

Design Challenge #2: a Simple 4-Channel DMA Controller

February 2007
www.bluespec.com

© Copyright Bluespec, Inc., 2007 All Rights Reserved.

Overview

If you are not yet familiar with Bluespec, then prepare yourself for something very different from what you have seen or experienced before. Though some in the algorithm space have been successful at efficiently synthesizing higher-level math/DSP designs, Bluespec is the only solution that succeeds for control and complex datapaths. We invite you to take a deeper look – the typical response we get upon learning about Bluespec is “Bluespec wasn’t what I expected at all”. What you’ll find is truly unique, both in approach and results:

- Elevated hardware design and modeling that keeps designers 100% in control of the architecture and micro-architecture of their implementations.
- A unified environment for virtual prototyping, architectural exploration and IC implementation.
- No opportunity cost in adoption. As it layers incrementally on current flows – and generates readable, predictable Verilog RTL – Bluespec can be used one block at a time, without upending your current toolsets and methodologies. With less than a week of training, designers have consistently completed their first project, including design and verification, in less than half the time.

This document outlines Bluespec’s solution to Design Challenge #2, a simple 4-channel DMA controller, as summarized in DeepChip’s ESNUG #459, Item #5. The motivation behind the design challenge was to shift from marketing claims about high-level design into actual examples using real designs. Hopefully, this will enable us to debate what it means to implement control logic intensive designs at a high level versus RTL and explore the quality of the resulting implementations.

We invite you to review this DMA controller design (as well as the separately outlined 2x2 switch interconnect design). These designs are intended to give you a taste of what it means to design control logic and complex datapaths at a level above RTL. And, please keep in mind that the design was written to conform to the problem specification – Bluespec provides complete control over the micro-architecture and can be used to design any type of DMA controller.

Highlights of the DMA design include:

- The **high-level transactions** that are used to describe the core functionality of the DMA controller. These transactions make the design faster to design correctly, easier to read, and much more extensible.
- The **quality of results** of the final design. In 65nm TSMC (using Artisan libraries for CLN65GP), using Design Compiler the area is 44.43 Kum² and the design frequency is 769 MHz under worst case conditions and extended operating range (SS, 0.90V, 125 Degrees C). In 0.18u TSMC (using Artisan libraries for CL018G), using Design Compiler the area is 212.87 Kum² and the design frequency is 286 MHz under worst case conditions and extended operating range (SS, 1.62V, 125 Degrees C).

This design was done using Bluespec SystemVerilog (BSV). Bluespec designs can also be done at a high-level using SystemC – an example of this is provided with the 2x2 switch design solution for your reference.

This document is distributed along with full BSV source codes, generated Verilog, testbenches, test logs,

and synthesis results.

Problem specification

For your reference, the problem specification from DeepChip is repeated in this section.

Design Challenge #2: a Simple 4-Channel DMA Controller

Let's consider a DMA module which is, of course, configurable, and supports multiple concurrent transactions (multi-channel). At its interface, the following 3 groupings can be considered:

1. Configuration port is a target interface (similar to the OCP-like socket interface described in the 2x2 interconnect).
2. Memory port is an initiator port (OCP-like socket interface) on which both read and write DMA transfers operate.
3. Third group contains interrupt lines which begin transfers and status lines which mark end of transfers on a per channel basis (there are a pair of interrupt/status lines per channel. Once a channel has been configured, an interrupt request indicates to the DMA controller to begin an operation; status indicates completion)

With regards to features:

- For uniformity, all busses are 32 bits wide.
- 4 channel DMA, where all channels can have pending read or write operations; channel number dictates priority for read/write operations. Channel 0 take priority over channel 1, etc.
- Memory requests can be sent every cycle; can be delayed due to back-pressure from the memory port.
- Memory responses are in-order for each channel, but may be out of order between channels. Response latency from the memory is completely arbitrary.
- Memory requests and responses should be tagged with a 2-bit thread ID to identify the request/response channel.
- Configuration allows setting of source address, destination address and number of words transferred, plus the enabling of the channel via software configuration or hardware interrupt.
- Write operations from the DMA controller take precedence over read operations.
- The ports should be fully utilized when possible.

Assume no more than 4 outstanding memory read requests per channel; largest transaction for the memory transaction is 64 bytes¹ (the DMA transactions may be larger).

Testbench Requirements:

We'll leave this pretty open ended. Must demonstrate the core functionality and behavior including

¹ As the memory subsystem is outside of the DMA and part of the testbench, the 64 byte memory transaction requirement doesn't make much sense in the context of the DMA implementation. Perhaps as a future enhancement, there should be a burst specification for the MMU interface.

interleaved operations.

John, as a measure of flexibility, and to more closely measure real life chip design conditions, I'd like to suggest that you later add 2 mystery features after the basic features have been implemented. Ideas for this include: pre-emption, reservation, additional channels, additional ports, additional addressing mode, etc.

Bluespec 101

Bluespec enables design at a much higher level than RTL, while leaving the specification of the architecture to the designer, giving him/her 100% control over the quality of the implementation including latency, area, power and timing. With Bluespec, designs and models are expressed using transactions to define both internal module behavior within modules and interfaces between modules.

This section is not intended to make you an expert, but it is intended to give you enough background to understand the design of the DMA controller.

There are four key ideas to keep in mind while looking through the code:

- **Explicit state** – all state, e.g. registers, RAMs, FIFOs, ROMs, any sub-module,..., is explicitly specified by the designer. No state is added or subtracted by the Bluespec compiler. Every module will have its state explicitly coded – though you get to leverage high-level, abstract types for more succinct and expressive specification of that state. You'll see this state reflected, as is, in the Verilog generated by the Bluespec compiler.
- **Transactions for behavior within modules** – instead of ALWAYS blocks, you'll see behavior coded in structures called RULEs, which describe behavior using isolated transactions. A rule is a description of what to do for a particular situation. A rule describes what state to update when certain designer-specified conditions are true. For example, if your design required some things to happen when an interrupt happened, you would code a rule to perform these things whenever the conditions for an interrupt were true.

The key advantage of this approach is that designers can code each "RULE" in isolation without regard to what else is happening in the system – this allows designers to design incrementally (which really simplifies the complexity of hardware design, where you typically have to pay extremely close attention to resource contention, such as multiplexing, communication, synchronization). For example, when you code the interrupt rule, you can do so without worrying about its interaction with the rest of the design, such as the logic responsible for "normal" operation – you let the compiler worry about properly flagging and handling unanticipated interactions. The tool is responsible for scheduling as many rules to run in parallel as possible. During this process, it will identify any shared resource situations, flag them for you, and automatically insert the proper multiplexing for you as well as the logic to schedule access to these common resources.

Here's the syntax for a rule (using Bluespec SystemVerilog (BSV)):

```
rule <ruleName> (<boolean cond>);  
    <state update(s)>  
endrule
```

On first blush, a rule sort of operates like an **if...then...** When the <boolean cond> is TRUE, then the <state update(s)> occur(s). If there are multiple updates, then all have to happen together, or none can happen at all.

Well, that's the simplified, introductory view. The power is in the additional sophistication. In addition to the <Boolean cond>, there are two additional considerations to whether a rule "executes" its state updates:

- The first is that, although most rules will execute in parallel in hardware, Bluespec's compiler ensures that each rule executes in the hardware *as though* it were the ONLY thing operating at that moment. Another way to describe these rules is ATOMIC transactions. This is what allows designers to think about and design each rule by itself and get it right – the compiler ensures that this atomic behavior is maintained in the hardware.

When does this really matter? It comes into play particularly when there are rules that might want to update common (or shared) state. In RTL hardware design, designers need to keep track of all the potential updates to shared state AND properly schedule access to this shared state by many different operations in the hardware. With Bluespec, these shared resource conditions are automatically recognized and scheduled (albeit with 100% designer visibility and control). And, the compiler will even recognize when rules have mutually exclusive access to the same state and ensure that they will always have access to that state when needed.

The end result: you can never have unintended race conditions accessing shared resources. This eliminates a major source of obscure, subtle errors in hardware design.

As such, a rule's Boolean condition may be TRUE, but its operation may conflict with another rule which takes precedence which may prevent its execution in a given cycle (due to an attempt to access shared state). The compiler generates the optimal parallel execution of rules in hardware while properly scheduling access to shared resources.

- The second consideration has to do with how interfaces work. With Bluespec, all state elements such as registers and memories have interfaces just like traditional design modules. But with Bluespec, these interfaces are transactional – they not only describe the wire interfaces, but they also understand how to pass data and the proper conditions for doing so.

These proper conditions also get considered in determining when a rule can execute. The easiest way to understand this is to review the following rule example:

```
rule pass_through (state == FULL_THROTTLE);  
  y <- fifoIn.deq();  
  z = f(y);  
  fifoOut.enq(z);  
endrule
```

This rule has an explicit condition that says it should execute when in the "FULL_THROTTLE" state. But, when it executes, it is supposed to dequeue a value from fifoIn and, at the same time, enqueue the result of f(y) into fifoOut. What if there's nothing to dequeue from fifoIn? What if fifoOut, the place you want to put the data, is full? Normally, designers have to design the control logic to check all of these things. With Bluespec, these conditions (for all interfaces of modules and state, such as registers, FIFOs, ...) become implicit conditions of the rules that use the interfaces. If ANY of the interfaces is not ready, then the rule cannot execute.

Even in this basic rule example, a lot of complex behavior is simplified. The "pass_through" rule will move data from one FIFO to another after manipulating it with f() only when:

- In the proper state: state == FULL_THROTTLE
- All the interfaces allow it (in this case: that there is data to be moved from fifoIn and there is also a place to put it into fifoOut)
- There are no other rules that want to dequeue from fifoIn or queue into fifoOut in this cycle. If there were, only one would be allowed.

When the rule executes, all three actions will happen simultaneously: data would be dequeued, passed through the combinational logic in the function `f()`, and the result would be queued into another FIFO. A lot of explicit control logic in RTL is handled succinctly and implicitly with rule and interface transactions with Bluespec.

- **Transactions for interfaces between modules** – as we just introduced in the last section, instead of port lists, interfaces are transactions defined using rule-based interface methods, as shown below, using some extensions to the notation of SystemVerilog or SystemC (the former is shown below):

```
interface FIFOBuf#(type x_t);
    method Action enq (x_t x);           // Note: an "Action" means that it changes state
    method ActionValue#(x_t) deq ();    // Note: an "ActionValue" means that it returns a value
                                        // and changes state
    method Action clear ();
endinterface
```

[Note: the `(type x_t)` construct is just SystemVerilog's notation for type parameterization (also known as polymorphism, or genericity), i.e., `x_t` is a type variable representing the type of items stored in the FIFO.]

The `enq` interface method encapsulates all the ports for enqueueing: the input data bus (whose width depends on the particular data type to which the generic type `x_t` is instantiated), the output `READY` signal (which tells the outside world when the FIFO is able to accept an enqueue), and the input `ENABLE` signal (which the outside world uses to tell the FIFO that it is enqueueing). Similarly, the `deq` method encapsulates all the ports required for dequeueing: the output data bus, the output `READY` signal (which tells the outside world when valid data is ready to be dequeued) and the input `ENABLE` signal. Based on the interface method's guard conditions in the design (e.g. "not full" for an enqueue), the compiler generates the ready and enable signals automatically in the output Verilog RTL.

What does it mean to encapsulate the ports? Let's look at the code for an enqueue interface method in a FIFO:

```
method Action enq(x) if (notFull);
    rw_enq.wset(x); // Pass the input through (when 'notFull')
endmethod
```

The *notFull* is a Boolean signal that is part of the internal FIFO implementation which simply indicates whether the FIFO has space or not. While not exposed to an external user of the FIFO, this value becomes an implicit condition on whether the method can be called and it forms the value behind the `READY` signal associated with this method in the generated RTL. This *notFull* condition becomes an additional condition that is ANDed into the composite conditions on any rules that use this method.

In general, method arguments become module input data bus ports. Method results (such as that returned by `deq`), become output data bus ports. A method can have multiple output data bus ports because return types can be structs (records) with multiple fields, and vectors. All methods have an output `READY` signal (which is the interface method's condition). All `Action` and `ActionValue` methods (like those shown) have input `ENABLE` signals. `Action` and `ActionValue` methods can cause a state change inside the module. A third kind of method, which we call `Value` methods (not shown in this example), are purely combinational—their results are combinational functions of their arguments and internal module state. The compiler optimizes away `READY` and `ENABLE` signals of a method if it proves that they are always asserted.

A client module that uses the FIFO contains Rules that invoke the `enq` and `deq` methods, as in the example below:

```

module mkClient (...);
  ... instantiate fifo ..
  .
  rule upstream (... cond1 ...);
    ... other actions ...
    fifo.enq (expr1);
  endrule

  rule downstream (... cond2 ...);
    x <- fifo.deq ();
    ... other actions ...
  endrule
endmodule

```

As discussed before, each rule has an explicit condition, depicted above as the expressions *cond1* and *cond2*. These are pure combinational Boolean expressions. Each rule also contains one or more actions that can be executed atomically only if the rule condition is true. For example, the *upstream* rule contains an action that enqueues the value of expression *expr1* into the FIFO, and the *downstream* rule contains an action that dequeues an item *x* from the FIFO.

The conditions of all methods invoked by a rule are incorporated into the overall condition of the rule. For example, the ENQ_READY signal is “AND”ed with *cond1* to determine the overall condition of the upstream rule. The DEQ_READY signal is “AND”ed with *cond2* to determine the overall condition of the downstream rule.

So to reiterate and summarize, a rule can only execute if all its conditions permit. When it executes (we call it: fires), all its actions, including all the actions in all the methods that it invokes, are executed simultaneously as one composite atomic action. Thus, the upstream rule can only fire if ENQ_READY is true, and then the enqueueing action becomes part of the overall atomic action of the rule. When the rule fires, the enqueued data is driven and ENQ_ENABLE is asserted.

The condition of a method or a rule is necessary, but not sufficient, for a rule to fire. In particular, since rules can share resources (such as the FIFO above), simultaneous firing might not be possible while maintaining atomicity, i.e., if simultaneous firing would lead to inconsistent states. The compiler emits scheduling logic to ensure that simultaneous firing is only possible if it maintains atomicity.

When compiling a FIFO implementation, the compiler performs a systematic analysis that infers whether the enq and deq methods can be operated simultaneously safely, and under what conditions. Note, different FIFO designs may or may not permit such simultaneous operation.

This interface information is recorded by the compiler with the FIFO implementation. Then, when compiling mkClient, the compiler uses this information to introduce suitable control logic in mkClient to guarantee that the upstream and the downstream rules can fire simultaneously only when conditions permit them to do so safely.

- **High-level, abstract types, parameters and polymorphism** – along with the power to express the behavior of the design as a set of transactions and transactional interfaces, you’ll also see more expressive design using high-level abstract types, such as structures and enumerated types. Also, though less emphasized in this particular design, there is the power to parameterize designs along almost any dimension. Examples include the ability to parameterize with: interfaces (which type of interface to use), functions (e.g. what kind of sort algorithm), types (polymorphic design), sizes, and even modules. At compile time, the tool will generate the hardware specified by the chosen parameters – and elaborate the hardware using the parameters as well as other procedural controls: conditionals such as *if*, *for* loops, and even recursion.

Problem Solution

The design for the DMA controller is outlined in a block diagram in Figure 1. It is designed such that each DMA channel has its own independent sets of behavioral descriptions, written with rules, and registers to maintain the channel's state. In addition, there are some common rules and registers that are not DMA channel specific.

Overview of the Basic DMA Operation

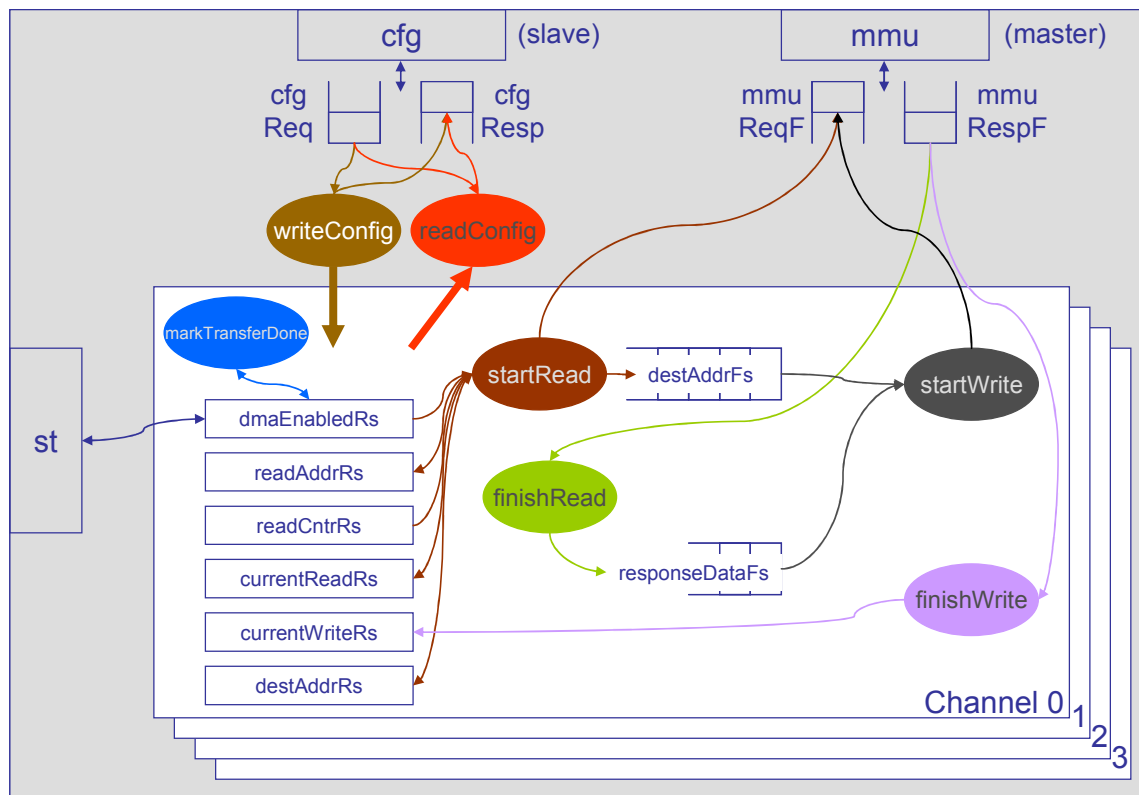
Once a DMA operation is configured on a particular channel, these are the basic steps the channel will go through to move memory from one address to another:

1. The relevant channel on the DMA controller issues a request for the next 32-bit word from the source address location. This is done by issuing a read request on the mmu socket for a RD operation for this channel and source address.
2. When the response information comes back from the mmu socket, which includes the data at the previously requested source address, the response data is queued up internally to get ready for the write operation.
3. The next queued write operation is issued as a write request on the mmu socket. The request includes the target address, the data to be written, and an indication that it is a WR operation for this channel.
4. The DMA operation for this given 32-bit word is complete when the response information for this write operation comes back from the mmu socket.

(Steps 1-4 will be repeated until the entire DMA operation is complete.)

5. When the entire transfer is done, the DMA channel is disabled until subsequent configuration.

Figure 1: Block Diagram of DMA Controller Design



DMA Design – Core Operation

The DMA design is straightforward because we can write a separate rule to define each DMA step, as outlined above. As there are five basic steps in each channel's DMA operations, the Bluespec design has five rules to describe the behavior for each channel. Since the rules are almost identical for each channel (the only difference being the channel number), instead of separately writing all 20 rules (5 rules/channel X 4 channels), we created two functions. One creates rules for the first four steps above. The other creates the rule that cleans up after a DMA is completed – that is, when the entire transfer is done.

The main design of the DMA controller is in DMA.bsv. After the registers and FIFOs are instantiated in the design, there is a function called generatePortDMARules(). This function creates the four of the rules governing the first four steps above. When called (this is done once at compile/elaboration time) with a channel number, it creates the following four rules for that channel:

startRead – this rule manages the first step described above under basic DMA operation. The explicit conditions for the rule are:

- The channel must be enabled
- The number of words to read are more than the number currently read

When this rule executes, it issues a read request on the mmu socket interface for the next word to be transferred. It internally queues up the destination write address for the data to be read. And, finally, it increments the following addresses and counters for the channel: the current source (read) address, the number of words read, and the current destination (write) address. All of these operations happen in parallel, together (or, atomically).

Note that there are also several implicit conditions that need to exist for this rule to execute. One of the beauties of designing with Bluespec is that all the proper checks for these conditions are automatically and correctly implemented – avoiding the manual implementation of a lot of tedious and error-prone control logic. There's no magic in this – you'll end up with the same implementation as you'd hand write for a given micro-architecture – but, as the designer, you get to focus on the functionality and micro-architecture rather than the low-level implementation details. What are these implicit conditions that are automatically checked?

- The internal FIFO that holds the queued destination write addresses needs to be able to take another address. It must be non-FULL (or, there must be a simultaneous dequeue)
- The mmu socket must be able to: 1. Take a request; and 2. Must not be in contention with another, higher priority, socket request (you'll find in DMA.bsv that startWrite takes precedence over startRead, as writes take precedence over reads, AND that lower number channels take precedence over higher number channels).

Writing this rule is simple. You describe the basic conditions of the first step of the DMA and what you want to happen.

finishRead – this rule manages the second step described above under basic DMA operation. The explicit condition for the rule is:

- The response info that arrived on the mmu socket is for a read operation on this respective channel

When this rule executes, it removes the response info from the FIFO in which it is stored. And, it also internally queues the response info for subsequent use by the write operations.

As with startRead, finishRead can only execute when both its explicit and implicit conditions are true. So, this rule will execute only when: 1. The response info from the mmu socket interface indicates that it is for a RD operation on this channel; and, 2. The internal FIFO, responseDataFs, must have space to hold the response info.

startWrite – this rule manages the third step described above under basic DMA operation. Note that there is no explicit condition defined for this rule. It should execute whenever it can.

When this rule executes, it issues a write request to the mmu socket interface based on the data at the head of the responseDataFs FIFO (which holds the response info that came back from the read operation) and at the head of the destAddrFs FIFO (which holds the destination address of the write). And, at the same time, it also removes the data at the head of both queues.

Of course, this rule can only execute when several implicit conditions all exist:

- There must be data in both of the internal FIFOs: both a destination address and response info data from the associated read.
- The mmu socket interface must be ready to take the write request, which means it can take another request on this cycle AND another write request isn't being made from a higher priority channel (remember: in this design, writes always take precedence over reads within a channel – but, a write from a higher priority (lower numbered) channel always take precedence over that from a lower priority channel).

When all these conditions are true, then the rule does its stuff. The design is simple and succinct.

finishWrite – this rule manages the fourth step described above under basic DMA operation. Its operation is pretty simple – if the response info on the mmu socket interface is an acknowledgement of the write operation for this channel, then dequeue the response and increment the number of writes.

For each channel, there is one more rule to write, which is managed by a separate function called

generateTransferDoneRules(). This function creates a single rule for the final, fifth step described above under basic DMA operation. It handles the cleanup when a DMA transfer is complete. When the function is called (this is done once at compile/elaboration time) with a channel number, it creates the following rule for that channel:

markTransferDone – when a transfer is complete, this rule does all the cleanup. It sets the Enabled flag for the channel to FALSE and resets to zero the counts of the number of reads and writes completed on the channel.

DMA Design – Setting Priorities for Channels and Read/Writes

The relative priorities for the rules is established with descending_urgency statements. Within the function generatePortDMARules(), there is a statement:

```
(* descending_urgency = "startWrite, startRead" *)
```

This statement says that write operations take precedence over read operations within each channel. Explicitly, it says the startWrite rule takes precedence over the startRead rule for this channel.

Later in the DMA design, there is a series of calls to rJoinDescendingUrgency(). These calls establish the priorities between channel numbers, with lower channel numbers taking precedence over higher channel numbers.

DMA Design – Configuration

In order to configure the DMA controller's internal registers, there are two main rules for writing and reading register configurations, writeConfig and readConfig, respectively. Configuration requests primarily come through the cnfg socket interface. Though this is the only way to write or read most of the registers, DMA channels can also separately be enabled through the status interface (per the specification).

When a configuration request comes in via the configuration socket interface, it first goes into the configuration request FIFO, cnfReqF. The request operation, reqOp, in the configuration request determines whether it is a write or a read request. The writeConfig rule or readConfig rule executes depending on the operation requested – an internal register is either read or written.

There is a function, selectReg(), to support reading and writing configuration registers. This function identifies the targeted register based on the reqAddr value.

DC Synthesis Results

For details on the synthesis results, including a Verilog netlist for the design and information about the runs, please refer to the directory: DMA/DCSynthesis. The design was synthesized for both 65 nm and 180 nm.

Here were the main assumptions used for 65 nm:

- Design Compiler Version: X-2005.09-SP3
- Technology: TSMC CLN65GP process
- Library: Artisan 10-Track Advantage (SS Process Spice Models, 0.90V, 125 Degrees C)
- Added Input/Output delay = 25% of clock period.

And the results for 65 nm were as follows:

- Frequency: 769 MHz
- Area: 27.77 K gates, 44.43 μm^2 (1 gate = 1.60 μm^2)

Here were the main assumptions used for 180 nm:

- Design Compiler Version: X-2005.09-SP3
- Technology: TSMC CL018G process
- Library: Artisan SAGE-X (SS Process Spice Models, 1.62V, 125 Degrees C)
- Added Input/Output delay = 25% of clock period.

And the results for 180 nm were as follows:

- Frequency: 286 MHz
- Area: 21.33 K gates, 212.87 μm^2 (1 gate = 9.98 μm^2)

Summary

This paper has outlined the design of a simple 4-channel DMA controller using Bluespec SystemVerilog (BSV). All of this was only possible because of the high-level abstraction mechanisms, formal semantics and composability of the constructs in Bluespec SystemVerilog – and, of course, we support these capabilities in SystemC as well. For the first time, there is:

- A solution that unifies virtual prototyping, architectural exploration and IC implementation.
- A high-level solution for control and complex datapaths – but with 100% of the control and quality of results of hand-coded RTL.

Hopefully you've been able to get a taste of why we say that Bluespec is a simpler and more scalable way of managing complex concurrency & interface protocols vs. RTL. We are also the only one significantly improving the design of control logic and complex datapaths. Such troublesome designs are:

- The hardest to get right. Managing complex concurrency with lots of shared resources (both local and across chip) is very hard to implement correctly. Think about the challenges properly managing back pressure, interface protocols, race conditions, deadlock/livelock conditions...
- The areas where most of the bugs (especially the subtle ones that later bite you) reside.
- The most prominent by far. One of our customers surveyed their IP development roadmap and found that designs with control logic and complex datapaths represented 90% of their planned projects.

This is where Bluespec shines. Everything else is just RTL. Can you afford not to take a closer look?