Chapter 3: An Example of Reducing RTL Design Complexity in Control-Dominated Designs

Mike Keating
3/28/2008

This chapter provides an example of how to reduce the complexity of the RTL design. It describes a project to rewrite a module and raise the level of abstraction without changing its functionality.

The project started with a module from a wireless USB design. This module initiates and controls the DMA from the MAC layer to the Amba AHB bus.

We use the following metrics to assess our success in simplifying the RTL code:

• number of lines of code (number of words, number of characters)
• internal state space of the design
• number of objects in the design
• amount of abstraction, partitioning, and information hiding in the design
• the overall readability of the code

One constraint of the project was that we could not change the interface of the module. To test the design, and prove that we had not changed its function, we needed to be able to interface to the other blocks in the existing wireless USB design. For this reason we are not addressing the issues of input and output state space size.

The techniques that we used to simplify the code include:

• partitioning the code into multiple files
• eliminating some syntactic fluff from the code
• restructuring the sequential code, moving it all into a single state machine
• using functions for the combinational code
• converting the state machine to a single, sequential process
• converting the state machine to a hierarchical state machine
• using the system Verilog struct construct to encapsulate important structures

Original Code

The initial design was chosen because it was one of the largest modules in the design and reasonably complex. It was also well-designed and well-coded; that is, it was designed and coded in compliance with the best common design practices. Any improvements would have to move beyond these practices and possibly require changes to the Verilog language.
The original Verilog2k code consisted twenty-seven pages for a total of 1600 lines of code (total lines of text including comments and blank lines), including:

- 2 pages of input and output declarations
- 4 pages of internal declarations (wires, regs, parameters)
- 7 sequential processes (always @(posedge clk))
- 10 combinational processes (always @*)
- 30 assign statements

The sequential processes included:
- One process that was 5 pages long (a state machine)
- Another process that was 3.5 pages long
- One process that was 1 page
- The rest of the sequential processes were less than 1 page each

The combinational processes included:
- One process that was 2 pages long
- One process that was 1 page long
- Rest of the processes were less than 1 page

Thus, of the 27 pages,
- 6 pages (22%) were declarations
- 12.5 pages (46%) were large processes

These statistics are useful because declarations, although certainly necessary, are largely a distraction when trying to read and debug code. Large processes (more than 1 page) are more difficult to read and understand than small processes. A state machine that takes 5 pages is very challenging to read and understand.

The rewrite project consisted of three steps:
- Analyze and understand the code
- Rewrite the code
- Test the new code in the wireless USB test environment

During the analysis phase, we did not consult with the original designer. We wanted to see how easy it was to understand the function of the design just from the code (and a very high-level specification document).

During testing, we were not able to use formal verification to show that the new version and old version were equivalent. We restructured the sequential code such that the two designs were different (from the perspective of formal verification) but functionally equivalent, in terms of passing all the regression tests.
State Space in the Original Design

The original state machine consists of 12 states.

The next two largest sequential processes assign values to 44 registers and 9 registers respectively. On first reading the code, it is not obvious which combinations of these 53 registers can occur and which are impossible. Thus, the state space of these two processes is $2^{53}$.

Without more information, we have to assume that any of these $2^{53}$ states can occur during any of the 12 states of the state machine.

Thus, the total state space, just of the state machine and these two processes, is $12 \times 2^{53}$. In human terms, this is essentially infinite.

The original designer, of course, intended a much smaller state space. In fact, the original designer would see most of the variables in the sequential processors as data or temporary variables rather than real states. Because of his understanding of the design intent, the effective state space for the original designer is much smaller than the state space as actually represented in the code.

During the analysis phase of the re-write project, one key goal was to restructure the code to make it easier to understand the state space. We could then rewrite the code to minimize the state space and make it explicit and obvious.

Partitioning

The first step in restructuring the design was to partition it into several files. This allowed us to aggregate related code into separate code segments that could be analyzed and rewritten independently.

We partitioned the design into the following files:
- input and output declarations
- internal declarations of wires, registers and parameters
- all combinational code (with a few exceptions, explained below)
- all sequential code (with a few exceptions, explained below)
- the top level code, consisting of:
  - the module declaration
  - `include statements to include the other files
  - combinational code that directly drives primary outputs
  - sequential assignments that simply pipeline primary inputs

With this partitioning, virtually all the real action of the module was contained in two files: the combinational code and the sequential code. During the subsequent work we could largely ignore the other files, referring to them only as required.
Comments

The next step was to eliminate most of the comments from the code.

```verilog
// Flip Flop registers for Byte Stripping and Alignment
always @(posedge hclk `DWC_UWB_HCLK_GRST_MODE) begin
    if (!hreset_gen_n) begin
        rxf_saved_data <= 0;
        rxf_saved_data_cnt <= 0;
    end else begin
        if (fsm_rxqreq_vld_clr ||
            (fsm_dmareq_vld_clr && ((fsm_q_sel == RXQ) ||
            (fsm_q_sel == DWQ)))
            begin
            rxf_saved_data_cnt <= 0;
        end else if (fsm_rxf_saved_data_load || (bium_wdata_pop &&
            (rxf_data_shift != 0))) begin
            // bium_fifo_busy not needed since no pop on BUSY
            // load RXF data register if:
            // - FSM instructs to do so
            // - or BIUM pops RXF and byte shifting is required
            rxf_saved_data <= bcu_rdata[`DWC_UWB_BUSWIDTH-1:8];
        // determine number of valid bytes saved in RXF data register based on
        // shift value
            case (rxf_data_shift[1:0])
                2'b00: rxf_saved_data_cnt <= 'h0;
                2'b01: rxf_saved_data_cnt <= 'h3;
                2'b10: rxf_saved_data_cnt <= 'h2;
                default: // 2'b11
                    rxf_saved_data_cnt <= 'h1;
            endcase
        end else if (bcu_rxf_pop || bcu_dwf_pop || bcu_rxf_pktcnt_dec ||
            (fsm_dmareq_vld_clr && (dmareq_f == DWF))) begin
            // clear num valid bytes if we don't load RXF data register
            // this clock and:
            // - BCU pops RXF (last entry)
            // - or BCU decrements RXF packet counter (upon DMA completion or RX
            // pkt flush)
            rxf_saved_data_cnt <= 0;
        end
    end
end
```

Note that the comments, although they contain some interesting information, completely interrupt the flow of the code. Scattering comments throughout a process actually makes reading the code harder. Instead, we moved the useful comments ahead of the process, so all the comments could be read at once, and then all the code read without interruption.

But we ended up eliminating many of the comments, because they simply restated what the code says.
Syntactic fluff

Preprocessing Sequential Code

Once the design was partitioned into manageable sized files, and extraneous comments removed, we still found the code difficult to read and understand. One of the most obvious causes of this was a large amount of syntactic fluff in the code. Consider the following (very small) sequential process.

```verbatim
always @(posedge hclk `DWC_UWB_HCLK_GRST_MODE) begin
  if (!hreset_gen_n) begin
    csr_debug_bcu1_lo <= 4'b0;
    bium_abort <= 1'b0;
  end
  else begin
    csr_debug_bcu1_lo <= {bcu_dwf_pop, bcu_rxf_pop};
    if (bium_dma_done)
      bium_abort <= 1'b0;
    else if (bium_abort_set)
      bium_abort <= 1'b1;
  end
end
```

This process consists of 13 lines of code. But all the meaningful action is described in lines 7 through 11. So five lines of code are of high-value, and the other seven are overhead. All they do is get in the way of understanding the code.

Note: the macro `DWC_UWB_HCLK_GRST_MODE` is used to allow the same code for both asynchronous reset and synchronous reset applications.

To make the code simpler and more readable, we developed a simple Perl script to act as a preprocessor for the RTL code. In this module, as in the vast majority of modules in the design:

- there is only a single clock and a single reset
- non-blocking assignments are used for sequential code, and only for sequential code
- only flip-flops are used, never latches

With a Perl script in place, the sequential process above can be rewritten in a streamlined fashion as:

```verbatim
csr_debug_bcu1_lo <= {bcu_dwf_pop, bcu_rxf_pop};

if (bium_dma_done) bium_abort <= 1'b0;
else if (bium_abort_set) bium_abort <= 1'b1;
```
The Perl script uses the “<=” as an indication that this is a sequential process, and re-writes the code as:

```verilog
always @(posedge hclk `DWC_UWB_HCLK_GRST_MODE) begin
  if (!hreset_gen_n) begin
    csr_debug_bcu1_lo <= 4'b0;
  end
  else begin
    csr_debug_bcu1_lo <= {bcu_dwf_pop, bcu_rxf_pop};
  end
end
```

This code is exactly equivalent to the original code.

Note that to re-generate legal Verilog code from the streamlined code, three additional pieces of information are needed:

1. the name of the clock
2. the macro describing reset
3. the reset values of the registers

In the streamlined code, all three of these are provided in the signal declarations by the following code:

```verilog
bit [3:0] csr_debug_bcu1_lo = 0;
bit [3:0] bium_abort       = 0;
%clock hclk
%reset `DWC_UWB_HCLK_GRST_MODE
```

That is, we use the initialization construct to define the reset value. And we declare the clock and reset explicitly.

With this Perl script in place, we were able to dramatically reduce the size of the sequential code. More importantly, by eliminating the distraction of syntactic fluff, we made the code much easier to read.
Note that this approach is possible because the module in question uses only a single clock. But for the vast majority of modules in digital design, this is the case. Good design practices dictate that whenever signals cross clock boundaries a small, separate module is used and that only this module contains multiple clocks. The approach described here does not apply to such a multi-clock module, but does apply to all the other modules in the design.

**Preprocessing Combinational Code**

We used a similar approach to combinational code, which is a much simpler case. Here the Perl script recognizes that a blocking assignment is used (in this kind of module) only for combinational code.

So we can reduce the following combinational process:

```verilog
1 always @*
2 begin
3   next_txf_saved_data_dec = 0;
4   for (q=0; q<DWIDTH_LANES; q=q+1)
5     if (q < next_txf_saved_data_cnt)
6       next_txf_saved_data_dec[q] = 1'b1;
7   next_txf_saved_data_en = next_txf_saved_data_dec;
8 end
```

To this:

```verilog
1   next_txf_saved_data_dec = 0;
2   for (q=0; q<DWIDTH_LANES; q=q+1)
3     if (q < next_txf_saved_data_cnt)
4       next_txf_saved_data_dec[q] = 1'b1;
5   next_txf_saved_data_en = next_txf_saved_data_dec;
```

The Perl script generates the appropriate legal Verilog syntax, adding back the `always@*`, `begin`, and `end`.

Once we had partitioned the code and multiple files and eliminated the syntactic fluff, we found the code dramatically easier to read, analyze, and re-factor. We first focus on improving the sequential code.

**Rewriting Sequential code**

**Recoding the State Machine**

There are two classic ways of coding a state machine:
Two-Process SM

```verilog
case (state)
  IDLE: if (foo) next_state = bar;
  STATE1: if (bar) next_state = ...
endcase

always @(posedge clk) begin
  state <= next_state;
end
```

One-Process SM

```verilog
always @(posedge clk) begin
  case (state)
    IDLE: if (foo) next_state <= bar;
    STATE1: if (bar) next_state <= ...
  endcase
end
```

In the original Verilog, the state machine used the two-process approach. Although this is a common way of coding state machines, and approved in the RMM, it can, in fact, lead to poorly structured, hard-to-analyze code. For example, we found states in the combinational state machine processes that looked like:

```
STATE_XYX : begin
  if (foo) doit = 1;
end
```

And many sequential processes elsewhere in the code that said something like:

```
always @(posedge clk) begin
  if (doit) begin
    sign_yy <= 1’b1;
  end
end
```

We concluded that there are some fundamental problems with the two-process state machine approach:

1) The combinational code ended up generating many flags to other processes. These flags added clutter to the code, obscuring meaning instead of clarifying it.
2) The real action in any RTL code is the sequential behavior of the module – this dictates how the module behavior evolves over time. In the original design, the sequential code was scattered throughout 27 pages. There was no real structure to the sequential code.

In response to these observations, we converted the state machine to the one-process type of state machine, converting it to be a single sequential process. This required a significant amount of work, but the result was code that was much cleaner and easier to understand.
Relocating Other Sequential Code

Our next step was to examine the rest of the sequential code in the module. We found that there were basically two types of code. One type looks as follows:

```hsv
if (prev_state_bcu == RDQ1) && (state_bcu == RDQ2) begin
  case (fsm_q_sel)
    TXQ: begin
      txqreq_rr[fsm_txqnum_sel] <= bcu_rdata[F_RAO_RR];
      txqreq_tag[fsm_txqnum_sel] <= bcu_rdata[F_RAO_TAG];
    . . .
  endcase
end
```

In this case, it was straightforward to move this code into the state machine, as part of the code for state RDQ2.

The second type of sequential code looked as follows:

```hsv
if (fsm_dmareq_vld_clr && (fsm_q_sel == TXQ))
  txqreq_vld[fsm_txqnum_sel] <= 1'b0;
else if (fsm_txqreq_vld_clr)
  txqreq_vld[ccub_txfnum] <= 1'b0;
else if (fsm_dmareq_vld_set && (fsm_q_sel == TXQ))
  txqreq_vld[fsm_txqnum_sel] <= 1'b1;
```

In this case, it's not clear whether this code can be moved into the state machine. But a careful analysis of the design indicated that the critical control signals (fsm_dmareq_vld_clr, fsm_txqreq_vld_clr, and fsm_dmareq_vld_set) could only change value in one state. Therefore, this code could, in fact, be moved into the state machine.

In one or two isolated cases, a small piece of code could be active in more than one state. In these cases, a single piece of code had to be moved (copied) into two states. But in all cases, moving sequential code into the state machine made the design much easier to understand. For example, to analyze the original code, one would have to search the entire 27 page file to find all the places where fsm_dmareq_vld_clr is referenced and then try to determine how this interacts with the rest of the state of the module.

To create the new code, we had to do exactly this analysis. But once we completed this analysis, we were able to move this code into a single state in the state machine. As a result, this code became trivial to analyze.

Using these techniques, we moved virtually all the sequential code into the state machine.

These changes were the single most effective step we took in simplifying the code. As a result of moving the sequential code to the state machine, all the sequential behavior of the design could be analyzed and understood from examining a single, small file. The
state machine itself became simpler, and the overall code for the module became dramatically simpler.

**Rewriting Combinational code**

After cleaning up the sequential code, we turned to the file that contained all the combinational code. This was a considerable amount of code and quite complex to analyze. Our major concern was how to restructure this code so it would be easy to understand. How could we hide the information that we could hide and make obvious the information that needed to be visible. In other words, what was the preferred encapsulation method for combinational code.

In the end, we decided to use functions. We ended up rewriting virtually all of the combinational code in the form of functions. For example,

```verilog
function [DWC_UWB_DWF_FIFO_LOG2P1+DWIDTH_LO-1:0] dma_cnt_dwf (input [DWC_UWB_BUS_AWIDTH-1:0] rxqreq_addr,
input [W_RAO_LEN-1:0] rxqreq_len);

bit [DWC_UWB_DWF_FIFO_LOG2P1-1:0] dwf_fifo_depth = DWC_FIFO_DEPTH;
reg [1:0] rxqreq_stbytes;

case (rxqreq_addr[1:0])
  2'b00: rxqreq_stbytes = 2'b00;
  2'b01: rxqreq_stbytes = 2'b11;
  2'b10: rxqreq_stbytes = 2'b10;
  default: rxqreq_stbytes = 2'b01;// 2'b11
endcase

if (rxqreq_len <=
   ((((11~DWC_FIFO_LOG2P1){1'b0}), dwf_fifo_depth, rxqreq_stbytes)))
  dma_cnt_dwf = rxqreq_len[DWC_UWB_DWF_FIFO_LOG2P1+1:0] ;
else
  dma_cnt_dwf = {dwf_fifo_depth, rxqreq_stbytes};
endfunction
```

In this case, a great deal of the complexity in calculating `dma_cnt_dwf` is hidden within the function definition. It is explicitly stated that the function depends only on to external variables (`rxqreq_addr` and `rxqreq_len`). This kind of encapsulation can be very effective in partitioning very complex combinational code into bite sized chunks that can be analyzed separately.

One concern we had about the file containing all the functions was that it was fairly large, about eight pages. The question was how to organize this code.
In the end, we made the following observation. The key behavior in the module is really captured in the state machine. It describes how the behavior of the module evolves over time. On the other hand, the combinational code is really a set of definitions. For instance, the function above defines `dma_cnt_dwf` in terms of `rxqreq_addr` and `rxqreq_len`. And there is a well-established paradigm for ordering definitions, namely dictionary (alphabetical) order.

So the file of functions is simply ordered alphabetically. The function `dma_cnt_dwf` appears just after the function `ccub_txfnum` and before the function `dma_cnt_txf`.

### Analyzing the New Code

Now analysis of the overall design is quite simple. One starts with the state machine. From analyzing it, we understand the basic algorithm, the basic sequence, that the module executes. In fact, the top-level algorithm is contained in the idle state of the state machine (shown in slightly simplified form):

```plaintext
IDLE: begin
  if (stop_condition)
    state_bcu <= CCUB_STP;
  else if (ok_to_read_q()) begin
    state_bcu <= RDQ1;
    else if (dma_rdy.txf & !ccub_txf_stopped)
      state_bcu <= DMA_TX_DF;
    else if ((dma_rdy.rxf & !ccub_rxf_stopped) | dma_rdy.dwf)
      state_bcu <= DMA_RX_DW;
  end
```

That is, if the module is told (by an external resource) to abort the DMA, we do the abort (initiated by state CCUB_STP). Otherwise, if there is a DMA request in one of the queues, we read the queue, initiated by state RDQ1. Otherwise, if there is a DMA request pending (as a result of reading the queue) then perform the DMA. There is a clear priority to the DMA: TX (transmit) has priority over RX (receive).

We can then analyze each of these activities (abort, read queue, do dma) separately by examining the states that execute them. These states in turn refer to functions (like `ok_to_read_q`) which can be found in the functions file (in alphabetical order).

We have now defined a preferred structure for a module, consisting of a state machine, encapsulating the sequential code of the module, along with a set of functions, which encapsulate the combinational code. The various states of the state machine call these functions. The state machine is the primary component of the module, and the place where we start analyzing the design. The functions are secondary components, called only by the state machine (or by each other).
This structure provides a systematic way of reviewing and understanding the code. In particular, it allows this analysis to be performed top-down and in sections of manageable size.

**Restructuring for a Hierarchical State Machine**

Now that we have created an overall structure for the code, we can make one more improvement to the state machine. The state machine consists of three meta-states: IDLE (including some error handling), ReadQ (that fetches the DMA request), and the DMA process itself.

The complete state machine state diagram is shown below.
From the diagram, it is clear that the three basic functions are largely, but not completely, independent. The idle state and the abort state are largely independent of the DMA activity and the read queue activity. But the abort state does, under certain conditions, jump to the RDQ2 state. This is because under certain abort conditions it is necessary to flush the queue. Similarly, the DMA activity is largely independent from the other states, except that it sometimes jumps to the RDQ2 state. This is because, for a long DMA, the queue needs to be read multiple times before completing.

To make analysis easier, we partitioned the state machine code into three separate files:
Analysis was now easier, because the files were smaller, and each file contained only one activity. For the most common activities - a small DMA transfer or reading a queue - the analysis of the state machine is now quite straightforward. But for more complex activities, such as aborts and long DMA transfers, there's still significant interaction between the three meta-states.

For a relatively simple state machine such as this, these interactions between meta-states make analysis of the overall state machine only slightly more complicated. But for more complicated state machines, such interactions can make analysis intractable.

For this reason, we decided to convert the state machine into a truly hierarchical state machine. We wanted to structure the state machine to comply with the Rule of Seven; that is, so the state machine and each of its sub-state machines would have no more than seven to nine states. The top level of the resulting state machine looks like this:
The DMA state is now a meta-state consisting of its own (sub) state machine:

When the top level state machine enters the DMA state, it calls the DMA sub state machine. This machine always starts in the DMA_ENTRY state and ends in the DMA_EXIT state.

The ReadQ sub state machine looks like this:
Converting the state machine to a hierarchical state machine adds a few state transitions to some activities. For our design, the added clock cycles did not impact performance. But the resulting design is much simpler.

Analysis of the DMA sequence, for instance, is now completely self-contained, in one small file and in one simple (sub)state machine. Under all conditions, the DMA sequence starts in a well defined state (DMA_ENTRY) and exits from a single state (DMA_EXIT).

If we want to modify the DMA sequence, we can do it locally in the DMA state machine, and not worry about its impact on abort or the read queue sequence. Previously, any changes in the DMA sequence would have to look at the entire state machine to understand its impact.

On the other hand, if we want to analyze or modify the read queue sequence, we can now completely ignore the DMA sequence.

These are the classic benefits of hierarchy and encapsulation: a hierarchical state machine is inherently simpler to design and verify than a flat state machine.

The actual code for the hierarchical state machine is provided in Appendix XX.
System Verilog

The final modification that we made to the code was to employ the SystemVerilog construct "struct". We use this technique to encapsulate some of the common structures in the design. For instance:

The fields read from the receive (rx) queue were originally coded as:

```systemverilog
reg rxqreq_vld = 0; // DMA request valid
reg rxqreq_ds = 0; // Descriptor
reg rxqreq_so = 0; // Strip off valid
reg rxqreq_rr = 0; // Response required
reg [W_RAO_TAG-1:0] rxqreq_tag = 0; // Tag ID
reg [W_RAO_STRIP-1:0] rxqreq_stripcnt = 0; // Strip off byte count
reg rxqreq_last_seg = 0; // Last DMA addr segment
reg [W_RAO_LEN-1:0] rxqreq_len = 0; // Transfer byte length
reg ['DWC_UWB_BUS_AWIDTH-1:0] rxqreq_addr = 0; // DMA address
reg [DWIDTH_LO-1:0] rxqreq_staddr = 0; // Starting DMA address
reg rxqreq_first_seg = 0; // First DMA addr segment
reg rxqreq_first_dword = 0; // First dword of request
reg [W_RAO_LEN-1:0] rxqreq_rbc = 0; // Remaining byte count
```

This code becomes:

```systemverilog
struct packed {
  bit [DWIDTH_LO-1:0] staddr; // Starting DMA address offset
  bit first_seg; // First DMA addr segment
  bit first_dword; // 1st dword of DMA request
  bit [12:0] rbc; // Remaining byte count
  bit vld;
  //DWORD1:
  bit [1:0] unused1;
  bit ds; // Descriptor
  bit so; // Strip off valid
  bit [2:0] unused2;
  bit rr; // Response required
  bit [3:0] unused3;
  bit [3:0] tag; // Tag ID
  bit [2:0] unused4;
  bit [12:0] stripcnt; // Strip off byte count
  //DWORD2:
  bit last_seg; // Last DMA addr segment
  bit [17:0] unused5;
  bit [12:0] len; // byte len (excludes stripcnt)
  //DWORD3:
}
```

Copyright © 2009 Synopsys, Inc. All rights reserved.
bit [31:0] addr ;    // DMA address
//DWORD4 :
bit [31:0] unused6 ; // DMA addr - upper 32 bits – unused in this version
} rxqreq = 0;

In the original design, the names of the signals were used to show that they were related. In the new versions, the “struct” makes this relationship explicit.

In the original design, there was no visible relationship between the signal declaration and its position in the queue FIFO. We would have to look up values like W_RAO_TAG (in a different file) to find out where rxqreq_tag fits in the dword read from the FIFO, but we still wouldn’t know which word it is in (since a rx request requires reading four words from the FIFO). In the struct definition, all the fields are shown, including how many dwords are read, where each field is located in which dword, and any unused bits. Thus, there is much more, and much more useful, information in the new declaration.

This approach also makes for simpler code when we are reading data from the fifo. Instead of reading each field separately:

```
taxqreq_last_segment[fsm_txqnum_sel] <= bcu_rdata[F_RAO_L];
taxqreq_len[fsm_txqnum_sel] <= bcu_rdata[F_RAO_LEN:-W_RAO_LEN];
taxqreq_rbc[fsm_txqnum_sel] <= bcu_rdata[F_RAO_LEN:-W_RAO_LEN];
taxqreq_bium_start_segment_pending[fsm_txqnum_sel] <= 1'b1;
```

We can say:

```
taxqreq[fsm.txqnum_sel][`DWORD1] <= bcu_rdata;
```

In addition, the number of objects declared in the design (reg’s, wires, and, in the new version, struct) went from about 75 in the old version to 25 in the new version. Again, encapsulation has simplified the design.

**Summary**

We have used the following techniques to simplify the code:

- partitioning the code into multiple files
- eliminating some syntactic fluff from the code
- restructuring the sequential code
- using functions for the combinational code
- converting the state machine to a single, sequential process
- converting the state machine to a hierarchical state machine
- using the system Verilog `struct` construct to encapsulate important structures

The result has been:

1. We have reduced the internal (shallow) state space of the design from about $2^{56}$ to about $2^7$. (Shallow and deep state space are defined in the next chapter).
2. These states are encapsulated in a hierarchical state machine with 3 meta states; the biggest of the sub state machines has eight states.
3. We have provided a systematic encapsulation of sequential and combination code, allowing a simplified, systematic review process.
4. We have reduced the number of objects declared in the design by 67%.
5. We have reduced the number of lines of code by about 30%.
6. We have eliminated the syntactic fluff to make the function of the code much more obvious.

One of the interesting side-effects of the code restructuring is that it allows a very concise, complete and accurate drawing of the design.

The design (BCU) communicates with the BIUM, CCUB, PFC, CSR, and DCUB blocks. All the possible input/output transactions are shown in the yellow arrows. The eight data structures (fsm, dmareq, dma_redy, txqreq, rxqreq, deqreq, rxf_save, txf_saved) contain virtually all the signals in the design. All the sequential code is in the DMA State Machine. The only parts of the code not show are the functions (combinational code) referenced by the state machine.