in the destination register, overshoots it and goes to the next stage also – within the same cycle, instead of waiting for one additional cycle. (The term Feedthrough has one more usage. We will look at another usage of the term in Section 8.4) Figure 2.1 shows an example, where – in each clock cycle, data from register *A* goes to *B*, and, that from *B* goes to *C*. Normally, a data which starts from Register *A* will reach *B* in one cycle, and, then, into *C* in yet one more cycle.

But, if the data from *A* reaches *B*, crosses it, and, goes to *C* – all within just one cycle, it is a situation of a *Feedthrough*. *Feedthroughs* can happen in VHDL also,

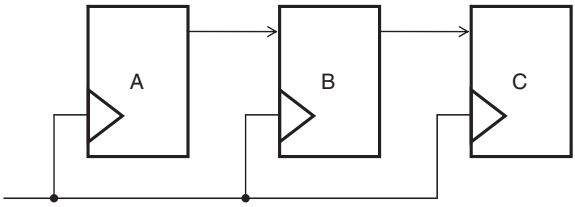

Fig. 2.1 Data transfer through registers

unlike *races* which happen only in Verilog. Some users refer to *Feedthrough* also as *race*. So, it should be understood that when spoken in the context of VHDL, a *race* typically means *Feedthrough*.

2.3.1 Feedthroughs Because of Races

One of the causes of *feedthrough* is races. Among multiple interpretations of a code, there might be one – which results in a *feedthrough*. Consider a Verilog code segment:

```
always @ (posedge clk)
q1 <= t1;

always @ (posedge div_clk)
q2 <= q1;

always @ (posedge clk)
div_clk <= ~div_clk;
```

Consider one possible scenario of events – for the above code. There is a positive edge on *clk*. *q1* will get updated. *div_clk* will also toggle. Both *q1* update and *div_clk* toggle will happen after infinitesimally small delay (of *clk* ***posedge***) – but at the same time stamp, because, they both have *NBA*. At the positive edge of *div_clk* (which happens after a while), the second ***always*** block gets triggered. Since this has been triggered after a while (actually, towards the end of the current time-stamp), *q1* could already have been updated. So, the updated value of *q1* will reach *q2*. Thus, *t1* has overshot *q1* and has gone into *q2*.

There is one more scenario (sequence of events) possible. This possible scenario is left up to you to work out. In that scenario, feedthrough would not happen.

One possible solution to this kind of situation is to avoid using *NBA* in clock-paths. In this case, the clock-path (generation of *div_clk*) has an NBA. If this NBA is replaced by the Blocking Assignment, the second ***always*** block will trigger instantaneously, before *q1* is updated. So, the correct way of writing the above functionality would be:

```
always @ (posedge clk)
q1 <= t1;

always @ (posedge div_clk)
q2 <= q1;

always @ (posedge clk)
div_clk = ~div_clk;
```

The example mentioned in Section 2.2.1.2 has two possible interpretations. One of those interpretations results in a *feedthrough*.

2.3.2 Feedthroughs Without Simulation Race

Feedthroughs can happen even if there is no simulation race. Consider the following *Verilog* code segment:

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

The above code has no ambiguity, but, there is a feedthrough. *d* goes into *c*, and then into *b* and then into *a* – all within the same cycle. This is usually not, what was intended. The solution to this situation is simple. Use *NBA*!!!!

2.3.3 VHDL Feedthroughs

VHDL has a concept of *delta-delay*. The simulator assigns an infinitesimally small delay (called *delta*) whenever there is an assignment to a *signal* (assignments to *variables* are instantaneous). Consider a clock signal *Clk1*. *Clk1* is used as a master to derive another clock signal *Clk2*. *Clk2* is delayed from *Clk1* – by a few *deltas* (these, additional deltas could be due to clock-gating – for example). The following code-excerpt shows this situation:

```
process (Clk1)
begin
SIG1 <= Clk1; -- one delta from Clk1
Clk2 <= SIG1; -- Two deltas from Clk1
end process;
```

Clk1 is also used in the sensitivity list of a ***process*** block that updates a signal *q1*. There will be one delta – for generation of *q1*.

```
process (Clk1)
begin
if Clk1'event and Clk1='1' then
q1 <= data;  -- one delta from Clk1
end if;
end process;
```

Clk2 is used in the sensitivity list of another ***process*** block (excerpt below) that samples *q1*. Because, *Clk2* is delayed by a few deltas, by the time the below mentioned ***process*** block is triggered; *q1* is already updated. So, the value of *q1* sampled in this ***process*** block would be the new-updated value.

```
process (Clk2) -- will trigger 2 deltas after Clk1
begin
if Clk2'event and Clk2='1' then
q2 <= q1;  -- q1 is already updated!!
end if;
end process;
```

In this case, *data* will move to *q1* and then into *q2*, all within the same cycle of *Clk1*. So, there has been a *feedthrough*. You can avoid this, by ensuring balancing of delta-delays across the clock network. That means, on all the clock-paths, there would be exactly same number of deltas. So, *Clk1* and *Clk2* must have the same number of deltas. That will ensure that *q1* cannot be created before *Clk2*. And, if *Clk2* is created from *Clk1* – as in the above example, then, *Clk1* is not used directly to update *q1*. Rather, *Clk1* is delayed further by the required number of (in this case, 2) deltas – and that delayed version is used to update *q1*. However, there are three issues with this approach.

- Most of the designs today use Mixed-Language. And, in Mixed-Language, there is no Language Reference Manual (LRM) – to define the behavior. So, there are some differences in the behaviors of different simulators, with respect to counting of deltas – especially as a signal crosses VHDL/Verilog boundary. So, it is possible that what is balanced in one simulator is no longer balanced in another simulator.
- It is too much of a trouble to keep counting all these deltas along various clocks.
- This requirement of balancing of deltas along clock-networks is much more stringent than what's really needed. What's really needed is: Deltas along: Clk1 --> q1 --> reaching onto the sampling signal have to be more than the deltas along Clk1 --> Clk2. So, if there is a minor difference in the delta-count along two clock-paths, but, there are enough delta-counts on the data-line, then, the clock network is as good as balanced.

Alternately, you may put a small delay while assigning a value to a signal, which will be used subsequently in another ***process***. The concept is shown in the following example code-snippet:

```
process (Clk1)
begin
if Clk1'event and Clk1='1' then
q1 <= data AFTER 1 NS;
end if;
end process;
```

This explicit ***AFTER*** clause ensures that *q1* is updated only after its older value is sampled in another ***process*** block. Effectively, this ***AFTER*** clause is acting as if it's creating a very high number of deltas in the data-path. This solution has its own risk, that, if the circuit's functionality was dependent on this specific value of explicit delay, the circuit might finally fail; because, synthesis will not honor this explicit delay assignment.

In fact, some designers always want to avoid explicit delay assignments – so that they inadvertently do not create a delay-dependent functionality. Besides, using explicit delays could cause the simulation to slow down considerably. On the other hand, some designers always want to use explicit delay assignments – so that they don't get into feedthrough conditions – due to mismatch of deltas.

2.4 Simulation-Synthesis Mismatch

Now that you have written your RTL in a manner, that there is no scope for ambiguity, you can simulate your design to ensure that it achieves the functionality that you desire. However, you have to ensure that not only the RTL should show the functionality that you desire, but, even the gate-level netlist that you will obtain from it after synthesis will also continue to exhibit the same functionality. Otherwise, there is no use, if RTL exhibits one functionality, but, the realized gate level shows a different functionality.

Simulation-Synthesis Mismatch refers to a situation, wherein, a given RTL showed some simulation results; however, when the same RTL was synthesized, and, the realized gate-level netlist was simulated using the same vectors, it exhibited a different behavior. Some of the most common reasons for Simulation-Synthesis Mismatch include:

- *Races* (as explained in Section 2.2)
- *Explicit Timing in RTL*. Say, the RTL code had an explicit delay assignment. When synthesis is done, the functionality is realized using technology gates. The delay for these technology gates could be totally different from what was

specified in the input RTL. And, if the functionality was dependent on the delay values, then, it is possible that the same functionality might no longer be visible – after synthesis, as the delay values have now changed.

- *Missing sensitivity list.* The RTL code-segment below is for a combinatorial block that misses some signals in the sensitivity list:

```
always @ (a or b)
if (sel)    z = a;
else        z = b;
```

  Here, while simulating the RTL, if *sel* was changed, the **always** block would not be triggered, and, hence, *z* will not exhibit the new value. But, after synthesis, this above code will become a MUX. And, when the gate-level circuit is simulated, as soon as *sel* changes, the value of *z* will be updated. Thus, the simulation results on gate-level design could be different from what was seen from RTL simulation. Use of *SystemVerilog* construct ***always_comb***. It is again helpful here – as there will be no need to explicitly specify the sensitivity list.
- *Delta unbalancing*. It is possible that the RTL was written using balancing of Deltas (as explained in Section 2.3.3). However, during synthesis, the tool did its own adjustments in terms of inserting some additional buffers (to meet the load requirements), or, removing certain gates (to improve the timing). These adjustments could modify the delta-balancing. Usually, this is not encountered too often, because, most designers use Verilog for gate-level and this issue of delta-balancing does not come into picture.
- *Initial Block*. Synthesis simply ignores the ***initial*** block. So, if an ***initial*** block is used to achieve a desired functionality at RTL stage, then, the realized gate-level netlist will not exhibit the same functionality. Usually, ***initial*** block should be a part of the testbench – used to initialize the circuit through external inputs, rather than being a part of the design itself. This same testbench will then be able to initialize the gate-level circuit also.
- *Dependency on X*: *x* or *X* is something that is available only in modeling. In the actual hardware, there is no such thing as *x*. So, if a dont_care value is being assigned or checked for in the RTL, the same behavior might not be visible in the synthesized gate-level netlist.
- *Comparison with Unknown*: A comparison with *x* or *z* could result in simulation-synthesis mismatch. *x* does not have any counterpart in the physical world. *z* means tri-state, but, usually, in the physical world, there will be some value – available on the net. So, comparison with tri-state would not have any meaning in the physical world. Similarly, if a design is dependent on use of === or ***!***==, or ***casex*** or ***casez***, there could be a mismatch, because, these comparisons consider *x* and *z* as if those were also a value. In the world of *VHDL*, *U* or *W* related comparisons could also result in simulation-synthesis mismatch, because they also don't have a physical-world equivalent that could be realized using logic gates.

- *Careless use of variables*: In *VHDL* RTL code, ***variables*** are not allowed by language to cross ***process*** boundaries. Consider the following code segment:

```
process(rst_n, clk)
variable v1_var : std_logic;
begin
v1_var := '0';
if (rst_n = '0') then
q <= '0';
elsif (clk'event and clk = '1') then
q <= data;
v1_var := '1'; -- simulator will respect this, but, synthesis will ignore
end if;
sig1 <= v1_var; -- sig1 will show simulation/synthesis mismatch
end process;
```

  The variable *v1_var* stays within the ***process***, but, goes outside the "*clock*"ed process. Synthesis will ignore the second assignment to *v1_var*, while, simulation will assign a value of *1* to *v1_var*. This can also result in simulation-synthesis mismatch. This mismatch will then be propagated to *sig1*, which is reading the value of *v1_var*. Reusability Methodology Manual (written jointly by *Mentor Graphics* and *Synopsys*) recommends against using ***variables***, because of their potential to cause such simulation-synthesis mismatches.

As an RTL designer, you need to avoid all the above mentioned situations and constructs – which can result in simulation-synthesis mismatch.

2.5 Latch Inference

Synthesis tools infer a latch, when a register is updated in some branches of a procedural block, but, not in all branches of the block. There might be instances of very large and complex procedural blocks, with a huge number of branching. Due to some omission, it is possible that for a specific register, it misses an assignment in some of the branches. This will result in synthesis tool inferring a latch. The following code excerpt will infer a latch:

```
always @ (a or b or c or d or e)
begin
if (a)
    begin
    if (d)    r = 1'b0;
    else      r = 1'b1;
    end
else
```

```
if (b)
begin
        if (e) r = 1'b0;
end
else
        if (c) r = 1'b1;
end
```

In the above code segment, though, not apparently visible – there are two branches, where, the value of *r* is not updated:

(i) *a* is not true; *b* is true; and *e* is not true.
(ii) *a*, *b*, and *c* are each individually false.

Because of missing assignment to *r* in these two branches, latches will be inferred. Latches can create complications in DFT (explained in Chapter 6). Hence, most RTL designers want to avoid latches (unless, the design is supposed to be latch based). Thus, you need to ensure that your RTL code does not infer a latch unintentionally.

SystemVerilog allows the use of keyword ***always_comb*** (instead of ***always***) to denote intent of combinatorial logic. And, if you want to infer a latch, you should use the keyword ***always_latch***. Software tools are allowed to flag a warning, if they find that the logic being inferred does not match the intent specified through ***always_comb*** or ***always_latch***. Thus, use of ***always_comb*** can warn you against unintentional latch inference.

Latches can also be inferred through ***case*** statements. Consider a 2-bit signal *sel* – which can take 3 values, viz: *00*, *01* or *10*. So, the case statement is written as:

```
case (sel)
2'b00: out = data1;
2;b01: out = data2;
2'b10: out = data3;
endcase
```

A synthesis tool might not be aware that *sel* cannot take the value *2'b11*. So, it will create a latch for the branch (*sel=2'b11*) – for which *out* has not been assigned any value. Synthesis tools allow pragmas embedded in the RTL (such as: *full_case*) to let the synthesis tool know – that all the possible values have been specified. This pragma tells the synthesis tool not to infer a latch for missing branches. However, if you miss a branch by mistake, you would not get any indication.

SystemVerilog has introduced a keyword – ***priority***, which serves multiple purposes. It is not a pragma; rather, it is a part of the language. So, this is understood not just by synthesis tools, but, also by the simulators and formal tools. And, if any tool sees that the case-selection variable takes a value that is not specified, it will give an Error. Consider the following code segment with the use of ***priority*** keyword:

```
priority case (sel)
2'b00: out = data1;
2;b01: out = data2;
2'b10: out = data3;
endcase
```

In this case, synthesis tool will not infer a latch. And, during simulation, if *sel* is found to take a value of *2'b11*, the simulator will give an Error. Just the presence of ***priority case*** does not mean complete protection against unintentional latch inference. ***priority case*** only means all the possible branches are specified. It is still your responsibility to provide the values to all the variables in all the branches. Consider the following code-excerpt:

```
priority case (sel)
2'b00: {out1, out2} = {data1, data2};
2;b01: {out1, out2} = {data3, data4};;
2'b10: out1 = data5;
endcase
```

The ***priority*** keyword only tells that all the possible branches have been specified. But, *out2* has not been updated in one branch. So, a latch would still be inferred for *out2* – despite using the ***priority*** keyword.

2.6 Synchronous Reset

Consider a flop *q* which can be cleared synchronously through assertion of signal *rst_n* (active-Low). This can be modeled in many ways. Some of them are given below:

```
always @(posedge clk)
if (!rst_n)
    q <= 1'b0;
else
    q <= d;
```

Or:

```
always @(posedge clk)
if (!d)
    q <= 1'b0;
else
    q <= rst_n;
```

Or:

```
always @(posedge clk)
q <= d & rst_n;
```

Or:

```
assign temp = d & rst_n;

always @(posedge clk)
q <= temp;
```

Or:

```
always @(posedge clk)
case (rst_n)
   1'b0: q <= 1'b0;
   1'b1:  q <= d;
endcase
```

All the various code-excerpts given above will behave the same – in terms of functional simulation. For the given example code-excerpts, even the synthesized netlist will have the same circuit realization as shown in Fig. 2.2.

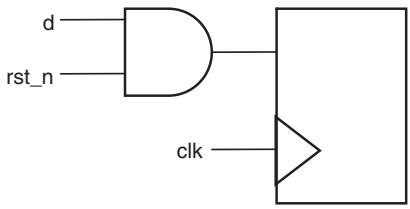

Fig. 2.2 Synchronous reset

As can be seen from some of the code-excerpts as well as the circuit realization, *d* and *rst_n* are interchangeable. If these signals were named something less meaningful (say: *a* and *b*), there is no way for a tool (or, even a human being) to distinguish between data and synchronous reset – atleast in some of the styles. Thus, it might appear that there is no need to distinguish amongst data and synchronous reset – because simulation results as well as the synthesized circuit are anyways the same. In terms of simulation results, there is actually no need to distinguish between data and a synchronous reset. However, synthesis tools try to treat data and synchronous reset signals slightly differently.

Certain ASIC libraries mark a synchronous reset pin through an attribute. Synthesis tool would try to connect the synchronous reset signal of the design to

such pins. In the absence of such a pin, the synthesis tool uses data pin itself to achieve the synchronous reset functionality. But, it still attempts to put reset as close to the flop as possible. This is because, reset being one of the control signals – synthesis tools try to have minimal combinatorial logic on it. The presence of synchronous reset is reported as part of register inference. For Synopsys® synthesis tool, the report would look something like:

Register Name	*Type*	*Width*	*Bus*	*MB*	*AR*	*AS*	***SR***	*SS*	*ST*
q_reg	*Flip-flop*	*1*	*N*	*N*	*N*	*N*	***Y***	*N*	*N*

However, if the synthesis tool is unable to identify a signal to be synchronous reset, it might not be able to give the differential treatment to synchronous reset, and treat those signals at par with data. If synchronous reset is not detected, the report would look like:

Register Name	*Type*	*Width*	*Bus*	*MB*	*AR*	*AS*	***SR***	*SS*	*ST*
q_reg	*Flip-flop*	*1*	*N*	*N*	*N*	*N*	***N***	*N*	*N*

Because of this reason, synthesis tools should be indicated which signal acts as a synchronous reset. Consider the following code-excerpt:

```
always @(posedge clk)
if (!rst_n)
    q <= 1'b0;
else
    q <= data1 & data2;
```

Figures 2.3a and 2.3b show two possible realizations of the above functionality.

While both the circuits are functionally equivalent, a synthesis tool should prefer to realize the circuit as shown in Figure 2.3a, where, the synchronous reset is closer to the flop. However, for this, the synthesis tool has to know, which signal is synchronous reset. This can be conveyed through a pragma (*sync_set_reset* for

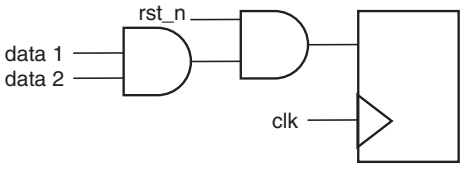

Fig. 2.3a Preferred realization

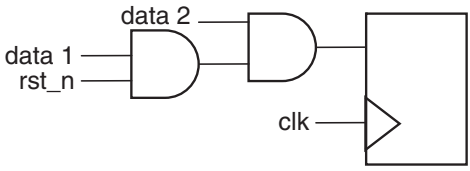

Fig. 2.3b Alternative realization

Synopsys synthesis tool) embedded in the RTL. Even with the pragma embedded in the RTL, the synthesis tool still depends upon the RTL structure to identify which signal is reset and which one is preset. For proper recognition of synchronous reset signals, it is important to have both the pragma as well as the right structure of the RTL.

In order to clearly communicate the intent of synchronous reset both for human understanding as well as synthesis inference, it is always best to code synchronous reset, using the style shown in the code excerpt below:

```
//pragma sync_set_reset rst_n
always @(posedge clk)
if (!rst_n)
    q <= 1'b0;
else
    q <= d;
```

This results in the following inference report:

```
=========================================
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================
|    q_reg      | Flip-flop |  1  |  N  |  N  |  N  |  N  |  Y  |  N  |  N  |
=========================================
```

If we remove the *sync_set_reset pragma* from the above code, the inference report would get changed to:

```
=========================================
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================
|    q_reg      | Flip-flop |  1  |  N  |  N  |  N  |  N  |  N  |  N  |  N  |
=========================================
```

Within the synchronous (i.e. clocked) portion of a sequential block, the first level *if* has to be the synchronous reset and then the entire data logic comes in the *else* section. This ***if*** condition should be for the asserted state of the synchronous reset signal.

The discussion mentioned in this section applies equally to synchronous preset also. In case both reset and preset are present, the first ***if*** condition has to be for the dominant of the two. The other synchronous signal should appear in the ***else if*** condition. And, the data logic should appear in the ***else*** condition. The code-excerpt below shows an example with both asynchronous and synchronous preset and reset. Both the signals are active Low and preset dominates over reset. In most cases, not all of these branches need to be present.

```
always @(posedge clk or negedge async_reset_n or negedge
async_preset_n)
  if (!asynch_preset_n) // Asynchronous preset – most dominant
     q <= 1'b1;
  else if (!async_rst_n) // Asynchronous reset
     q <= 1'b0;
  else begin // start of synchronous/clocked portion
    if (!synch_preset_n) // Synchronous preset – dominant
     q <= 1'b1;
    else if (!sync_rst_n) // Synchronous reset
     q <= 1'b0;
    else // data assignment
     q <= d;
    end
```

2.7 Limitations of Simulation

Simulation is the most popular and one of the most reliable methods used for validating that the HDL code meets the desired functionality. However, you should understand the major limitations of simulation. Some of the most important limitations are:

- As explained in Section 2.2, sometimes the same RTL code can be interpreted in multiple ways by a simulator. So, if your code is written in a manner that can give multiple results, your simulator will pick any one of those interpretations. Your simulation might pass with that interpretation, while the actual functionality realized could be different.
- Simulation based verification's effectiveness is limited to the quality of vectors and the quality of monitors being applied. With designs being so huge and with many storage elements, it is simply impossible to exercise the design for all possible situations. Vectors decide how the various parts of the circuits are being exercised. So, if a faulty portion of the circuit is not even being exercised, simulation will not be able to catch the fault. Monitors refer to what are you observing or checking for during the simulation. So, even if a faulty portion has been exercised, unless you are checking for something that depends on the values on that portion of the circuit, you will not be able to detect the fault.

- Even though, simulators have a concept of timing, they are not self-reliant at timing based verification. At the RTL level, most designs don't have timing. Even if there is some timing mentioned, it is not necessary that the same timing would be realized after the design is synthesized. So, RTL simulation does not give any sense of timing – other than being just accurate at the level of clock-cycle. And, at gate level, HDL languages do not provide for a good timing model. The best that they provide is *specify* block for Verilog and *VITAL* (*VHDL Initiative for Timing in ASIC Libraries*) for VHDL. These mechanisms are mostly place-holders for actual timing numbers. They don't do any delay calculation themselves. In order to do accurate timing simulation on a gate-level design, you have to do the delay calculation outside simulation, and bring back the delay values to the simulator through SDF back-annotation. This concept is discussed in the next chapter. So, for accurate timing simulation there is a dependence on an external delay calculation mechanism.
- Besides functionality, there are a lot of other aspects (test, power, routing congestion, etc.), which are important for a design. Simulation does not do anything to validate any of these aspects of the design. Though with the advent of some new formats (CPF/UPF) to specify power related intent, simulators can now do some validation of power aspects.

Because of these limitations, it's now a standard practice to use static rule checkers as part of the verification flow. These tools will verify many aspects of the design without the need for accurate vectors. These static checkers could be assertion based formal tools or rule-based checkers. In both cases, significant errors can be found that would otherwise go undetected.

http://www.springer.com/978-1-4419-9295-6

Principles of VLSI RTL Design
A Practical Guide
Churiwala, S.; Garg, S.
2011, XV, 182 p., Hardcover
ISBN: 978-1-4419-9295-6