

# Using Vera to Test a DMA Engine

Stefen Boyd

President, Boyd Technology, Inc.

stefen@boyd.com

## ABSTRACT

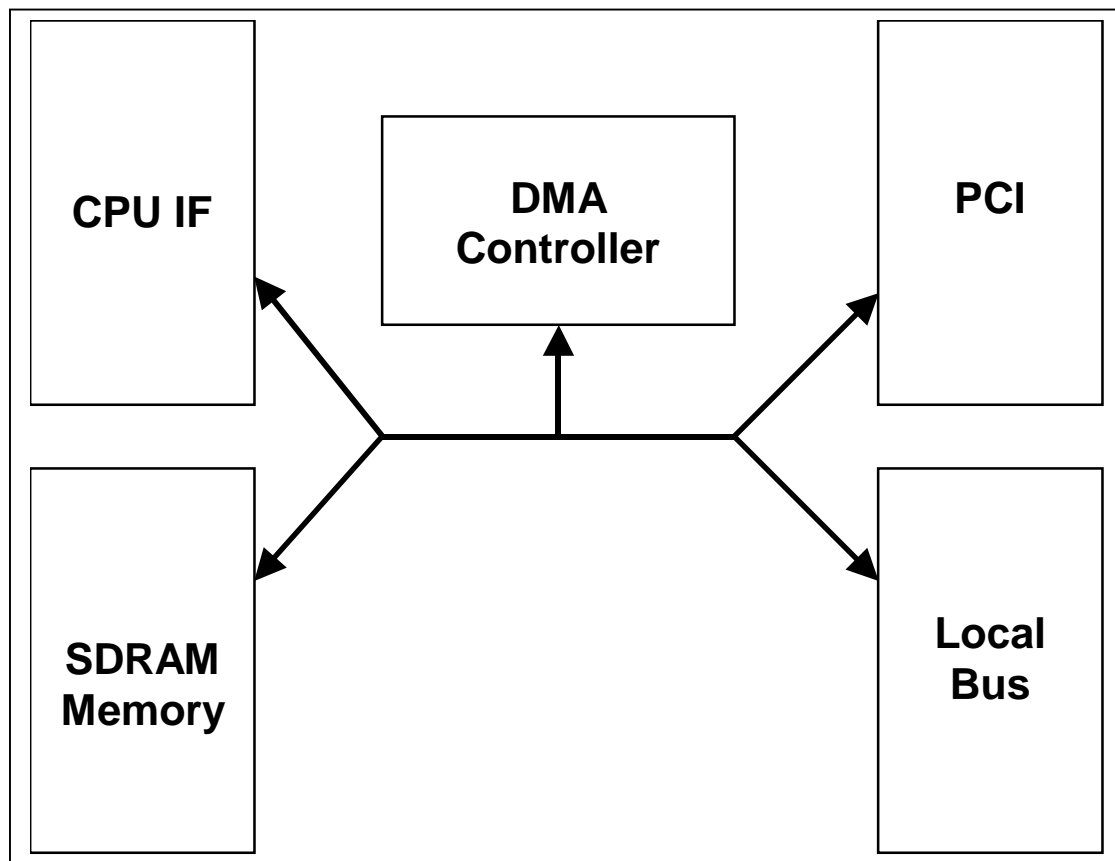
Vera adds powerful language capabilities for verification. The setup for a self-checking environment of a DMA engine requires the ability to anticipate a variety of potential transactions at the interfaces of the DUT. Since the interaction between the DUT and the environment is complex, the elements of the test environment must be able to handle the complexity. If the verification engineer has to deal with those complexities, the testing of the DUT will suffer. Instead of focusing on exercising the DUT, the engineer will focus on getting tests to work. Vera provides features that allow an environment to be built which can hide the complexity of testing from the verification engineer. This paper will show how the use of Vera features, such as re-entrant tasks and the ability to pass objects by reference, allow the verification engineer to focus on what, not how, to test.

## 1. Introduction

Before getting into the details of what was done in the test environment to verify the DMA engine, I will talk a little about the device being tested and some of the philosophies behind the testing. Since I'm talking about the testing of a DMA engine, I'll cover the interfaces of the device but will not go into detail about the internal architecture. I will go into some of the details of the DMA engine as it impacts the design of the test environment.

After the brief introduction to the problem, I will talk about the two parts of the verification effort required. First, there was a need to improve the way tests were written to stimulate the device. Second, the results must be validated or checked to make sure the results of the stimulus were what was expected. I will discuss the solution to these problems and examine the Vera code used to implement that solution. Finally, I will add some reflections and recommendations.

## 2. The Design



**Figure 1: Block Diagram of DUT**

The DMA controller is part of a CPU host bridge (Figure 1). Resources are accessed as 64-bit memory-mapped interfaces. The CPU interface is able to access resources in the other portions of the chip using base address registers. The DMA controller uses the same base address registers (BARs) as the CPU interface. Although there is a separate set of BAR for

masters on the PCI interface to access resources through the device, they don't affect the operation of the DMA controller.

As mentioned above, the controller uses the same address mapping as the CPU interface for determining the source and destination of a transfer. This mapping table, therefore, will be important to ensure the test environment and the chip use the same address mapping. The controller supports four channels with a linked list of chained commands, but only one channel can be active at one time. Each channel passes control to the next one in a round robin fashion after completing a command. Since the channel may have a linked list of chained commands, this means that the activity that is setup on a particular channel will not necessarily complete before another channel is given a chance to initiate a transfer.

Each command contains the following information: source address, destination address, transfer length, controls, and chaining address. The controls allow the use of a fixed source and/or destination address, chained commands, poll versus interrupt for command completion, command abort, and command suspension. The first command must be written into the device registers but additional chained commands are written into the SDRAM memory model at the address specified in the previous command.

### **3. Verification**

There are two parts to the verification effort. There needs to be a plan for the stimulus of the design and for the validation of the outputs from the design. I considered the stimulus and validation already in use and several possible improvements.

When I first became involved with this project, it was clear that new methodology was needed to validate the DMA engine. The controller was tested at the top level of the design. This meant that the interfaces for the testing were all the standard external interfaces. It also meant that the simulations were slow and had to run on large machines because the entire design was being simulated. The benefit was that there were commercial models for all the interfaces. Although there would have been a significant benefit in performance to test just the DMA engine separately, the effort was better spent improving the stimulus and the validation of the design.

#### **3.1. Stimulus**

Since there were already models that interacted with the design at the transaction level, the tests were written using bus functional models (BFM). To create a test, the BFM calls would be cut and pasted into the test. In order to create more than one command, there would be more cut and paste. The problem was that every test required a huge investment of time to debug. Unfortunately, structured programming was not used to abstract the creation of the stimulus.

The first step was to create task that would require only the minimal user input to create a DMA transfer. The goal was to easily write tests that could initiate multiple DMA transfers. It must automate the process of setting up the DMA in either DUT registers or off-chip memory depending on whether it was the first DMA of a chain. It must also to handle

resource sharing and pacing so the test writer would not have to be concerned about two DMA transfers being setup simultaneously on the same channel.

The `do_dma` task was created as the single task a test writer would need to create tests with any amount of complexity. It is the only task a user must call in order to perform a DMA transfer. All the setup and controls for the transfer are handled inside this task.

```
task do_dma (bit [63:0] src_add,
            bit [63:0] dst_add,
            bit [19:0] size,
            (bit [2:0] channel = 4), // 0-3 -> select chanel (when avail)
            ((integer ctl_mask = // bit 2 == 1'b1 -> sel next avail
              0)), // see notes with #define DMA_CTL_XXX
            (((integer rand_mask = // see notes with #defines DMA_RAND_XXX
              0))), // default: don't enable any rand
            (((bit [63:0] chain_base =
              64'hFFFFFF_FFFF_FFFF_FFFF))))
)
```

**Figure 2: do\_dma Task Definition**

As can be seen from the task definition, there are three required parameters to control the source and destination addresses and the size of the transfer. What was previously a long and complicated test to write now become several calls to `do_dma`. The optional arguments are described later in this section.

### 3.1.1. Simple Test Example

The example in Figure 3 shows how simple a test can be. In this example I am using all the defaults for the initialization values of base address registers. Once the initialize method of the `chip_init` class has been called, I am ready to begin making `do_dma` task calls. Since the base address registers are stored in the `chip_init` class, I can either use a constant to select the address I wish to use as a source or destination of the transfer (the PCI address of this example), or I can use the value stored in the object and add an offset (the Local Bus address of this example). The latter form is more robust as it allows tests to be copied and modified with less chances for error. A larger example is shown in Section 3.4 that exclusively uses the base addresses stored in the object when creating DMA transactions.

```

program simple_test {
  chip_init_class chip_init = new;
  integer dmatest_done_sem = alloc(SEMAPHORE, 0, 1, 0);

  {
    // Initialize the chip
    chip_init.initialize();

    // pci to localbus
    do_dma(64'h0000_0000_8001_0000,
          {chip_init.dcs4_base[63:8], 8'h0} | 64'h1000,
          'h80);

    while (semaphore_get(NO_WAIT, dmatest_done_sem, 1)) {
      Delay(10);
      semaphore_put(dmatest_done_sem, 1); // put back... expectors
    } // remove all of them
    Delay(25); // wait just a little longer

    printf("Test complete: simple_test \n");
  }
} // end of program simple_test

```

**Figure 3: Simple Test Fragment**

### 3.1.2. Addressing

This device uses a flat memory model with an address space of 64 bits. The physical device interface used as a source or destination of a transfer depends on a table of base addresses registers. Because `do_dma` needs to know what these register initializations are in order to initialize the memory at the source interface, a global class (`chip_init_class`) is used to hold all the register initialization values and perform the initialization of the device. Using this class, `do_dma` determines which interface contains the memory being initialized for a transfer. It is also used for the output checking as described in Section 3.2.4.

### 3.1.3. Initializing Source Memories

With the exception of the PCI, all interface memories were initialized using CPU writes. Although it was more of a side effect than planned feature, there was a benefit to filling the other memories using CPU writes. Since each transaction was enabled as soon as it was setup, the CPU writes for memory initialization provided background traffic for the DMA transfers in progress and uncovered a number of bugs. The Vera PCI models had some unique benefits that will be covered in more detail in Section 3.2.5.

### 3.1.4. Optional Arguments

Optional arguments aren't as nice as polymorphism. Polymorphism in languages such as C++ allow multiple definitions of the call with different types of arguments. That would have made it a little easier to create different `do_dma` calls with unrelated sets of optional arguments, but it is still a nice Vera feature. By putting the more common optional arguments toward the front of the list, most tests don't need to include all the arguments.

This means that it's easier to understand what a test is doing, even if it is using some of the optional arguments.

Channel number tells `do_dma` to either select the next available channel (value > 3) or select a specific channel (0-3). `do_dma` will use this to determine which channel to setup. `ctl_mask` allows for different behavior of the DMA engine to be tested. By default, the testbench cheats and looks at the internal status bits to determine if a DMA is complete. The control bits can tell it to use the interrupt controls on the CPU interface or to poll for the DMA completion. It can also abort or suspend (for a random delay) a DMA in progress. Since the engine can use a single address instead of a block of addresses for either the source or destination, there are controls for these as well. One of the more useful controls is chaining enable. This will use the next `do_dma` call to supply the next link in the chain, rather than setup a new DMA transaction for another channel. All controls are enabled by a bitwise or of defined constants that each have one bit set in a single bit position.

The `rand_mask` parameter allows some of the other parameters to be used as masks for randomization. A random request for `src`, `dst`, or `size` will bitwise or the random value with the mask provided. Random completion check will randomly select `probe`, `poll`, or `interrupt`. Random chain enable creates chains of random lengths. Finally, instead of doing round robin selection of the next channel for a DMA, the channel can be chosen randomly.

```
if (rand_mask & DMA_RAND_SIZE)
{
    size = 0;
    while(!size) { // keep trying until we get non-zero
size
        size = random() & size; // use size as a mask
    }
}
```

**Figure 4: Setting Random Size Code Fragment**

The implementation of random size is shown in Figure 4. This is typical of all the random masks. The only requirement made on the generated size is that it comes out non-zero. So if the size was 'h44, potential sizes are 'h04, 'h40, or 'h44.

### 3.1.5. Concurrent Execution

To properly test the DMA engine, all channels need to potentially be active simultaneously with chained commands. This means that multiple chains of commands could be setup before the first command has even begun to be transferred. Since Vera tasks are re-entrant (the variables declared inside the task are created on a stack compared to the static variables of Verilog), there was no problem of having multiple tasks running simultaneously. The `do_dma` task also takes advantage of the ability to fork a process into the background. Once initialization of the source memory has completed, the rest of the setup is done in the background. This allows the DMA controller to execute the commands in a different order than the `do_dma` calls.

Since the completion of the test cannot be determined by the completion of the last DMA command, a semaphore is used. Each call of `do_dma` adds one more semaphore and the

completion of that command removes a semaphore. The main test is able to determine that the last command has completed when there are no more semaphores. The roles semaphores played in the verification effort is detailed in Section 3.2.6.

### **3.2. Output Checking**

Output checking on the project was inadequate. When writing the test, data validation was almost completely done by visual examination of the waveforms. This is bad because it is not automated and when verification is not automated, the quality will degrade over time. The primary indication of passing or failing tests was simply whether the DMA engine indicated it was done.

The decision had been made to use a higher-level approach to stimulating the design with the `do_dma` task. Since this task centralized all the stimulus, it also made it easy to consider new approaches to how outputs would be checked. The first two approaches were considered, but the last one was the one implemented.

#### **3.2.1. Use “golden” log file for comparisons**

Since one of the problems was validating that the test resulted in the same behavior as a previous run, one approach would be to compare logs with a previous run. This requires displaying all the information about the simulation transactions to the log. The benefit is that there is not a dependency on a human observing the transactions every simulation run. It does require that the transactions are validated by hand the first time the simulation is run, but then a simple comparison of the logs will ensure that the proper behavior is observed every time.

It still requires hand validation the first time and any behavior not reflected in a display cannot be checked automatically. Another problem is that the same test can't be used with different clock speeds at interfaces. Performing comparisons fails when the timestamps or even the order of the transactions are different in the log files.

#### **3.2.2. Stimulate and compare memories at interfaces for expected results**

Another low effort improvement would save the results from each transaction into memories at each interface. At the end of the test, the values in the memories could be compared against the expected values and a pass or fail status could be displayed. This requires a little more work than simply using a golden log file, but it is not dependent on the speed of the interfaces or other implementation changes that would affect the exact timing of individual transactions, or the relative timing between different interfaces.

Since DMA data is usually moved from one interface to another, the expected data would likely be in another memory for another interface. There are some general problems with this approach, and some which are specific to this type of DMA engine. In general, there is the problem that not all data traveling over an interface will have a unique address. This can be overcome in part by planning the stimulus with non-overlapping data. Problems occur with data overruns because they can go undetected when they do not modify the final result. In this application, there is the ability to send an entire block of addresses to a single address or vice versa. When filling an entire destination block of addresses from a single address, all

the written data is observed. But if a block is written to a single location, only the last data value written is observable.

### 3.2.3. Stimulate and account for all transactions on interfaces

This is the approach chosen for checking the outputs of the design. It is best to ensure that every transaction is expected. Checking all the transactions on the fly will not just ensure that the expected data was observed at the correct interface, but will ensure there is no unexpected data slipped through unnoticed. My effort was in creating transaction checkers that looked at the data passing through the interface.

Since the interfaces adhere to standard protocols, the correct operation of that interface can be checked using a protocol monitor. Although I did not have time to ensure that every interface had a protocol checker or monitor, some of the interfaces had monitors already implemented. The memory models checked the protocol at the SDRAM interface and the PCI models had a checker as well. This relieves the data validation from looking at low-level signals.

This will catch a wide variety of errors. It catches the simple errors where the wrong data is transferred. Since it is done when the transfer occurs, the user will be informed about the problem much closer to the point of failure. Whereas errors reported at the very end of the simulation require considerable effort to find the transaction that caused the failure, this method will immediately identify the transaction causing the failure. Additional information can be displayed for the user in a DMA application that will help the debugging process that would otherwise be more difficult. For example, information about the DMA request can be provided along with the error to ease the job of determining which DMA was in progress at the time of the failure.

### 3.2.4. Output Checking and the chip\_init\_class

The chip\_init\_class was important for the stimulus to know where to preload the source data for the DMA, but even more important for checking the outputs. Since each output had output checkers (i.e. the SDRAM had an sdram\_expect task), do\_dma needed to know which interface to expect the reading of data as the source and writing of data as the destination. As can be seen from the code fragment in Figure 5, the address can easily be used to determine which device and which interface has been selected.

```
dev_select add_select = chip_init.getbar(address);
bit [63:0] dev_add;          // address expected by device

dev_add = chip_init.getdevadd(add_select, address);

case (add_select) {
  SDRAM0:
    sdram_expect(exp_data, do_write, dev_add);
}
```

**Figure 5: Expector Selection**

### 3.2.5. Vera PCI Models

Since the model was written in Vera, it was able to create dynamic slave models on the fly. I was able to use this feature to dynamically create slaves that would be used as the source or the destination of the transaction. This effectively allowed the creation of an infinite number of slave models that would dynamically appear as needed. The biggest benefit was that I didn't have to write a monitor for the PCI bus to get the data I needed. I could use the models to wait for a transaction, and let the `pci_expect` task that belonged to that transaction either provide data (on a read) or do a simple compare of the data (on a write). The models took care of the hard work of handling all the different oddities of the protocol.

### 3.2.6. Semaphores and Mailboxes

Semaphores were used extensively to synchronize the many threads that were active simultaneously. The semaphore already mentioned was the `dmatest_done_sem`. This was used as a way to keep track of the transactions that were still outstanding. Note that all the tests had the same code at the end as is seen in Figure 3. The top level of the test has to wait for all the threads to complete before it can terminate the simulation. To do that, it checks to see if there are any remaining semaphores left in the bucket. If so, it throws it back in and waits a few clock cycles before checking to see if an expector finally pulled it out on completion.

Another use was to create a simple high-level version of the round-robin arbitration done inside the DUT. By passing a semaphore between the different expectors, the one that was supposed to see an active DMA could know that it was the only one that should be active. This is important since I wanted to ensure that a DMA taking too long would produce a timeout error. A timeout error on an inactive DMA channel would be bad.

The mailbox was another useful feature used to synchronize the different threads. The `expect` process starts in parallel with the setup of the device registers. But the `expect` process needs to know information that will not be available until after the setup is complete. Since the setup process will not return immediately when it is supposed to pause or abort a transaction, this information can't be passed sequentially, so I used a mailbox to pass the information from the setup to the expector. Figure 6 and Figure 7 show fragments of the code sending and receiving the messages.

```
mailbox_put(expect_mailbox, DMA_MSG_CH);
mailbox_put(expect_mailbox, curr_channel);

// was chained
if (!chained)
    mailbox_put(expect_mailbox, DMA_MSG_CHAINED);

...

mailbox_put(expect_mailbox, DMA_MSG_GO);
```

**Figure 6: Setup Task Mailbox Sending**

```

while(!go) {
    case(mailbox_get(WAIT, expect_mailbox)) {
        DMA_MSG_GO: go = 1;
        DMA_MSG_CH: channel = mailbox_get(WAIT, expect_mailbox);
        DMA_MSG_CHAINED: lastchained = 1;
        default:
            error("ERROR: message %d from dma setup is not yet implemented!\n");
    } // case
} // while (go)

```

**Figure 7: Expector Receiving Mailbox Messages**

### 3.3. Implementation

Most of the verification effort went into the body of `do_dma`. To enable some of the stimulus and validation portions of the task, the initialization had to be rewritten to use a class to hold the initialization data. Another simple object was created inside each `do_dma` task to hold the data for use in stimulus and checking expected data.

#### 3.3.1. `chip_init_class`

This class allowed the `do_dma` task to know important configuration information about the environment. These values could have been placed into global variables (and many of them were previously stored in global variables), but maintenance of those variables was extremely painful. The declaration with tasks stubs is shown as Figure 8.

```

class chip_init_class
{
    bit [63:0] sdram_base;
    bit [63:0] pci_base;
    bit [63:0] dcs2_base, // local bus devices
              dcs3_base,
              dcs4_base,
              dcs5_base,
              dcs6_base;

    task new() {} // all the default inits
    task initialize () {}
    function dev_select getpdar (bit [63:0] add) {}
    local function bit pdar_match (bit [63:0] pdar,
                                   bit [63:0] add) {}

    function bit [63:0] getdevadd(dev_select sel,
                                  bit[63:0] add) {}
} // class chip_init

```

**Figure 8: `chip_init_class` Definition**

### 3.3.2. do\_dma Body

The flow of the DMA engine verification can be seen inside the body of the do\_dma call (Figure 9).

```
exp_data_class exp_data = new;
integer expect_mailbox;          // ctlregsetup can tell expect about
                                // dma reset or suspend actions

semaphore_put(dmatest_done_sem, 1); // we are starting a new dma test

// since we can't use chip_init to initialize last arg,
// workaround that here
if (chain_base == 64'hFFFF_FFFF_FFFF_FFFF)
    chain_base = {chip_init.sdram_base[31:21],21'h0} | 64'h3_E000;

// tell expectors & fill_mem about nosrcinc or nodstinc
if (ctl_mask & DMA_CTL_SRCINC) {
    exp_data.clr_srcinc();
}
if (ctl_mask & DMA_CTL_DSTINC) {
    exp_data.clr_dstinc();
}

// First setup source data for dma
dma_setup_src(src_add, dst_add, size, rand_mask,
              exp_data, chain_base);

// simultaneously startup the dma_expecter and reg_setup
// we will exit once we have either setup the dma or had
// the expecter fatal
expect_mailbox = alloc(MAILBOX, 0, 1);
fork
    dma_setup_ctlregs(src_add, dst_add, size, channel,
                     ctl_mask, rand_mask, chain_base,
                     expect_mailbox, chip_init);
    dma_expect(src_add, dst_add, size, ctl_mask,
              expect_mailbox, exp_data);
join any
```

**Figure 9: do\_dma Body**

#### 3.3.2.1. Local Variables

There are three important local variables used inside this task. The expected data will be setup in the exp\_data\_class instance exp\_data. For PCI this will be used to dynamically drive the source data onto the bus, but since the other interfaces have Verilog models, they are initialized using the CPU interface. All interfaces will use this class when it's passed to the dma\_expect task, which is forked off into the background when the setup of the control registers are completed.

I use a mailbox between the setup of the control registers and the expector to inform the expector of important information, such as a failure to setup the command, or that the command is chained and busy will not go away until a later command has been completed.

### **3.3.2.2. Setup Data**

The default base address for the location of chained DMA commands can't be set in the header of the task, so I use a bogus value to indicate that I should set the default value. Because I use an object for the expected data, I am able to indicate whether the expector should see incrementing addresses at the interfaces. It's worth noting that I validate the transactions at every interface. I account for both the source and destination transactions. The setup of the data patterns and the filling of the source memories is done by the `dma_setup_src` task.

### **3.3.2.3. Setup Command and Expect Data**

Normally, the command setup will complete before all the expected data has been observed. In this case, the `dma_setup_ctrlregs` will complete and leave the `dma_expect` running in the background. I take advantage of the ability to leave the validation running after the command is setup and will return the control to the test. This allows multiple commands to be setup without having to block execution at the test level. The setup task informs the expector when the DMA setup has been completed, which channel I am using, and if the command will be chained to another.

### 3.4. Larger Example

```
program larger_test{
  integer dmasize = 16;
  chip_init_class chip_init = new;
  integer dmatest_done_sem = alloc(SEMAPHORE, 0, 1, 0);
  {
    // setup array for source/dest pairs
    bit [64:0] srcadds[3];
    bit [64:0] dstadds[3];
    integer h,i,j;
    integer maxactive;
    bit [2:0] src,dst;          // src/dst indexes

    // Initialize the chip
    chip_init.initialize();

    // setup addresses... for this simple test use dcs3 & pciw0
    srcadds[0] = {chip_init.sdram_base[63:8],8'h0}|64'h300;
    srcadds[1] = {chip_init.pci_base[63:8],8'h0}|64'h40;
    srcadds[2] = {chip_init.dcs3_base[63:8],8'h0}|64'h1000;

    dstadds[0] = {chip_init.sdram_base[63:8],8'h0}|64'h0;
    dstadds[1] = {chip_init.pci_base[63:8],8'h0}|64'ha00;
    dstadds[2] = {chip_init.dcs3_base[63:8],8'h0}|64'h10;

    // do different src/dest pairs on every chain
    h = 0;
    for (src = 0; src < 3; src++) {
      for (dst = 0; dst < 3; dst++) {

        maxactive = random()%3+2; // have variable number of channels active
                                // simultaneously from 2-4
        for (i = h; i < maxactive+h; i++) { // do var number of channels
          for (j = random()%3+1; j >= 0; j--) { // randomly choose length of chain
            if (j) // should chain this one
              do_dma(srcadds[src], dstadds[dst], dmasize,i%4,DMA_CTL_CHAIN);
            else // end of chain
              do_dma(srcadds[src], dstadds[dst], dmasize);
            srcadds[src] += dmasize;
            dstadds[dst] += dmasize;
            dmasize += 8;
            if (dmasize > 128)
              dmasize = 16;
          } // for j
        } // for i
        h++; // start next batch at next channel
      } // for dst
    } // for src

    while (semaphore_get(NO_WAIT, dmatest_done_sem, 1)) {
      Delay(10);
      semaphore_put(dmatest_done_sem, 1); // put it back... expect should
    } // remove all of them
    printf("Test complete: Indus_small_chain_morech \n");
  }
} // end of program larger_test
```

Figure 10: Larger Example Test

## 4. Conclusions and Recommendations

The easy part was improving the stimulus. It wasn't too hard to handle the different requirements for setting up DMA commands in the chip or in memory. What proved much harder was creating the dynamic checkers. The only checker that was easy was the PCI since it already had tasks that could be used to look for transactions and get transaction data. All the other interfaces needed transaction snoopers created that could extract out the transaction information for doing the expected data checking. This proved to be the most time consuming to get right.

This implementation was done under tight time constraints (what project isn't). As a result, there were a number of alternate implementations that were not explored. If `do_dma` had been implemented as a class instead of a task, there would have been a little more flexibility with parameters and configurations. Because this would be harder for many engineers to maintain, care would have to be taken to make sure it was well documented. The usage would have to be thought through so that the flexibility wasn't paid for with excessive verbosity. This might be the best long-term solution since it would allow for additional functionality without breaking any of the existing tests.

This is a large enough block of a design to benefit from module level testing. If there were an effort to test many of the large blocks at the module level, there would have been models for the internal shared interfaces that would have made creating an environment for this block much easier. The resulting testing could have exercised many more odd corner cases on this block with less simulation effort.

Vera's optional task arguments are a great improvement over Verilog, but since there is still a restriction of a single task with a particular name, overloading can't be done. In some of the common tests, excess arguments had to be passed just to fill positions to get to the one I wanted.

It would have improved performance to use backdoor access to the memories in the local bus slaves and the SDRAM. It's an enhancement that could be considered since it would cut down on some potentially wasted bandwidth. But since this bug has been exploited as a source of background traffic, the environment would need to be enhanced to support background traffic generation as well if this were removed.