designAbstraction Limited

Common HDL Design Errors

1.1 Introduction

Visit the American National Transportation Safety Board web-site (www.ntsb.gov) and you will discover that there are two *standard ways* to kill oneself in an aeroplane. One of those is known as *controlled flight into terrain*, basically flying oneself into the side of a hill. The second is the *stall/spin* accident. Both types of accident are well understood, easily avoided and great efforts are made to educate pilots in their prevention. Nevertheless, they reoccur with depressing regularity. It seems that the hard-won lessons of previous generations are not always learned.

Nobody, certainly not I, would attempt to compare the importance of writing HDL code without making basic errors with the seriousness of avoiding aeroplane crashes. Nevertheless, there are number of *standard errors* that are repeated over and over again by legions of HDL design engineers. This is in part, I believe, because there is no equivalent official drive to promulgate good practice within the HDL design community. The purpose of this article then, is to introduce a few of the very basic faults, to describe the problems that they produce, and to suggest alternative, arguably superior, methodologies. In many cases, these errors are made at the foundation level. Consequently, everything that is built upon them thereafter is, shall we say, sub-optimal.

1.2 Design Issues

This section of the document addresses problems often seen within RTL code, i.e. that code that is used to generate the design itself, usually with the aid of a synthesis tool. Section 1.3, on the other hand deals with common test bench errors.

1.2.1 RTL friendly specification

Each stage in a chip-design process builds upon work done in the previous stage. The higher one builds, so to speak, the harder it is to make changes to the supporting foundations or structure. Nowhere is this more important than in the production of the initial design specification. You do write one? Many designs proceed with little or no formal documentation. Whilst acceptable (just) for a one-man design, a properly written specification is the only way to achieve a quality design that functions correctly first time.

The advantages of having a properly written specification are manifold:
1. Each member of the design team knows what he/she is expected to produce.
2. The specification provides something against which the RTL may be reviewed.
3. Testing is performed with reference to the specification, not against the RTL. There is an argument for a proper test specification to be written though, especially in the case of FPGA designs (as opposed to ASICs), this is not always done. Anyhow, the design specification provides the basis of the test specification.
4. The specification provides a full description of the device behaviour. At the sample testing stage it should not be necessary to refer to the RTL.

The specification should always be archived for future reference.

It should be possible to code a design by reference to the specification (in combination with any documents referenced therein). However, the straightforwardness of that process may be enhanced if the specification is written to be *coding friendly*. Here are a few suggestions towards that end:

1. Define all external interfaces. Name all inputs, outputs and bi-directional signals. These names should be used verbatim in the RTL.
2. Define the device architecture, naming each module. This architecture should appear within the RTL with all names maintained.
3. Define all internal communication paths, in particular any internal buses. If possible, name all the associated signals. Describe the purpose of these communication paths and what kind of information flows along them.
4. Describe, preferably in words and pictures, the operation of each sub-module.
5. Name all memory-mapped registers using appropriate mnemonics. These names should appear within RTL.

Clearly the specification represents the thoughts of the designer at an early stage in the project. Naturally, corrections and improvements will be made as the design progresses. It is paramount that these changes find their way into the specification and it gets re-issued. That is something that is frequently overlooked in the pressure to move onto the next project and can be the cause of headaches for engineers who are unfortunate enough to inherit the design in the future.

1.2.2 Maintain naming conventions.

This used to be one of my most common bugbears though, I admit, most designers are now getting the message. There is nothing more infuriating that chasing through a design trying to trace a signal whose name changes at each level in the hierarchy. This problem more frequently occurs when design is done bottom up. In other words, each block is written in isolation by a different engineer. It is not until the top-level integration stage that the mismatches become apparent, by which time it is frequently too late, or too onerous, to fix the problem, especially, as the project is sure to be running late!

A corollary of the previous point is that names are best chosen in terms of what signals do, rather than where they come from, how many bits they have or whatever. Such information can easily be determined from the code. What is nowhere near as obvious is what a signal actually does or even why it exists at all.

To further aid clarity, it is also, in my opinion, good practice to avoid abbreviating wherever possible. Such techniques help to make the code reasonably self-documenting.

1.2.3 Concatenate memory-mapped registers, don't chop them up.

This is probably the single most frequently occurring RTL faux pas. Perhaps the design specification has defined an eight bit memory-mapped register as shown in Figure 1.

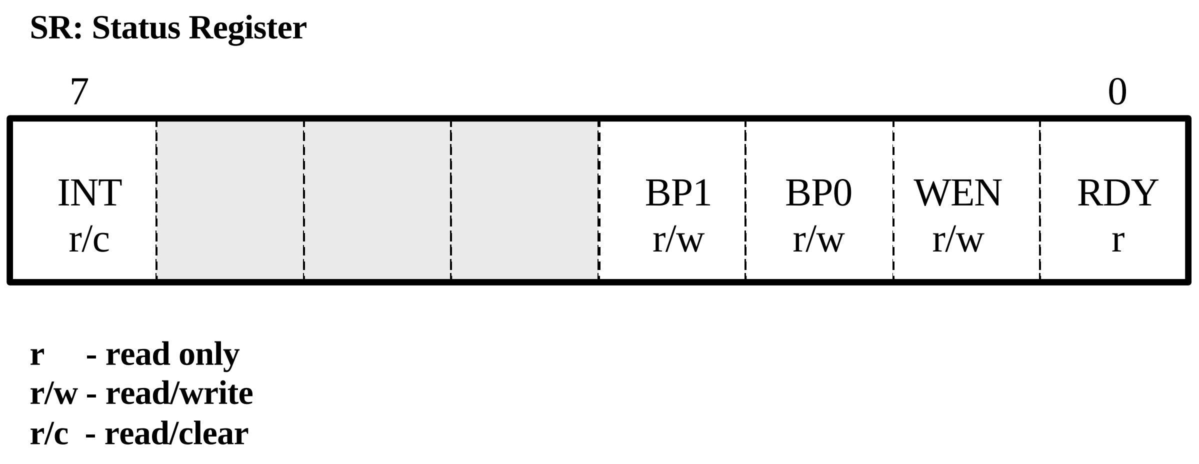

Figure 1 Example Register Definition

The designer might choose to code this up as shown in the following Verilog code fragment (note that I have omitted any reset functionality for reasons of simplicity. Module I/O is not shown either but should be obvious):

```
reg [7:0]   SR;
parameter c_SR = 8'h0;
parameter  RDY = 0,
           WEN = 1;
`define    BP    3:2
parameter  INT = 7;

always @( posedge clk )
    begin

    // Handle reading of regiser SR.
    if( read && address==c_SR )
        begin
        readData = SR;

        // Clear INT whenever it is read.
        SR[INT] = 1'b0;
        end

    // Handle writing to register SR.
    if( write && address==c_SR )
        { SR[`BP], SR[WEN] } = writeData[3:1];

    // Set INT flag whenever a particular event happens.
    if( particularEvent )
        SR[INT] = 1'b1;
    end
```

Doesn't look too bad? But there are three problems with it as follows:

1. Reg SR has been declared as an eight bit quantity even though only five locations are used. Three of these bits are necessarily unused. With luck, the synthesiser will realise and eliminate the three bits but this is by no means certain.
2. It was necessary to declare the address of register SR as the constant c_SR to avoid a name clash. No disaster but a little ungainly.
3. It was necessary to *chop up* the SR register twice, once when writing it to avoid assigning the unused bits, and once when setting the INT bit. This is particularly bad because it locks the code to the particular memory map. For example, it would be awkward to move the INT bit into a different location. In the case of large fluid designs, the problem can become totally unmanageable. Far better to declare the individual fields individually and then concatenate them into the address map as follows (using VHDL this time):

```
    clockedLogic:  process ( clk )

        constant SR:   std_logic_vector(7 downto 0) := x"00";
        variable RDY,
                 WEN:  std_logic;
        variable BP:   std_logic_vector(1 downto 0);
        variable INT:  std_logic;

        begin

        readData <= (others=>'0');  -- Default assignment

        -- Handle reading of regiser SR.
        if read='1' and address=SR then
            readData(7)          <= INT;
            readData(3 downto 2) <= BP;
            readData(1)          <= WEN;
            readData(0)          <= RDY;

            -- Clear INT whenever it is read.
            INT := '0';
        end if;

        -- Handle writing to register SR.
        if write='1' and address=SR then
            BP  := writeData(3 downto 2);
            WEN := writeData(1);
        end if;

        -- Set INT flag whenever a particular event happens.
        if particularEvent='1' then
            INT := '1';
        end if;

        end process clockedLogic;
```

Note now that the address map is entirely defined by the sections dealing with reading and writing. All other functionality, in this case setting of the INT bit, is independent of the address map. INT is handled as a stand-alone field; there is no need to continually chop it out of the SR register. Moreover, there are no unused bits declared that, we hope, the synthesiser will eliminate.

1.2.4 Distribute control & status registers.

A commonly used architectural technique is to centralise all memory-mapped registers into a single *processor interface* module. It is my belief that this method is adopted because of its perceived simplicity in comparison to the alternative which is to distribute the registers into the various architectural blocks to which they pertain. This in turn necessitates the provision of a simple internal read/write bus for access to them. However, the overhead involved with setting up such a bus is minimal whereas the dividends it pays in terms of clarity are manifold.

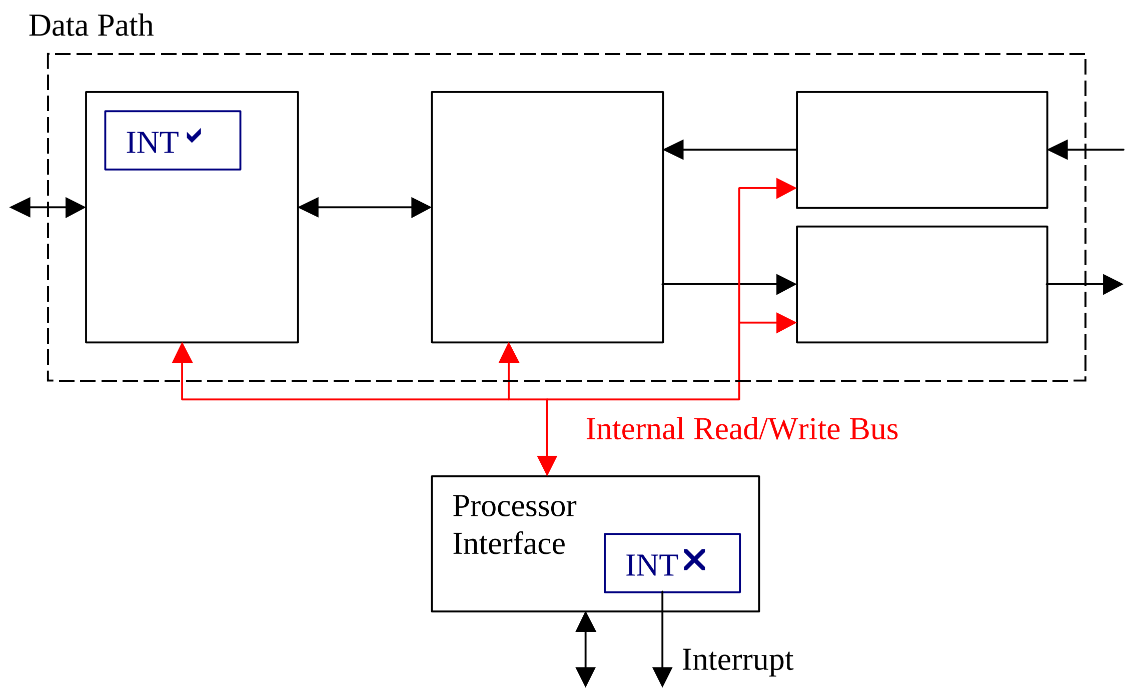

Figure 2 Example design using an internal read/write bus.

The examples shown in section 1.2.3 assume a simple read/write bus.

Consider the simple design hierarchy shown in Figure 2. The INT bit represents an interrupt. There are two places it might reside. It could be sited within the processor interface (denoted ✗) or it could be placed within the module in which the event that causes the interrupt will occur (shown with a ✓). The examples shown in section 1.2.3 assume the latter. It can be seen how cleanly the INT bit is handled, principally because all of its behaviour is encompassed in the single *clockedLogic* process. Nevertheless, many designers choose the former location. But consider the difficulties:

1. When the particular event that sets the interrupt occurs, the detecting module must signal to the processor interface to set the interrupt. This requires a dedicated wire. The processor will then set the INT bit.
2. When the processor reads the SR register (in which the INT bit resides), INT gets cleared. The INT bit must then be wired back to the detecting module so that it knows that the interrupt has been cleared and it can again start looking for the interrupt causing event. It is essential that the INT bit is cleared and the interrupt detection circuitry is re-armed on exactly the same clock edge. If this is not the case, there will exist an intermittent, and very difficult to find, bug whereby, if an interrupt causing event happens close to the point of the interrupt being read, there is a possibility of the interrupt being missed. Placing the interrupt setting and clearing circuitry in the same always block (Verilog) or process (VHDL) by locating the INT bit within the detecting module (✓) makes this task much easier.

The other obvious disadvantage of centralising memory-mapped registers is that the processor interface needs a connection for every single one. Not only is this very untidy, it makes the addition or removal of bits or fields unduly onerous. This is because it is necessary to edit (in the case of VHDL) two entities and two architectures in order to implement the bit and connect it up. By contrast, an extra bit can be added locally with a couple of lines of code.

1.2.5 Inside out code.

Here's a segment of not untypical Verilog:

```
parameter S1     = 1,
          S2     = 2,
          S3     = 3,
          S4     = 4,
          ACTIVE = 5;

reg [2:0] state;

always @( posedge clk )
    case( state )

        S1:     if( enable && sig1 )
                    state = S2;
                else if ( enable && !sig1 )
                    state = S1;
                else
                    state = S1;

        S2:     if( enable && sig1 )
                    state = S3;
                else if ( enable && !sig1 )
                    state = S1;
                else
                    state = S2;

        S3:     if( enable && sig1 )
                    state = S4;
                else if ( enable && !sig1 )
                    state = S1;
                else
                    state = S3;

        S4:     if( enable && sig1 )
                    state = ACTIVE;
                else if ( enable && !sig1 )
                    state = S1;
                else
                    state = S4;

        ACTIVE: if ( enable && !sig1 )
                    state = S1;

    endcase
```

Ostensibly ok. But take a closer look and you will notice two common mistakes. Firstly, there is a lot of repetition. I call this *inside out code* because the repeated sub-test should really be the outer test. Whenever an engineer finds himself repeating the same lines of code, he should wonder whether he can enhance its coniseness by re-nesting the tests.

1.2.6 Redundant Code

The other fault that is readily observed is the fact that each state contains a default assignment to itself. These should be removed because they confuse the reader who will have to mentally score them out anyway. Let's re-write the fragment more concisely (in VHDL this time):

```
clockedLogic:  process ( clk )

    constant S1:     std_logic_vector(2 downto 0) := o"1";
    constant S2:     std_logic_vector(2 downto 0) := o"2";
    constant S3:     std_logic_vector(2 downto 0) := o"3";
    constant S4:     std_logic_vector(2 downto 0) := o"4";
    constant ACTIVE: std_logic_vector(2 downto 0) := o"5";
    variable state:  std_logic_vector(2 downto 0);

    begin

    if enable='1' then

        case state is

            when S1     => if sig1='1' then
                               state := S2;
                           end if;

            when S2     => if sig1='1' then
                               state := S3;
                           end if;

            when S3     => if sig1='1' then
                               state := S4;
                           end if;

            when S4     => if sig1='1' then
                               state := ACTIVE;
                           end if;

            when ACTIVE =>

            when others =>

        end case;

        -- Override previous assignments.
        if sig1='0' then
            state := S1;
        end if;

    end if;

    end process clockedLogic;
```

1.2.7 Minimise Hierarchy.

It is quite common for designs to be made up of large numbers of very small blocks. This is inefficient in terms of the amount of boilerplate that must be written to produce a given amount of functionality. Furthermore, it is very difficult for any engineer other than the original designer to read. This is due to the requirement to *chase* signals up and down the hierarchy in order to determine what they do and how the various modules interact. I would also argue that it is inflexible. Iteration is a normal part of the design process. Distributed behaviour like this is, by its nature, intrinsically woven into the hierarchical structure. Modifications will very often entail, not just changes to behaviour but also lots of re-wiring, a tedious job (alleviated somewhat if appropriate schematic capture tools are available).

In general, hierarchy should really only be introduced for one of two reasons:
1) It is unavoidable, for example when repeated functionality is required;
2) there is an obvious need for it, for example to separate distinct and independent functions.

It is not generally a good idea, as is commonly seen, to introduce additional hierarchy to separate simple functions such as counters, multiplexers, state machines, ALUs and so on. To do this is to miss the point of VHDL or Verilog.

1.2.8 Don't separate control and data hierarchically.

The previous section recommended not separating out state machines from other logic. This is at odds with quite frequently accepted practice. The reason for this recommendation is simply that to do so introduces far to much dependence on hierarchy, is very inflexible and quite simply achieves nothing for the addition of considerable complication. A common design technique is to create a data-path module, possibly pipelined, that is controlled by a number of control signals (or handles) as shown in Figure 3.

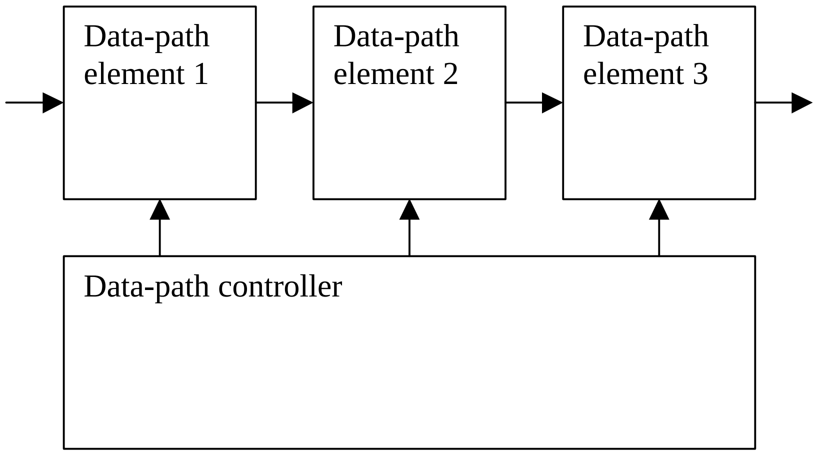

Figure 3 Data-path with separate controller

Quite apart from the complexity of separating control and data, such techniques soon run out of steam, especially when there is not a linear relationship between the amount of data entering a link in the chain and the amount leaving. Typical data-path elements that would prove difficult to control externally like this might include packetisers, frame alignment modules, CRC checkers, pattern generators, pattern recognition modules, encoders, decoders, modulators, demodulators, error detection and/or correction units and so on. Better by far simply to embed the data-path into the control logic state machines. Inter-module control flow can then be achieved by means of hand-shaking. The whole system then becomes self timing.

The beauty of such an approach is its flexibility and robustness. Individual elements in the data-path might have two or more modes of operation with totally different data throughputs as well as sporadic data rates. Imagine say a module whose job it is to detect a particular frame alignment pattern within a data-stream, discard it and pass on the resulting payload data (example used in section 1.2.8.1). Changes may also be made quickly in the light of simulation or perhaps as a result of failing to meet synthesis timing constraints. For example, it would be straightforward to add an additional pipeline delay into one of the data-path elements. In the case of an external controller, an increase in the latency of a particular stage would necessitate a commensurate additional state delay within the controller.

There is an exception to this rule which is when it is intended to use replication and custom layout techniques to produce an optimised physical data-path. Such techniques are quite rare however. If the intention is to implement the data-path using simple random logic (e.g. Standard Cell, Gate Array or FPGA), it is quicker and more elegant to embed the data-path into the control logic.

1.2.8.1 Example

There follows a simple example data-path element (Figure 4) whose function is to detect a frame-alignment pattern (Table 1) in an incoming data-stream, extract the payload (D_0, D_1, D_2…D_{n-1}, D_n) and status information (S) and pass them on to the subsequent data-path element. The module has its data-path embedded within the state machine and the output data makes use of the data abstraction techniques described in section 1.2.9.1.

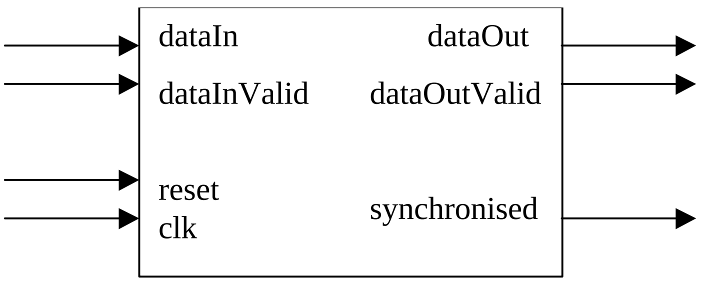

Figure 4 Example Frame Alignment Module

1	0	1	1
0	D_0	D_1	D_2
0	D_3		D_{n-2}
0	D_{n-1}	D_n	S

Table 1 Example Frame Alignment Pattern

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
PACKAGE frame IS
type   t_dataOutValid is ( IGNORE, DATA, STATUS );
END frame;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

LIBRARY EXAMPLE;
USE EXAMPLE.frame.all;


ENTITY frameAligner IS
    PORT(
        clk          : IN     std_logic;
        reset        : IN     std_logic;
        dataIn       : IN     std_logic;
        dataInValid  : IN     std_logic;
        dataOut      : OUT    std_logic;
        dataOutValid : OUT    t_dataOutValid;
        synchronised : BUFFER std_logic
    );
END frameAligner ;

ARCHITECTURE RTL OF frameAligner IS

BEGIN

clockedLogic: process (clk, reset)

    variable count:   integer range 0 to 15;
    variable match:   boolean;

    begin

    if reset = '1' then
        count        := 0;
        synchronised <='0';
```

```
            dataOutValid <= IGNORE;
        elsif clk'event and clk='1' then

            dataOutValid <= IGNORE;   -- Default

            if dataInValid='1' then
                match := true;        -- Default

                case count is
                    when 0         => if dataIn /= '1' then
                                          match :=  false;
                                      end if;
                    when 1         => if dataIn /= '0' then
                                          match :=  false;
                                      end if;
                    when 2         => if dataIn /= '1' then
                                          match :=  false;
                                      end if;
                    when 3         => if dataIn /= '1' then
                                          match :=  false;
                                      end if;
                    when 4         => if dataIn /= '0' then
                                          match :=  false;
                                      end if;
                    when 8         => if dataIn /= '0' then
                                          match :=  false;
                                      end if;
                    when 12        => if dataIn /= '0' then
                                          match :=  false;
                                      else
                                          synchronised <= '1';
                                      end if;
                    when  5| 6| 7|
                          9|10|11|
                         13|14     => if synchronised='1' then
                                          dataOut      <= dataIn;
                                          dataOutValid <= DATA;
                                      end if;

                    when 15        => if synchronised='1' then
                                          dataOut      <= dataIn;
                                          dataOutValid <= STATUS;
                                      end if;
                    when others    =>
                end case;

                if match=false then
                    synchronised <= '0';
                    count        :=  0;
                else
                    count        := (count+1) mod 16;
                end if;
            end if;
        end if;

    end process clockedLogic;

    END RTL;
```

1.2.9 Don't handle by timing what can be better handled as data.

There is a tendency for HDL designers to break down problems in terms of their timing. For example, the status bit, S, in the previous problem might be used to provide a separate communication channel with a bandwidth one eighth that of the main data channel. An engineer might therefore feel the need to synchronise it to a clock that runs at an eighth of the speed of some main data clock before passing that re-timed channel onto some further logic. Such a constraint would probably be entirely arbitrary however, and would most likely add unnecessary complication. Far better to deal with data entirely asynchronously whenever possible. If the data happens to arrive with some regular period, regard that as a bonus, not a requirement.

Such an approach is in many ways analogous to the difference between interrupt routines and background processing in *real time* software. Robustness and simplicity is enhanced be doing as much work as possible in the background. Only constrain timing when critically necessary.

1.2.9.1 Data Abstraction

The example design uses data abstraction techniques to differentiate between data and status. Both are extracted from the incoming data-stream. However, instead of passing them out through two discrete wires, both are passed out on *dataOut* annotated by the value of *dataOutValid* (either DATA or STATUS).

The technique works particularly well when a device has to work in one of several, mutually exclusive, modes of operation. The same physical wires may be used to transfer multiple different types of information at numerous data rates. By this means, a single data-path can perform as many functions as are required thereby avoiding the need for multiple data-paths with untidy multiplexing to switch between them. Each element in the data-path would most likely consist of one or more state machines, each capable of several programmable operating modes.

1.2.10 Write in terms of what it does, not what's in it.

There is a reluctance within the design community to trust the synthesiser to do its job. The upshot is that many engineers write VHDL or Verilog at a much lower level of design abstraction than necessary. In essence, most RTL defines the *transfer function* for any given piece of logic, in other words how the next state of the circuit depends upon the inputs to the circuit and its present state. That, therefore, is the optimum level at which to write. In general, it is unnecessary, indeed unhelpful to explicitly write in term of gates. A common example is address decoding. Engineers sometimes declare specific logic to do this task in the belief that it will produce an optimum implementation. However, the key to reducing such logic is to apportion the address map with care. The RTL should, where possible, declare addresses by means of easily understood mnemonics. It is the mapping of said mnemonics to hard numbers that should be hand-crafted, usually in the form of a package (VHDL) or include file (Verilog). This applies also when choosing state assignments.

1.2.10.1 Artifacts

Apart from expediency, coding up a piece of logic in terms of its function makes the finished design far more comprehensible to other engineers. A corollary is the need to avoid artifacts. What are artifacts? Consider the following code segment:

```
assign weekday = day==MONDAY    ||
                 day==TUESDAY   ||
                 day==WEDNESDAY ||
                 day==THURSDAY  ||
                 day==SUNDAY;

assign weekend = day==SATURDAY || day==SUNDAY;

always @( weekday or weekend )
    if( weekday )
        alarmClock = 1'b1;
    else if( weekend )
            alarmClock = 1'b0;
```

Regs *weekday* and *weekend* are artifacts. More often than not, when such artifacts are introduced they are divorced from the behaviour to which they pertain forcing the reader to conduct a search. Admittedly, they are sometimes unavoidable, for example the variable *match* in the frame alignment example (1.2.8.1). If they are used, then variables (VHDL) or regs (Verilog) are preferable to signals (VHDL) or wires (Verilog) because they have to be assigned *in-line* and therefore read better.

The following idiom would, I contend, be preferable:

```
always @day
    case( day )
        MONDAY    |
        TUESDAY   |
        WEDNESDAY |
        THURSDAY  |
        FRIDAY:    alarmClock = 1'b1;
        SATURDAY  |
        SUNDAY:    alarmClock = 1'b0;
    endcase
```

In some cases, this methodology leads to bloated code. Functions, tasks (Verilog) or procedures (VHDL) can help to keep things succinct:

```
function isItTheWeekendYet;
    input day;  reg [2:0] day;
    case( day )
        MONDAY    |
        TUESDAY   |
        WEDNESDAY |
        THURSDAY  |
        FRIDAY:    isItTheWeekendYet = 1'b0;
        SATURDAY  |
        SUNDAY:    isItTheWeekendYet = 1'b1;
    endcase

endfunction // isItTheWeekendYet

always @day
    alarmClock = !isItTheWeekendYet( day );
```

Note that all three versions will synthesise identically.

1.2.11 Banish asynchronous logic to the periphery.

In general, asynchronous logic is bad news. A design will usually consist of one or more *isochronous zones*, i.e. zones running at a given clock frequency. Timing analysis within such zones is straightforward, a simple check of set-up (critical path) and hold times (shortest path) being all that is required. The places where timing analysis becomes more problematical are (i) at the interface between timing regimes and (ii) at the interface to the outside world. Such areas usually require careful crafting. It is good practice to keep any asynchronous logic close to the periphery and locate as much functionality within the synchronous domains as possible.

1.2.12 Design out test logic.

In the FPGA world, manufacturing test logic is not required; the manufacturer will have taken care of it. It is much more significant when designing ASICs. The predominant manufacturing test methodology is scan test. Test logic is sometimes added to devices to facilitate scan test of the not-fully-synchronous sections of the design. For example, multiplexers might be added to allow internal clock signals to be controlled by the tester. Similarly, gates might be added to asynchronous *presets* and *clears* to prevent them transitioning as vectors are scanned in and out. I would argue that test logic ought to be thought of as a last resort to be avoided if possible. Sometimes, careful crafting of the design itself can obviate the need for additional logic.

An example I remember was a control/monitoring interface within an otherwise synchronous design. The obvious way to have implemented the interface would have been to gate the address lines and write strobe together and clock the data using the resultant signals (Figure 5). However, this produced a *derived clock* that was iherently untestable. It would have been necessary to either multiplex in an externally controllable clock for test purposes or to generate custom functional test vectors. By using the write strobe as a clock and using the address and chip-select pulses as enables (Figure 6), the design became inherently scan testable without any test logic. On the tester, the write strobe was simply treated as another scan clock.

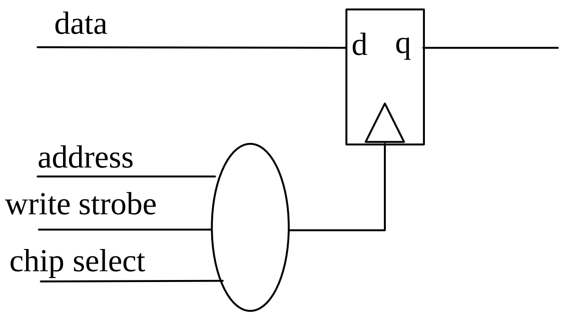

Figure 5 Untestable architecture

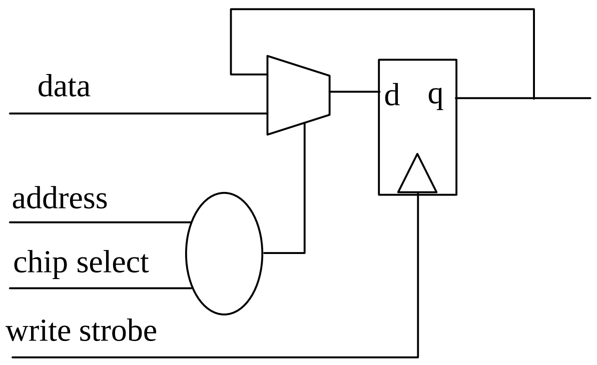

Figure 6 Functionally equivalent but inherently testable alternative

1.2.13 Think Layout

It is not essential to make the logical hierarchy of a design match the physical hierarchy. Many designers do not give a thought to physical layout until very late in the design flow. I would argue that it should be thought about even as early as the specification/block diagram stage. The key thing to remember is that an IC is, by its nature, a two-dimensional entity. If possible aim to create a design that can be laid out like floor tiles. Wires connecting adjacent tiles will be short, easy to lay out with small propagation delays. Conversely, wires connecting modules at opposite sides of the chip will have larger propagation delays that are more likely to produce timing problems. Schematic capture tools are very helpful in visualizing the ease of layout of a given design. The block diagram shown in Figure 2 would probably prove straightforward to lay out. The design abstraction techniques highlighted in section 1.2.9.1 are also likely to minimise connectivity. Careful pin assignment should be used to keep core to output pad distances short. Basically, if the design can be drawn easily in two dimensions on a screen, it can probably be laid out equally easily on silicon.

1.3 Test Bench Issues

This section introduces a few common errors seen in test benches. The basic object of a test bench is to exercise the functionality that has been implemented in the RTL. Any self-respecting test bench will be self checking. That is, it will stimulate the device under test and check its resulting behaviour against a predicted response. Any errors will be recorded to the transcript. Such errors I refer to as *run-time errors.* The beauty of this method is that it avoids the need to examine simulation waveforms manually and pays particular dividends at the regression test stage. That is when tests are repeated many times as the design is tweaked and bugs are ironed out.

1.3.1 Run-time v data processing

The self checking technique starts to run out of steam when the function of the device under test is predominantly concerned with *data-transformation*. For example, consider trying to test a GSM receiver. Clearly, the test bench would need to understand the intricacies of demodulation, echo cancellation, de-interleaving, error correction and so on. Though this can be done, it is probably a better bet to take this task outside the realms of the VHDL or Verilog simulator. A common technique is to use a 'C' model for generating and checking test data.

Said data is read into the VHDL/Verilog simulation and applied to the device under test. The resultant output data is written out into different files. These are then either checked manually (in simple cases) or post-processed (possibly using the original 'C' model) to deem correct operation. The basic concept is shown in Figure 7.

This avoids the need for too much intelligence to be built into the test bench itself. Of course, it is perfectly sensible, in the case of very complex functions, to test sub-modules hierarchically. For example, in the above case, it would be sensible to check say the Viterbi decoder as a stand-alone function.

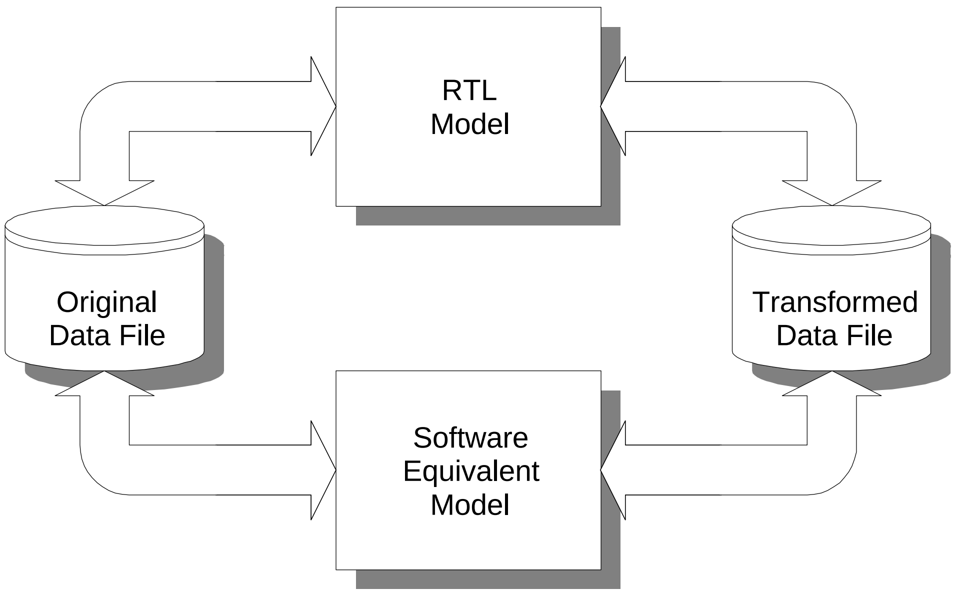

Figure 7 Use of a software model to generate and check test data.

1.3.2 Stimulate and check together.

A common mistake, usually brought about by the desire to make rapid progress with the RTL, is to leave the checking side of the test harness until later with the thought that, "I'll add it later". Consequently, the checking functionality is regarded as a separate function. This is a mistake and leads to fundamentally flawed test benches. Always, always stimulate and test together. Separate data checking modules are rarely fully satisfactory. In particular, it becomes unwieldy to orchestrate the behaviour of stimulus and checking parts of the test bench with respect to one another.

Pretty much the simplest device to test is the humble inverter. Here is a test bench for an inverter written according to these guidelines:

```
procedure checkInverter(
                       signal a: out std_logic;
                       signal z: in  std_logic
                       ) is
    begin
    a <= '0';
    wait for 5 ns;
```

```
compare( z, '1', "check inverter" );
a <= '1';
wait for 5 ns;
compare( z, '0', "check inverter" );
end checkInverter;
```

The procedure *compare* is defined in a package called testHarnessUtilites.vhd and is available from the Design Abstraction web-site (http://www.designabstraction.com/). It may be freely used and distributed so long as the copyright message is retained. A similar Verilog equivalent, testHarnessUtilities.v is also available. The simulation transcript produced by calling the function is as follows:

```
#    5.000 ns  <<< PASS >>> check inverter;  expected 1 and indeed found it.
#   10.000 ns  <<< PASS >>> check inverter;  expected 0 and indeed found it.
```

It is impossible to check the output of a circuit at all times. In general, I tend to think of a test bench as a series of gates through which the device under test must pass, rather like a canoe slalom coarse (Figure 8). The path taken between *gates* is therefore irrelevant. The inverter test, above, for example allows 5ns before checking the output. It doesn't matter whether the propagation delay of the inverter is 1ns or 4ns, it will still pass this test (though not if it takes 6ns).

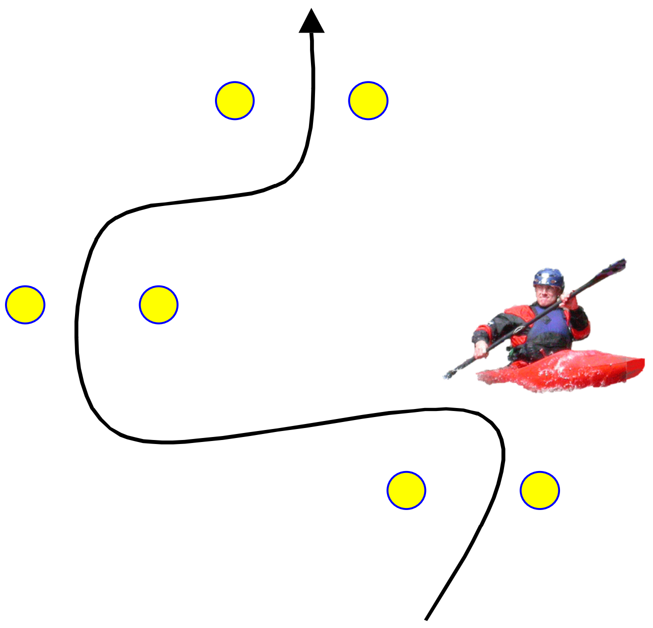

Figure 8 Test Bench Techniques!

In Verilog, most of the utilities required to create a quality self-checking test bench are built into the language. The include file testHarnesUtilities.v consists mainly of the function *compare*. By contrast, it is much less straightforward to produce messages to the simulation transcript using VHDL. That is why package testHarnessUtilities.vhd is considerably more comprehensive.

1.3.3 Avoid file driven run-time control.

My absolute favourite gaffe, as regards test bench design is concerned, is the file-driven test bench. Typically, an engineer will create a file format to drive a microprocessor interface, perhaps something like this (I've made this up so don't try and make sense of it!):

```
write  status   XX110101
delay  5
write  tx       10101010
read   control  00100001
read   control  00100001
```

```
delay  20
read   control  00100011
write  status   XX110100
write  tx       01010101
read   control  00110001
delay 2
read   rx       00011000
```

This file clearly causes the test-bench processor model to go through a sequence of read and write operations to various registers within the device under test (perhaps it is a UART). In the case of writes, the data to be written is supplied whereas for reads, the expected data is supplied. "Seems reasonable", I hear you say. But there are several major disadvantages that, to my way of thinking, make this methodology particularly painful:

1. It is necessary to write a parser to interpret the file, a non-trivial task that achieves little in itself. The parser would require a routine to recognise the various commands in the file. Recognition of a particular command would then trigger an associated task (Verilog) or procedure (VHDL). Why not simply call such procedures straight from the test harness?
2. It is necessary to predict everything that the device under test will do and when it will do it because there is no mechanism for intelligence within the test bench. It might be argued that this is a good feature. It is certainly very inflexible. Consider polling say an *Rx Ready* flag until it rises. It would most probably be necessary to run the simulation once to find out when it will happen and then hard code the delay into the control file. Far better simply to build a bit of intelligence into the test harness using a simple loop. Granted, if the flag never rises, the test bench might poll forever but that would soon be noticed and fixed.
3. Every time a new instruction is required, the parser has to be enhanced to recognise it.
4. It is very difficult to coordinate the timing of the processor interface to what happens on other interfaces to the device under test (for example the data-path inputs and outputs).
5. It is impossible to run individual threads simultaneously. The Verilog *fork/join* construct is fantastic for this though it can equally be achieved using VHDL (with a little more difficulty).

Some might argue that using files allows the development of multiple different tests. This is true…just. But more often than not, what distinguishes one test from another is very often the test data itself (different data rates say). The control side of things will usually be pretty similar aside from the initial configuration of the device. All in all, file driven test-bench control achieves, with considerably less finesse, what VHDL procedures or Verilog tasks can achieve directly.

The use of this particularly bad technique is virtually endemic across the industry. Why, I know not. My suspicion is that many engineers are hell bent on using file I/O to prove how clever they are, irrespective of its usefulness! Notice that I am not against file I/O per se. It's great for handling data (section 1.3.1) but it's manifestly inadequate as a means of coordinating the operation of a test bench.

1.3.4 Test for correct operation.

Opinion differs on what constitutes full and comprehensive test of a piece of design work. I would argue that the requirement is satisfied by a two part solution:

1. Exercise the device-under-test through its full range of features as defined in the specification. This includes any error conditions which are specifically described as requiring a particular response. For example, the specification might require that an interrupt be set in the event of a CRC failure. Or it might be specified that an incorrectly formatted data packet be discarded.
2. Peer-to-peer review the RTL against said specification and satisfy oneself that the RTL is capable of meeting the specification but that it has no redundant functionality.

That's it. I content that if one meet these two requirements, one will get bug free silicon every time. It is not necessary to attempt to predict ways by which a piece of code might fail and test for that. Apart from anything else, there are an infinite number of ways in which a failure might occur so it would require an inordinate amount of tests to achieve what can much more reliably be achieved by review.

1.3.5 Always build tests upward from the lowest common denominator.

This point might seem obvious. Nevertheless, I have seen a surprising number of cases where this rule is violated to the considerable future inconvenience of the perpetrator. For example, a given design might deal with say packets of data comprising header byte, justification bits, payload data and a CRC byte. This would naturally lend itself to the use of a structure (e.g. a VHDL record) to handle packets as single entities. Should it be necessary to deal with several packets of data, then it might be convenient to define an array of such structures. What would not be good practice, though I have seen as much on numerous occasions, would be to define a conglomerate structure able to hold say five packets because that is the number of packets required by the particular test in hand. Sooner or later, it would become necessary to do something that required three packets and, of course, no suitable structure would be available.

Similarly, in the case of designs that interface to a microprocessor (which tends to be most designs), it is good practice to provide a test bench task (Verilog) or procedure (VHDL) to perform reads and writes. These are then strung together in the test bench as necessary to stimulate the device under test as if the test bench was a piece of *firmware.* Seems obvious enough but I've seen people define a task to say write two bytes because that was what was needed at the time. Why not simply call the write task twice?

Incidentally, I always provide the Read/Write task with three modes of operation as shown in Table 2. Note that the very powerful READ_RETURN mode opens up a whole host of capabilities simply not possible in a file driven test bench.

Mode	Function	Purpose
WRITE	Writes data to the processor interface.	Self explanatory.
READ_CHECK	Reads data from the processor interface and checks it against an expected value.	Use when the value read can be predicted beforehand.
READ_RETURN	Reads data from the processor and returns it to the calling routine.	Use when the test bench needs to react to the value read, for example to poll a particular bit until its value changes.

Table 2 Read/Write Tasks

1.3.6 Think about gate-level timing

When designing a test bench to test the RTL model, remember that the test bench will later be used to simulate the synhthesised gate level net-list. So even though, at RTL, it is possible to change inputs on the active (usually rising) edge of the system clock using signal assignments (VHDL) or non-blocking assignments (Verilog), this will no longer work with real gate delays. Therefore, the test bench would need to be modified to achieve cycle-identical results. This problem can be avoided by changing data inputs and sampling outputs away from the active edge of the clock. The simplest expedient is to use the inactive (falling) edge.

The other purpose in not changing inputs close to the active edge of the system clock is that it will avoid set-up and hold violations when running gate level simulations. In the case of truly asynchronous inputs (where double buffering is used to re-time the signal to the clock), this is artificially optimistic. Set-up and hold violations would be expected in real life and should equally be expected during gate-level simulation. The transcript should be examined to ensure that all such warnings are only produced by registers where this is known to be the case and nowhere else.

1.3.7 Think about test vector generation.

Many ASIC vendors require the user to supply functional test vectors in addition to (or instead of) scan vectors. These must run on the IC testing machine and are expected to be in a format known as *cycle based* whereby each vector is applied to the device under test for a single cycle period. Any particular input is always changed at a fixed point in the cycle. Similarly, each output is strobed (or tested) at a fixed point in the cycle also.

By far the easiest way to produce cycle based vectors is to use the original test bench. This means that the test bench must be persuaded to run in a cycle-based mode. It is then a simple matter to write out a print-on-change file (using for example $dumpfile followed by $dumpvars in Verilog) and translate it into the Vendor's required format using the translation tools that they should also supply.

Instructions on how to translate print-on-change files into the required format will usually be supplied by the Vendor. The key point that I am trying to stress is to give some thought to this process at an early stage of test bench creation.

It is a good idea to only run the test bench in cycle based mode for the purpose of generating vectors. In normal mode, input timings should be more representative of real life. When working in verilog, this is easily achieved using conditional compilation.

1.4 Conclusion

This article is necessarily subjective. Nevertheless, there are some observations here that might provide some insight into the kind of problems that engineers typically cause themselves. Rules, as we know, are for the guidance and wise men and the obedience of fools. The wise man, however, acquaints himself with the received wisdom before flaunting it. Feel free to ignore the contents of this article. But I invite you to try some of these techniques in your next design. You might like them.

Glossary

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CRC	Cyclic Redundancy Check
FPGA	Field Programmable Gate Array
RTL	Register Transfer Level
Rx	Receive
UART	Universal Asynchronous Receiver Transmitter
Verilog	
VHDL	Very High Speed Integrated Circuit Hardware Description Language