DMA Controller Lab

January 18, 2012

This lab will lead you through the basic aspects of the BSV (Bluespec SystemVerilog) language and usage of the Bluespec Development Workstation for Bluesim and Verlog simulations. In each part you will find descriptions of the actual code to write along with instructions on how to use and invoke each of these tools with the Development Workstation.

The lab is divided into multiple directories, each containing lab and solution `.bsv` files. For each lab, you'll be modifiying the file `DMA.bsv`. Copy `DMA.bsv.lab` to `DMA.bsv` before you start each section to ensure you are working from a clean copy.

Note: *The directory `$BLUESPECDIR/doc/BSV/` contains useful documentation, including the language Reference Guide, the tool User Guide, a KPNS (Known Problems and Solutions) guide, and others.*

The directory `$BLUESPECDIR/training/BSV/examples/` contains some useful examples for future reference. The full set of training materials, documentation and examples can be accessed from the file `$BLUESPECDIR/index.html`.

1 Exercise 1: BSV Basics and Writing Rules

1.1 Objectives

- Define basic BSV objects including Regs, FIFOs, and enumerated types
- Learn how to express module behavior with rules
- Gain experience working with the Bluespec Development Workstation (BDW) to compile, link, and analyze a design
- Become more familiar with Bluespec tools

1.2 Setup

The one-channel DMA controller is shown in Figure 1. It is connected to a testbench and a target memory, `mem`, which acts like a memory for testing purposes. The memory target has a minimum 2 cycle latency.

The directory for this lab contains the following `.bsv` files:

- `DMA.bsv`: the DUT, a simple one-channel DMA controller.
- `Tb.bsv`: a testbench module which instantiates the DUT alongside a BRAM memory. The testbench configures the DUT to move 10 words of data from address 0 to address 100. It then reads 10 words of data at address 100 to check for the expected results.
- `Socket_IFC.bsv`: a file defining a generic socket interface in which requests and responses are decoupled.
- `Targets.bsv`: a memory module with a socket slave interface, used by the testbench.

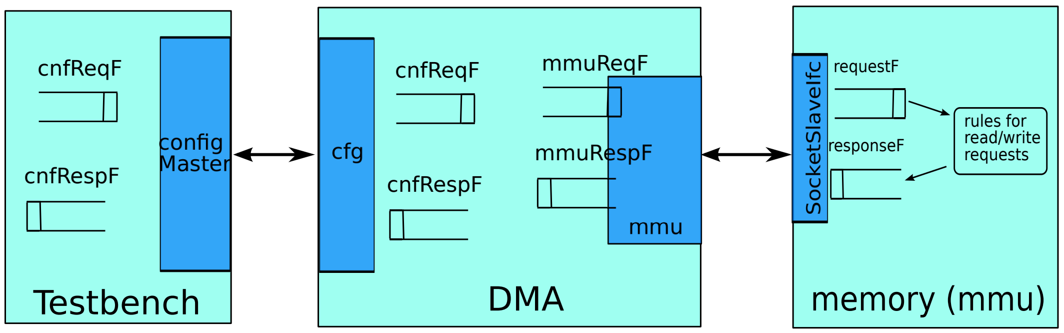


Figure 1: Overview of DMA

The design uses a simple socket interface, shown in Figure 2, in which requests and responses are decoupled. The DMA provides a slave socket interface to the testbench and a master socket interface to the memory. This is a pipelined design; several requests may be outstanding.

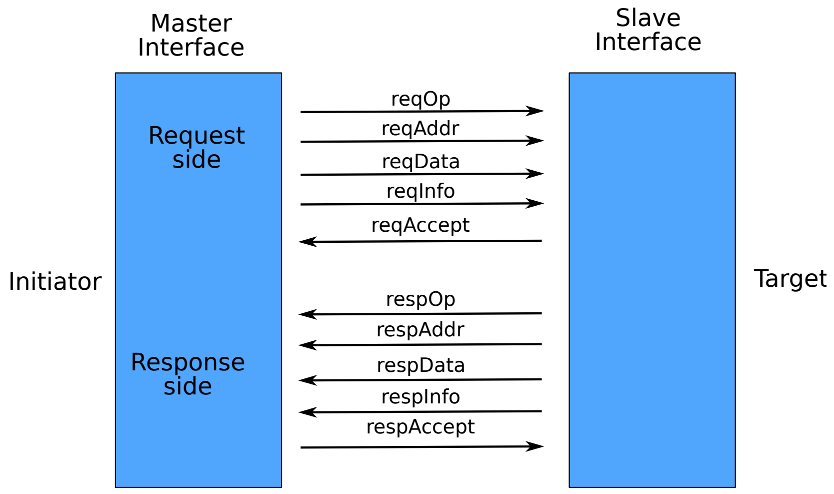


Figure 2: Socket Interface

1.3 Rules, registers, interfaces, and FIFOs

Figure 3 illustrates the main rules, FIFOs and registers in the DMA block (`DMA.bsv`). To simplify the diagram only a subset of the rules are displayed.

The DMA contains the following registers, interfaces, and FIFOs:

- There is a single `Socket_slave_ifc` named `cfg` communicating with the testbench.
- There is a single `Socket_master_ifc` interface, `mmu`, representing the memory.
- Each socket has two FIFOs, one for the request and one for the response.
- The destination address is stored in a register named `destAddrR`
- A register (`responseDataR`) is used to pass the read response to the write side, allowing pending transactions and concurrency.
- A register (`destAddrR`) is used to pass the destination address for each read over to the write side.

1.4 Rules

- `startRead`: When the DMA is enabled and there is data to move, the conditions for this rule are met. The rule enqueues the write destination address in the register `destAddrR`.
- `finishRead`: When the correct response data is on the mmu response FIFO, the read is finished. The data is passed to the write side of the DMA through the `responseDataR` register.
- `startWrite`: This rule conflicts with the `startRead` rule, so the `descending_urgency` attribute is used to finish writing before the next read.

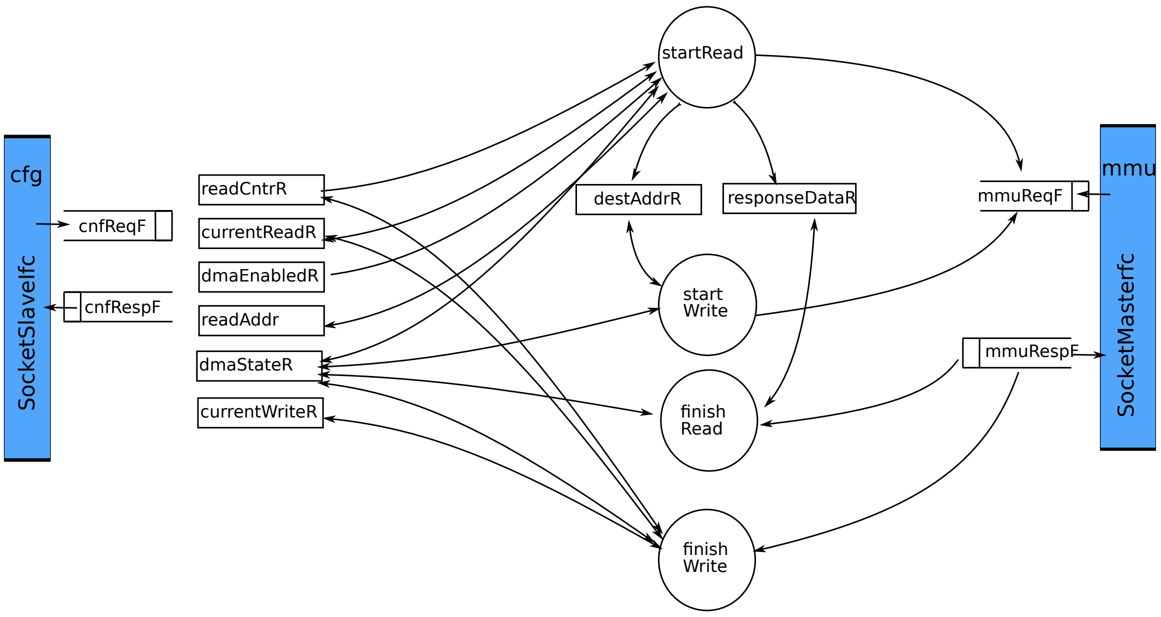


Figure 3: Overview of DMA example

- `finishWrite`: The rule waits for the write to finish.
- `readConfig` and `writeConfig` (not shown): These rules use the function `selectReg` to map from address to specific registers to read to and write from the `cfg` interface.

1.5 Complete DMA

Note: Copy the file `Part1/DMA.bsv.lab` to the file `DMA.bsv` to ensure you're working from a clean copy.

1. Open the project with the workstation:

   ```
   % bluespec DMA.bspec &
   ```

   The `.bspec` file is the project file for the design, defining the options and parameters for the BDW, including compile and link flags.

   The BDW Main Window and Project Files window will open.

2. To review the options and create the working directories, open the **Options** window from the **Project** menu. Press **OK** and the BDW will create the working directories specified on the **Files** tab.

3. In the window titled **Project Files**, double click on `DMA.bsv` to open the file. The default editor can be changed on the editor tab in the **Project Options** window. Most of the file is complete. In the next steps you will add the missing pieces.

4. Define an enumerated data type named `DMA_State` to hold the state of the DMA. The states are: `Idle, ReadFinish, Write, WriteFinish`. Don't forget the provisos `(Bits, Eq)` on the typedef statement.
5. Using the config port FIFOs as an example, instantiate a FIFO for the master request (`mmuReqF`) and response (`mmuRespF`). They should be guarded, sized FIFOFs. The master request FIFO has the same arguments as the slave response FIFO, the master response FIFO has the same arguments as the slave request FIFO.
6. Instantiate a register named `dmaStateR` with an initial value of `Idle` to hold the value of the state register.
7. Define the rule `startRead`
   - The explicit rule conditions are:
     - The register named `dmaEnabledR` is `True`
     - `readCntrR` is greater than `currentReadR`
     - The `dmaStateR` is `Idle`
   - Construct a read request named `req` of the type `Socket_Req` with the following struct element values:
     - `reqAddr` is set to the value of `readAddrR`
     - `reqData` is 0
     - `reqOp` is `RD`
     - `reqInfo` is 1
   - Enqueue the value `req` into the master request FIFO (`mmuReqF`)
   - Increment the read address (`readAddrR`) and the counter (`currentReadR`)
   - Set the DMA state (`dmaStateR`) to `ReadFinish`
8. Define the rule `finishRead`
   - The rule should fire when the state is `ReadFinish`
   - Set the value of `resp` (of type `Socket_Resp`) to the first value in the master response FIFO (`mmuRespF`)
   - Dequeue `mmuRespF`
   - Save the response data in `responseDataR`
   - Set the state to `Write`
9. Save the file.

1.6 Build and Review

1. **Compile**: On the main workstation window, press **Compile** from either the **Build** menu or using the icon on the toobar - this will generate Bluesim objects from the source
   - Compile errors will be displayed in the message area in red. Double click on the message and it will open the file and position the cursor in the line with the error.

    - Correct the error
    - Save and recompile. Continue until you have no more errors.

2. **Link**: Press **Link** - it will link the above objects to an executable named `out`, as specified on the **Link Simulate** tab of the **Project Options** form.
3. **Simulate**: Observe the results. Do you see what you expected?
4. **Analyze**: The workstation provides a set of windows and browsers, along with links to third-party wavefrom viewers to analyze and debug your design.

   View waveforms:

   - Open the Module Browser
   - **Wave Viewer→ Start**
   - **Wave Viewer→ Load Dump File**
   - **Module→ Load Top Module**
   - Expand `dma DMA1`
   - Check the `WILL_FIRE` signals to see which rules are firing by selecting a rule and the **Send Will Fire** button. Rules are displayed in blue in the module browser list.
   - You can also view the rule conditions through the **Send Predicate** and **Send Body** buttons.

   Observations:

   - The rules `startRead`, `finishRead`, `startWrite`, and `finishWrite` can never fire in the same cycle since the state machine can only be in a single state at a time.
   - The latency from the DMA to the memory target is at least 2 cycles.
   - There really isn't any way to speed up the DMA in this example.

2 Exercise 2: Using Implicit Conditions

In Exercise 1 you developed a simple state machine to control a single-channel DMA using the register `dmaStateR` to determine the state of the DMA. In this exercise you will modify the DMA to use the implicit conditions of FIFOs to help determine which rules should fire, instead of the `dmaStateR` register.

2.1 Objectives

- Understand the power of implicit conditions and dynamic semantics, especially as they relate to rules
- Understand the incremental changes required to scale a design
- Continue to gain experience working with the Development Workstation to build and analyze BSV designs

The directory `Part2` contains the same `.bsv` files as those used in the previous exercise.

2.2 Implicit Conditions

Whether a rule can fire is determined by both its explicit and implicit conditions. Calls to methods from within the rule (or the rule's condition) add the method's implicit conditions to the rule's explicit conditions. The implicit conditions of the interface methods become part of the conditions of the rule.

For example, let's examine the rule `finishRead` in our DMA design from the previous exercise.

```
rule finishRead ( dmaStateR == ReadFinish ) ;
   Socket_Resp resp = mmuRespF.first ;
   mmuRespF.deq ;

   responseDataR <= resp.respData ;

   dmaStateR <= Write ;
endrule
```

The explicit condition states that the rule fires when the state variable is equal to `ReadFinish`. An implicit condition of the rule is that the FIFO `mmuRespF` must not be empty, since you cannot read the value of an empty FIFO.

In this exercise, we are going to replace the registers with FIFOs, and take advantage of the implicit conditions of the FIFOs to determine which rules can fire, instead of using the state variable.

2.3 Modify DMA example

Note: Copy the file `Part2/DMA.bsv.lab` to the file `Part2/DMA.bsv` to ensure you're working from a clean copy.

1. Remove definition and all references to `dmaStateR`.

2. Replace the registers storing the destination address and the response data with FIFOs.

   - The FIFO `responseDataF` is an internal FIFO used to pass the read response to the write side. It has a data type of `ReqData`. It is a `mkSizedFIFO(2)`.
   - The FIFO `destAddrF` is used to enqueue the destination address of the next data transfer. The depth of the FIFO limits the number of outstanding reads which may be pending before a write. It has a data type of `ReqAddr` and is a `mkSizedFIFO(4)`.

3. Modify the rule `startRead`

   - Modify the explicit condition by removing the state condition variable (`dmaStateR`). The remaining conditions do not change.

    - Enqueue the destination address in the FIFO `destAddrF`.

4. Modify the rule `finishRead`
    - Modify the explicit conditions for the rule `finishRead`.

      The rule should fire when there is a response on the mmu and it is the correct response (hint: `respInfo==1`).
    - Save the data in the FIFO named `responseDataF`
    - Remove any unneeded statements
5. Modify the rule `startWrite`
    - Modify the explicit condition. The write can occur when there is a write address in the `destAddrF` FIFO. (hint: the rule condition includes both the explicit and implicit conditions from the methods).
    - Generate the write request using the FIFOs
    - Dequeue the FIFOs
    - Modify the housekeeping statements
6. Add a rule to indicate when the transfer is done.
    - Name the rule `markTransferDone`
    - The explicit conditions are:
        - the dma is enabled;
        - all the values read have finished the read (`currentReadR == readCntrR`)
        - all the values read have been written
    - set the `dmaEnabledR` register to False
    - set the values of read and write counters to 0

2.4 Build and Review

1. Build (Compile, Link, Simulate). This can be done in a single step from the toolbar.

   Note: If when you compile, you get a message "Directory does not exist", the working directories have not been created. Open the Project Options window and select OK and the workstation will create the specified directories for you.
2. Analyze using the browsers in the workstation and the waveform viewer.

Observations

- Multiple states of the DMA can be simultaneously active, improving throughput from the previous example.
- Change the size of the destination address FIFO (`destAddrF`) and see how it changes the waveform.

3 Exercise 3: Multi-channel DMA Controller

In this example you will add an additional channel to the single-channel DMA controller, allowing the DMA to process two transfers simultaneously. The new design uses the type `NumChannels` to represent the number of channels.

The design has two significant modifications from the single-channel DMA:

- The registers for the configuration channnels have been replaced with vectors of registers to hold values for multiple channels.
- The rules are defined within a function, allowing the design to generate rules for both channels without rule duplication.

3.1 Objectives

- Learn how to use the `Vector` package
- Experience the power of static elaboration
- Learn how to use the `fromInteger` function
- Learn how to express module behavior with rules
- Continue to gain experience working with the Development Workstation to compile, link, and analyze a design

3.2 The `Vector` Package

In this example we add vectors to handle multiple ports without duplicating the rules for each port.

The `Vector` type, provided in the package of the same name, is a synthesizable type containing a known number of elements of one type. To use vectors, you must import the `Vector` package, which includes many functions which create and operate on vectors.

The vector type is written:

```
    Vector#(number of items, type of items)
```

A vector of 4 2-bit registers would be written:

```
    Vector#(4, Reg#(2))
```

The length of a vector (number of elements) must be resolvable during type-checking.

Part of the power of vectors comes from the large set of functions defined in the `Vector` package. One such function is the `genVector` function. It generates a vector where the value of the element is the index number.

Example:

```
Vector#(3, Integer) a_vector = genVector();
//a_vector = {0,1,2}
```

The `map` function allows you to apply a function to all elements of a vector.

```
Vector#(4, Int#(16)) b_vector = map (fromInteger, genVector);
```

The `map` function above applies the `fromInteger` function to each element generated by the `genVector` function. In other words, `genVector` creates a vector of 4 elements (0,1,2,3), each of type `Integer`. The `map` function then applies the `fromInteger` function to each element, returning a vector of 4 elements, each of type `Int#(16)`, defining and initializing the declared vector variable `b_vector`.

The `mapM` function is similar to the `map` function, but is used with certain types of functions, such as module instantiations. The `replicateM` function is a map-like function that generates a vector of elements by using the given function, such as a module instantiation. In this example we'll be using the `replicateM` function to instantiate vectors of registers.

```
Vector#(NumChannels, Reg#(ReqAddr)) readAddrRs <- replicateM(mkReg(0));
```

3.3 Function of rules

There are a few different ways we could modify the rules in the DMA to handle an additional channel. The simplest would be to duplicate each rule for each channel. While straightforward, the duplication is by no means the most elegant solution and is difficult to extend to more channels. A better solution is to define a function that will generate the rules for each channel. These rules are based on the rules from the previous exercises.

We define a function inside the module so it can access some of the registers without passing too many arguments. The function takes as arguments the FIFO interfaces and the channel and returns a set a rules. The rules within the function are identical to the set used in the one mmu port case.

The function header is:

```
   function Rules generatePortDMARules (FIFOF#(Socket_Req)  requestF,
                                        FIFOF#(Socket_Resp) responseF,
                                        Integer chanNum);
```

The rules are generated for the channel when the function is called and passed the arguments defining the port channel.

Example:

```
   ruleset = rJoinDescendingUrgency (ruleset,
                  generatePortDMARules( mmu1ReqF, mmu1RespF, 0));
   ruleset = rJoinDescendingUrgency (ruleset,
                  generatePortDMARules( mmu2ReqF, mmu2RespF, 1));
   for (Integer ch = 0; ch < valueof(NumChannels) ; ch = ch + 1)
      ruleset = rJoinDescendingUrgency (ruleset,
                    generateTransferDoneRules(ch));
```

In this exercise the function and the processing of the rules is defined for you. You'll be modifying the rules within the function to use vectors, so that a single rule can process different channels depending on the vector values.

3.4 Modify DMA instantiations

Note: Copy the file `Part3/DMA.bsv.lab` to the file `Part3/DMA.bsv` to ensure you're working from a clean copy.

The new channel is an additional mmu port which could represent a mmu with separate read and write ports, different memories, or a separate bus for peripherals. The interface provided by the DMA has been modified to have two mmu ports, `mmu1` and `mmu2`.

```
interface DMA2 ;
   interface Socket_slave_ifc    cfg ;
   interface Socket_master_ifc   mmu1 ;
   interface Socket_master_ifc   mmu2 ;
endinterface
```

The naming convention in this exercise adds an `s` for the vector version of the registers and FIFOs. For example, the `readCntrR` register in the single-channel DMA becomes the `readCntrRs` vector of registers in the multi-channel DMA.

In the file we've defined the number of channels using the type `NumChannels`, which you can use to define the number of elements in each vector.

```
typedef 2 NumChannels;
```

1. Import the `Vector` package
2. The FIFO definitions for mmu1 are already in the design. Add the definitions for mmu2.
3. Use the `replicateM` function to instantiate a vector of registers for the following, replacing the register instantiations from the single-channel DMA:
   - `readAddrRs`
   - `readCntrRs`

- currentReadRs
- currentWriteRs
- portSrcDestRs (this is a 2 bit register)
- destAddrRs

4. Use the replicateM function to instantiate vectors of FIFOs for:
   - responseDataFs (mkSizedFIFO(2))
   - destAddrFs (mkSizedFIFO(4))

3.5 DMA rules

In this section you are going to modify the main read and write rules from the single-channel DMA version (Exercise 2) to handle multiple channels.

A few things to notice about the multi-channel DMA file you are working on:

- The variable chanNum is used as the index for Vectors. It refers to the current channel.
- The state element names are the names used in Exercise 2. When modifying the rules you'll need to change both the name of the register (or FIFO) as well as adding the index for Vector elements.
- In many of the rules Exercise 2 refers directly to the mmu FIFOs (mmuReqF and mmuResponseF). In this example, the rules use the FIFOs in the function argument (requestF and responseF) instead.

1. Rule startRead
   - Modify the explicit condition to include the index for the vectors of registers.
   - Define the request, (let req=), including the index chanNum where necessary. Note that the value for reqInfo is now fromInteger(0 + 2*chanNum);.
   - Modify the housekeeping statmenets to add the vector indices as necessary.
2. Rule finishRead
   - The explicit condition has been modified for you already.
   - Modify the resp statement to use the correct response FIFO. Don't forget the dequeue statement.
   - Add the vector indices when you pass the data to the write side of the DMA (responseDataFs).
3. Rule startWrite
   - Modify the write request. The reqInfo is now fromInteger(1 + 2*chanNum).
   - Enqueue the request into the requestF FIFO.
   - Modify the housekeeping statements.

4. Rule `finishWrite`

    - The explicit condition is based on the response FIFO. The value for `reqInfo` is now `fromInteger(1 + 2*chanNum)`.
    - Take the response from the correct FIFO.
    - Modify the increment counter statement.

3.6 Build and Review

1. Build (Compile, Link, Simulate). This can be done in a single step from the toolbar.

2. Analyze using the browsers in the workstation and the waveform viewer.

4 Exercise 4: Using BSV's TLM interface

A common paradigm between blocks uses a get/put interface: one side *gets* or retrieves items from the interface and the other side *puts* or gives items to the interface. These types of interfaces are used in *Transaction Level Modeling* or TLM for short. This pattern is so common in system design that BSV provides parameterized library interfaces, modules and functions for this purpose.

In this example, shown in Figure 4, we replace the socket interfaces with TLM interfaces and connect the DMA into an AXI bus fabric using the BSV AXI library. The library provides transactors to implement an AXI bus, connecting TLM interfaces on one side with AXI interfaces on the other side.

4.1 Objectives

This exercise is primarily a walk-through demonstrating how to:

- Build a reusable IP (in this case a multi-channel DMA) using BSV's generic TLM interface facilities
- Use BSV's libraries to build generic memories
- Use BSV's libraries to attach generic TLM interfaces to the memories
- Specialize the interfaces to a particular bus interface by using BSV's library of transactors (in this case, converting TLM to AXI)
- Use BSV's libraries to build an AXI interconnect fabric
- Build a small testbench to exercise the overall system.

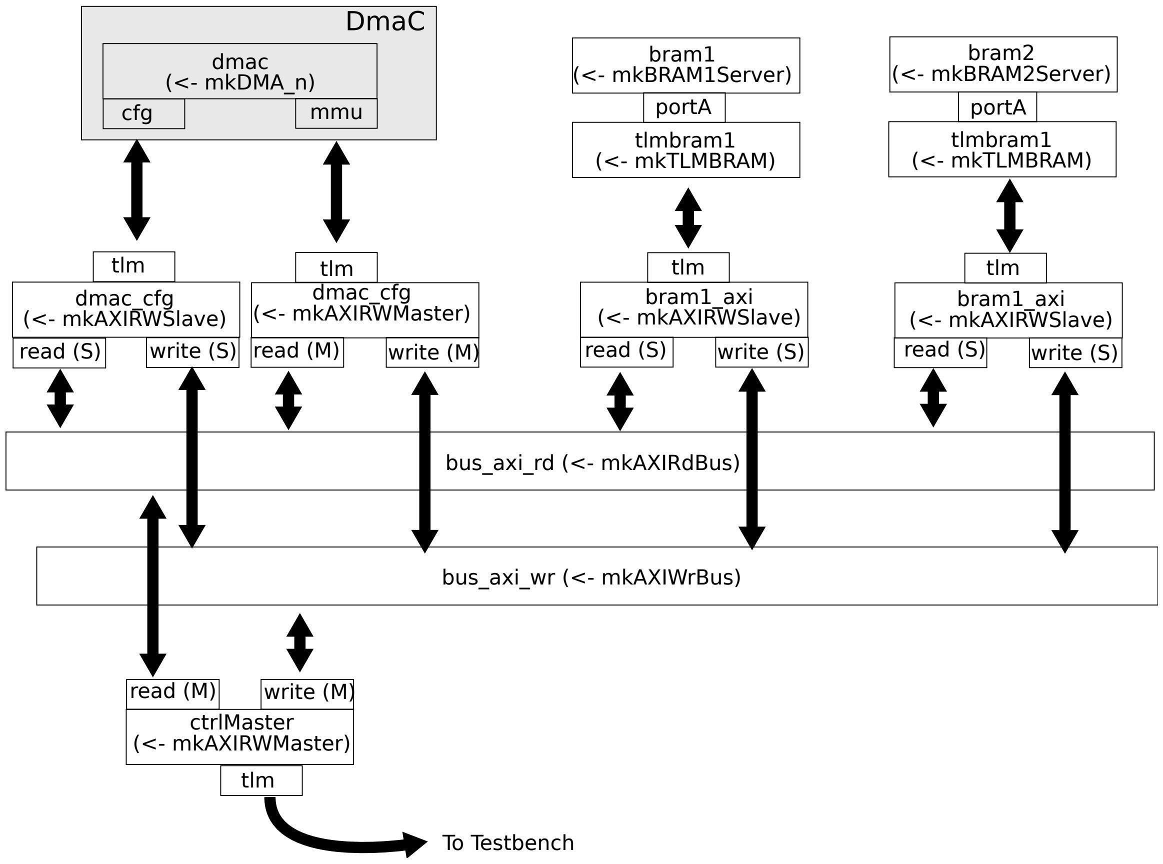


Figure 4: Overview of DMA example using TLM and an AXI bus fabric

4.2 Setup

The example directory Part4 contains all the files necessary to run the system, explore the code, and try variations.

- `DMA.bsv`: The DMA we've been working on, modified to use TLM interfaces in place of the socket interfaces used in the previous examples.
- `GDefines.bsv`: Global type definitions for the DMA.
- `SocAXI.bsv`: A system on a chip (SoC) subsystem, consisting of a DMA engine, two memories, an AXI interconnect fabric, and a master port into the AXI to configure the DMA, access memory, etc.
- `Top.bsv`: A simple top-level testbench that initiates two concurrent DMA transfers and waits for them to complete.
- `DMA.bspec`: The project file for the Bluespec Development Workstation (BDW).
- Axi library: As part of the AzureIP Premium offerings, Bluespec has available packages to support development of bus-based desings implementing AXI and AHB protocols. A version of the Axi package is provided here for use in this lab. The Axi

library groups the AXI data and protocols into reusable, parameterized interfaces which interact with TLM interfaces.

4.3 TLM Interfaces

The TLM package facilitates creating bus-based designs independent of any specific bus protocol. The TLM package includes two basic interfaces: `TLMSendIFC` and `TLMRecvIFC`. These interfaces use basic `Get` and `Put` subinterfaces as the requests and responses. The `TLMSendIfc` generates requests and receives responses. The `TLMRecvIFC` receives requests and generates responses.

4.4 DMA

The primary change to the DMA is the change in interface provided by the DMA. Internally, the DMA still uses the socket ports and then connects the interface FIFOs to the socket ports.

In the previous examples, we implemented a DMA that provided Socket-based `DMA` interface:

```
interface DMA ;
   interface Socket_slave_ifc    cfg ;
   interface Socket_master_ifc   mmu1 ;
   interface Socket_master_ifc   mmu2 ;
endinterface
```

In this example, we replace the `DMA` interface with the TLM-based `DmaC` interface:

```
interface DmaC #(numeric type numChannels);
   interface TLMRecvIFC#(‘ARM_RR) cfg;
   interface TLMSendIFC#(‘ARM_RR) mmu;
endinterface
```

The number of memories supported by the DMA is determined by the `numChannels` parameter, determined when the DMA is instantiated. By making the number of channels polymorphic, the number of channels controlled by the DMA can be changed without changing the DMA design.

Because the interface provided by the DMA has changed, the interface definitions, describing how the data is handled by the interface, has also changed.

```
interface TLMRecvIFC cfg;
   interface Get tx;
      method ActionValue#(BusResponse) get;
         cnfRespF.deq();
         return socketRespToBusResp(cnfRespF.first());
      endmethod
   endinterface
   interface Put rx;
      method Action put(x) = cnfReqF.enq(busReqToSocketReq(x));
   endinterface
endinterface

interface TLMSendIFC mmu;
   interface Get tx;
      method ActionValue#(BusRequest) get;
         mmuReqF.deq();
         return socketReqToBusReq(mmuReqF.first());
      endmethod
   endinterface
   interface Put rx;
      method Action put(x) = mmuRespF.enq(busRespToSocketResp(x));
   endinterface
endinterface
```

Functions are included to translate the socket ports into TLM interfaces.

```
function BusResponse socketRespToBusResp(Socket_Resp tmp);
   BusResponse r = defaultValue;
   r.error = !(tmp.respOp==OK);
   r.data = tmp.respData;
   r.write = (tmp.reqOp==WR);
   r.id = unpack(truncate(tmp.respInfo));
   return r;
endfunction

function BusRequest socketReqToBusReq(Socket_Req tmp);
   BusRequest r = defaultValue;
   r.byteen = '1;
   r.address = tmp.reqAddr;
   r.data = tmp.reqData;
   r.write = (tmp.reqOp == WR);
   r.id = unpack(truncate(tmp.reqInfo));
   return r;
endfunction

function Socket_Req busReqToSocketReq(BusRequest x);
    Socket_Req r = unpack(0);
    r.reqOp = x.write ? WR : RD;
    r.reqAddr = x.address;
    r.reqData = x.data;
    r.reqInfo = pack(extend(x.id));
    return r;
 endfunction

 function Socket_Resp busRespToSocketResp(BusResponse x);
    Socket_Resp r = unpack(0);
    r.respOp = x.error ? NOP : OK;
    r.respData = x.data;
    r.reqOp = x.write ? WR : RD;
    r.respInfo = pack(extend(x.id));
    return r;
 endfunction
```

4.5 SocAxi

The file `SocAxi` contains the SoC subsystem consisting of:

- Instantiation of the DMA engine with 2 channels

```
// DMA of given size
module mkDMA_n ( DmaC #(2) );
   let ifc <- mkDMA;
   return ifc;
endmodule
```

```
   DmaC#(2) dmac <- mkDMA_n;
```

- Instantiation and initialization of 2 BRAM memories (`bram1` and `bram2`) only 1 of which is shown here

```
   BRAM1Port#(Bit#(16), Bit#(32)) bram1 <-mkBRAM1Server(bcfg);
   TLMRecvIFC#(‘ARM_RR) tlmbram1 <-mkTLMBRAM(bram1.portA);
   AddressRange#(AxiAddr#(‘ARM_PRM))   bram1_params = defaultValue;
   bram1_params.base       = 32’h0010_0000;
   bram1_params.high       = 32’h0020_0000 - 1;
```

- Instantiation of a Master port into the AXI

```
   AxiRWMasterXActor#(‘ARM_RR,‘ARM_PRM)  ctrlMaster <- mkAxiRWMaster;
```

- Definition of the AXI bus fabric

4.6 Build and Review

1. Open the file `Part4/DMA.bspec` in the BDW.
2. Build (Compile, Link, Simulate). This can be done in a single step from the toolbar.
3. Analyze using the browsers in the workstation and the waveform viewer.

5 Exercise 5: Implementing the DMA with emVM

The Bluespec emVM (virtual emulator) environment allows modeling at multiple levels of abstraction and supports verification through both simulation and emulation. The DMA used in the AXI bus fabric can be easily emulated on an FPGA using emVM.

An emVM system, shown in Figure 5, has two major components: an FPGA containing a Device under Test (DUT) and a PC host containing an emulation console. The two sides communicate over a physical PCIe link connecting with the FPGA and emulation console through implementation-independent transactors. The environment also supports simulating the design on the FPGA side with a software testbench on the host side using the same transactors and components.

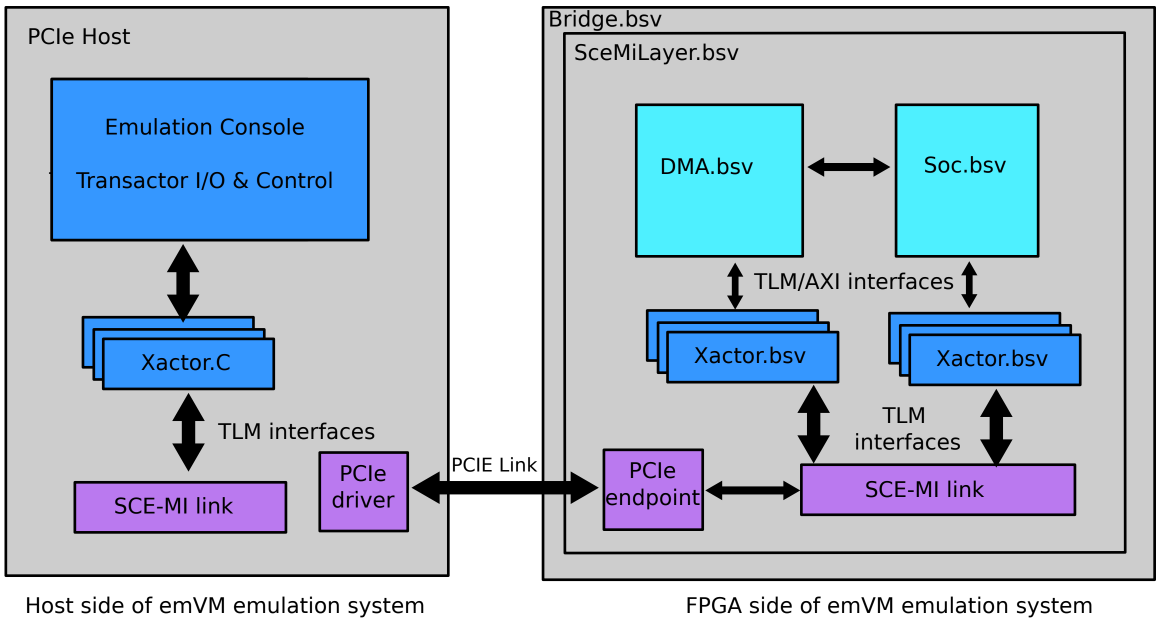


Figure 5: Overview of DMA within the emVM emulation system

5.1 Objectives

This exercise is a walk-through demonstrating:

- The components of an emVM system
- How to modify an existing design to use as the DUT in emVM
- Building and simulating the DUT
- Running the simulation from a GUI

5.2 Setup

The example directory Part 5 contains all the files necessary to run the system, explore the code, and try variations.

As can be seen from Figure 5, the components of the system can be divided into the hardware side (emulation on the FPGA or simulation) and the host side.

The following BSV files are on the hardware side:

- `DMA.bsv`: The DMA DUT does not change at all from the previous example.
- `Soc.bsv`: The SoC consisting of DMA and BRAM, connected to an AXI bus. The SoC provides the `Soc` interface, which is a TLM interface.
- `SceMiLayer.bsv`: This emVM connection layer links the platform-independent DUT (DMA and SoC) with the FPGA-dependent bridge, inserting the SoC into the emVM environment.

- `Bridge.bsv`: An FPGA-specific template file, delivered as part of the emVM system. This file is provided by Bluespec and is not modified by the user.
- `Tb.bsv`: A testbench module for simulation of the hardware side without using a host testbench.

The host side contains an untimed software testbench along with software transactors to communicate with the hardware transactors. Bluespec also provides a set of Tcl/Tk components to add a front-end GUI to control the testbench.

This exercise includes the following files for the host side:

- `DMATester.cxx`, `DMATester.h`: Application-specific software transactor to connect with the hardware side.
- `TclTb.cxx`: Host side testbench.
- Tcl files to build and run the Gui. For more detailed information on building a host testbench and GUI, refer to the *Bluespec emVM User Manual.*
  - `gui_dut.tcl`
  - `gui_top.tcl`
  - `pkgIndex.tcl`
  - `gui`

The example contains one additional file, the `project.bld` file, which is used when building the project.

5.3 Hardware-side Components

5.3.1 DMA

The DMA does not change at all from the previous example. It is instantiated within the SoC subsystem implemented in `Soc.bsv` providing the `Soc` interface:

```
   typedef  TLMRecvIFC#(‘AXI_RR) Soc;
```

5.3.2 Message ports

The hardware side of the message channel is defined as a message port. The hardware transactors use these ports to access messages being sent to and received from the testbench. In an emVM system, each message port from the hardware side is paired with a port proxy on the software-side testbench. Bluespec provides port proxies for both BSV and C++ testbenches.

5.3.3 SceMiLayer.bsv

The file `SceMiLayer.bsv` connects the requests and responses from the SoC into transactors that communicate with the testbench. The testbench may be a C++ testbench on the host that communicates with the hardware side over PCie, or may be a BSV testbench on the hardware platform.

The file `SceMiLayer.bsv` in this example includes the following actions:

- Define the uncontrolled Clock and Reset

```
Clock                     uclk      <- sceMiGetUClock;
Reset                     urst      <- sceMiGetUReset;
```

- Define the controlled clock and reset for the DUT

```
SceMiClockConfiguration  clk_cfg   = defaultValue;
clk_cfg.clockNum        = 0;
clk_cfg.resetCycles     = 4;
SceMiClockPortIfc        clk_port  <- mkSceMiClockPort( clk_cfg );
let cclock = clk_port.cclock;
let creset = clk_port.creset;
```

- Instantiate the SoC

```
Soc                       dut       <- buildDut( mkSoc, clk_port );
```

- Define control transactors for the testbench

```
Empty                     xshutdown <- mkShutdownXactor;
Empty                     xcontrol  <- mkSimulationControl( clk_cfg );
```

- Instantiate the inport and outport message port transactors, and connect them to the SoC

```
Get#(BusRequest)          xrequest  <- mkInPortXactor( clk_port );
Put#(BusResponse)         xresponse <- mkOutPortXactor( clk_port );
mkConnection(dut.rx, xrequest);
mkConnection(dut.tx, xresponse);
```

5.3.4 BSV Testbench

The DUT can be controlled by a testbench written in BSV which resides on the hardware side. The file `Tb.bsv` contains proxies to connect with the message ports in the `SceMiLayer.bsv` file. The names of the message ports come from the `mkBridge.params` file.

```
// instantiating the scemi proxies to connect to the system
SceMiMessageInPortProxyIfc#(BusRequest)   request  <-  mkSceMiMessageInPortProxy
                        ("mkBridge.params", "", "scemi_xrequest_inport");
SceMiMessageOutPortProxyIfc#(BusResponse) response <-  mkSceMiMessageOutPortProxy
                        ("mkBridge.params", "", "scemi_xresponse_outport");

SceMiMessageInPortProxyIfc#(Bit#(32))     simreq   <- mkSceMiMessageInPortProxy
                        ("mkBridge.params", "", "scemi_xcontrol_req_in");
SceMiMessageOutPortProxyIfc#(Bit#(32))    simresp  <- mkSceMiMessageOutPortProxy
                        ("mkBridge.params", "", "scemi_xcontrol_resp_out");

SceMiMessageInPortProxyIfc#(Bool)         shutdown <- mkSceMiMessageInPortProxy
                        ("mkBridge.params", "", "scemi_xshutdown_ctrl_in");
SceMiMessageOutPortProxyIfc#(Bool)        done     <- mkSceMiMessageOutPortProxy
                        ("mkBridge.params", "", "scemi_xshutdown_ctrl_out");
```

The remainder of the file is an FSM controlling the test sequence.

5.4 Building the Example

5.4.1 Infrastructure Linkage

Infrastructure linkage takes the description of the hardware side and generates a text file (the parameters file) describing the clocks, transactors, message ports, and link data. This file is read by the software side during initialization to bind the software-side message port proxies to the hardware-side message ports. Bluespec provides an infrastructure linkage tool called *scemilink* which reads the `.ba` files generated by Bluespec and creates a `.params` file. The default name for the file is `mkBridge.params`, where the top module is `mkBridge`.

5.4.2 Build utility

The `build` utility runs from a project configuation `.bld` file; the default configuration file is named `project.bld`. The file is like a makefile in that it is composed of a set of build targets, where each target describes a component of the project (DUT, testbench, etc.) along with commands (directives) describing the options and parameters of the target including directories, file names, module names, compiler options, emulation boards, and emulator options. When run, the utility determines which commands to execute, and in what order, for the given target.

There can be multiple definitions or versions for a particular target. For example, you could have two targets describing how to build the hardware DUT, where one generates Verilog code while the other generates Bluesim. When running the `build` utility, you specify the target for the current iteration.

To run the utility, enter the command `build` on the command line, along with one or more targets. The utility will determine the correct commands and the proper sequence of commands necessary to execute each stage. The project is built consistently each time and the defaults only have to be specified once per project.

The *Bluespec emVM User Manual* contains a complete reference for the `build` utility.

5.4.3 Simulating with the BSV Testbench

Entering `build` without any targets builds the default targets. Review the `project.bld` file in the directory `Part5`. The default targets in that file are `bsim_dut` and `bsim_tb`.

```
[DEFAULT]
default-targets:     bsim_dut bsim_tb
```

To build and run the simulation:

- Compile and link:

  ```
  build
  ```

- Once it completes, start the DUT:

  ```
  ./dut &
  ```

- Start the testbench:

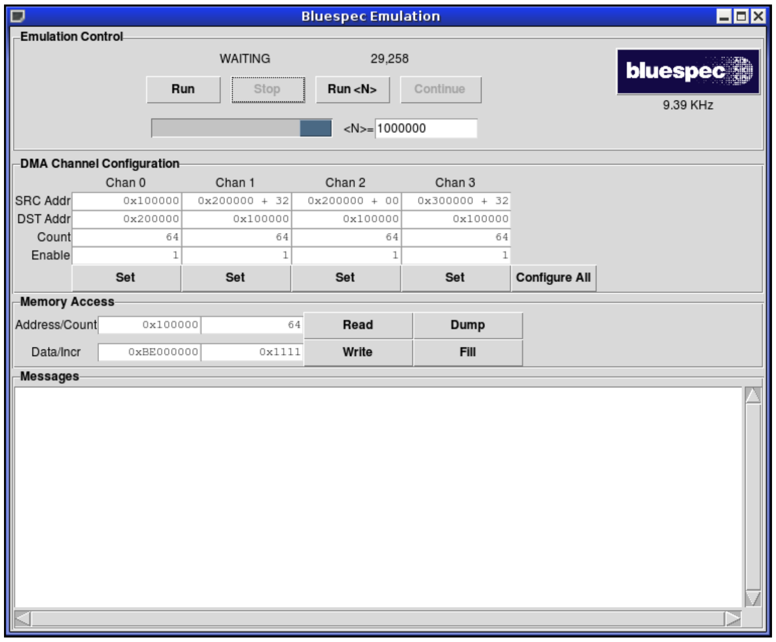
  ```
  ./tb
  ```

5.5 C++ Testbench and GUI

The testbench could also be written in C++ with a Tcl/Tk-based GUI residing on the host platform. An example of a C++ testbench and GUI is provided in this exercise. The GUI is shown in Figure 6.

As with the testbench written in BSV, the C++ testbench also contains proxies to communicate with the message ports on the hardware side. Each software transactor constructor definition includes the data type and name which must match the message port data type and name in the SCE-MI parameters file, `mkBridge.params`. The transactor is defined in two files: the header file (`DMATester.h`) and the source code file (`DMATester.cxx`).

The testbench code is in the file `TclTb.cxx`. All files to generate a GUI are also provided.



WAITING 29,258

9.39 KHz

<N>= 1000000

	Chan 0	Chan 1	Chan 2	Chan 3
SRC Addr	0x100000	0x200000 + 32	0x200000 + 00	0x300000 + 32
DST Addr	0x200000	0x100000	0x100000	0x100000
Count	64	64	64	64
Enable	1	1	1	1

Address/Count	0x100000	64
Data/Incr	0xBE000000	0x1111

Figure 6: GUI Testbench

5.6 Simulating with C++ Testbench and GUI

- Run the `build` utility to build a Verilog simulation executable and the C++ testbench:

```
    build vlog_dut tcl_tb
```

- After the build has completed successfully, start the Verilog simulation:

```
./dut &
```

- Once the design is running in simulation, you can start the GUI testbench:

```
./gui
```

Similar steps would be used if you were emulating the hardware side on an FPGA.

The GUI is completely customizable. The definition for the GUI is in the files:

- `gui_dut.tcl`
- `gui_top.tcl`
- `pkgIndex.tcl`
- `gui`

The testbench must also have the proper transactors to communicate with the GUI. You control the testbench and simulation through the GUI.