

Design of a multi-port memory controller in Bluespec SystemVerilog

Gert-Jan Tromp

**Dizain-Sync
G.J.Tromp@d-sync.com**

This report describes the design of a multi-port DDR2 memory controller using the high-level hardware description language Bluespec SystemVerilog (BSV). The purpose of the implementation of the design was the evaluation of Bluespec SystemVerilog and the Bluespec compiler and to compare the Bluespec design flow with a traditional design flow using register transfer level Verilog or VHDL.

Introduction

The introduction of the hardware description languages VHDL (Very High-speed integrated circuit Description Language) and Verilog in the 1980's was a significant step forward for the development of digital electronics. It freed digital designers of the use of schematics and allowed the use of programming language techniques instead. However, as far as the hardware description languages are concerned, state-of-the art in digital design has not much changed since the late eighties and early nineties. Re-use in software engineering has been very successfully improved, supported by programming constructs available in languages such as C++ and Java. The lack of such constructs in VHDL and Verilog has limited the application of re-use to relatively large blocks often referred to as Intellectual Property (IP).

The Bluespec SystemVerilog (BSV) language, based on the upcoming standard SystemVerilog language has more sophisticated programming constructs which allow re-use at a much lower granularity. To illustrate the useability of those constructs, this report describes the implementation of a multi-port DDR2 memory controller using BSV. This design was chosen for the evaluation, because it contains both low-level issues (related to the DDR2 interface) and higher-level issues (related to the multiple ports). The remainder of this report is organised as follows :

- a specification of the multi-port memory controller is presented.
- the architecture and the two layers of the design are described:
 - the lower level, which deals with the DDR2 memory at the signal level

- the higher level which implements the multiple ports and the scheduling
- the lower level part of the design is described.
- the higher level part of the design is described
- the verification environment is described
- the synthesis environment and synthesis results are described
- conclusions and recommendations are presented

Specification of the multi-port memory controller

SPECIFICATION

The evaluation design is a multi-port DDR2 Memory Controller with the following specifications:

- DDR2 memory controller with n ports,
- each port is either a read or a write port.
- The sequence in which each port get access to the memory is fixed, which guarantees the throughput for each port.

The following items are parameterized:

- number of ports n
- direction of each port
- schedule (e.g. the sequence in which each port gets access to the memory)
- burst size (minimum number of transfers of each entry in the schedule)
- fifo size of each port

Refresh and initialization are performed by the controller

Design of the multi-port memory controller

This section describes the design of the multi-port memory controller. First, the architecture will be described, and then some of the implementation details of the design will be described.

ARCHITECTURE

Figuur 1 Shows the architecture of the multi-port memory controller.

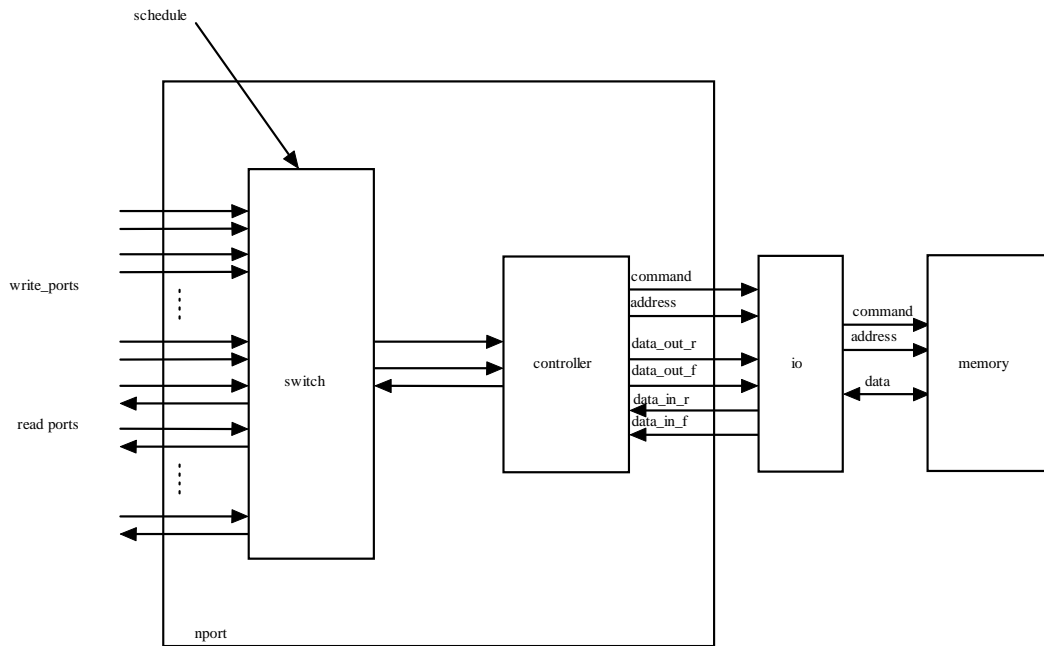


FIGURE 1. Architecture of the multi-port memory controller

The architecture in Figure 1 contains the following sub-blocks:

- *nport*: the core of the multi-port memory controller
- *switch*: the block that allows one of the write or read ports access to the memory
- *controller*: the block that performs the accesses to the memory
- *io*: the block containing the DDR2 specific IO's, including phase shifting and double-data rate conversions
- *memory*: the external memory

The ports will be serviced according to a fixed schedule (see Figure 1). This schedule consists of an array of port numbers. The external memory is industry standard DDR2 memory. For the purpose of this report, it is not relevant about the discuss the operation of the DDR2 memory interface in great detail, however, some of the interface functionality is explained in the next section to illustrate some of the design choices.

The remainder of this section is split in two parts: one part that deals with the external memory interface (the *controller* block in Figure 1) and one part that discusses the higher level functionality (the blocks *nport* and *switch* in Figure 1).

DDR2 interface level

To illustrate some of the requirements that exist and trade-offs that can be made when interfacing with an external DDR2 memory, Figure 2 shows a basic write operation.

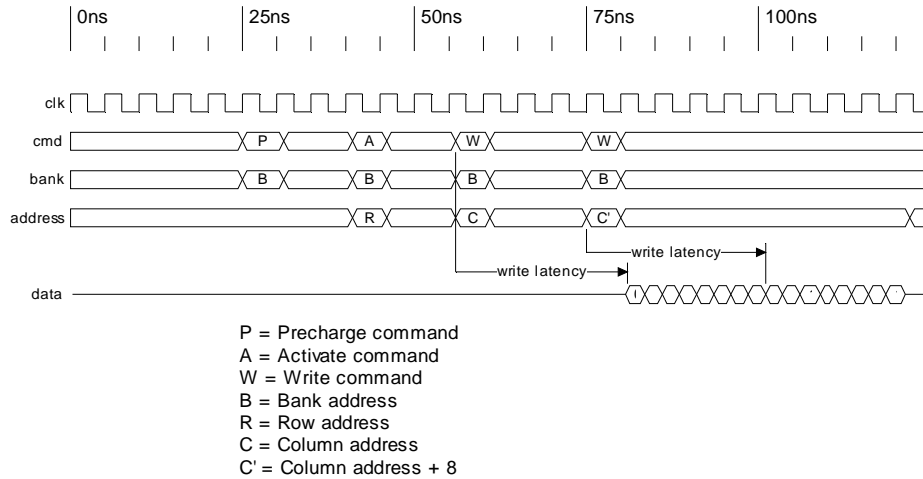


FIGURE 2. DDR2 Write burst timing diagram

In order to access a memory cell inside a DDR2 memory, the following operations are necessary:

1. **Precharge** of the bank in which the memory cell is located (a DDR2 memory can have upto 8 banks)
2. **Activate** of the row in which the memory cell is located (a 1Gbit DDR2 memory has 8192 rows).
3. A **Write** command must be issued together with the column address of the memory cell (a 16 bits wide 1Gbit DDR2 memory has 1024 column addresses in a single row). Row address lines and Column address lines are shared
4. After a specified delay (the write latency) a sequence of data items is passed to the memory, one item on each edge of the clock. A burst of 4 or 8 data items is then written to the memory. For this report only a burst size of 8 is considered. As shown in Figure 2, to write a burst of 16 consecutive data items to the memory, two **Write** commands have to be issued and the second can overlap with the execution of the first command, so that the full bandwidth of the interface can be utilized for accesses to the same row.

The data items are accompanied by a strobe signal which alternates in value for each data item and which is used to compensate for PCB routing delay. The necessary strobe signals are implemented in the *io* block in Figure 1 and are considered of no interest for this report.

For completeness, in Figure 3, the timing diagram of a read burst is shown.

DDR2 interface level

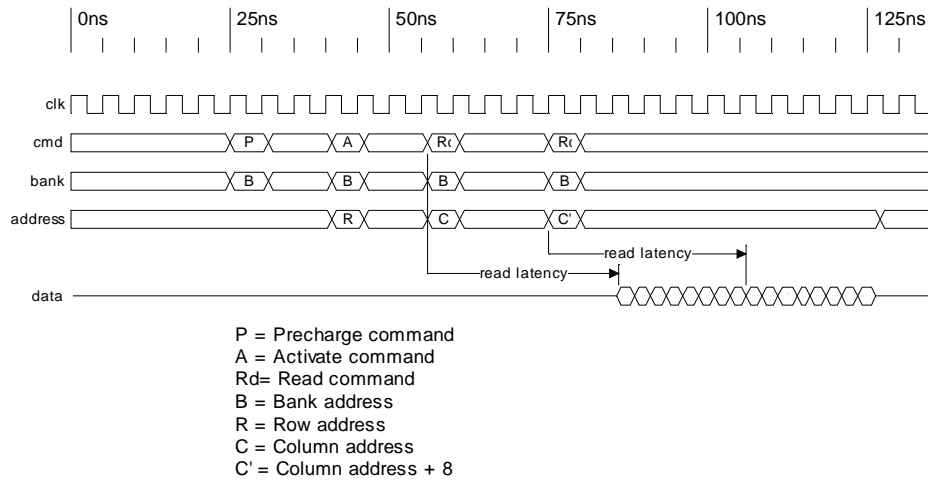


FIGURE 3. DDR2 Read burst timing diagram

The operations are similar as for the write burst; a **Read** command is issued instead of a **Write** command and the data on the memory's data bus is driven by the memory instead of by the controller.

In Figure 4, the hierarchical design of the controller is shown. The controller itself can be accessed by *read* and *write* methods and the associated data is stored in and obtained from the controller by *put* and *get* methods. At the heart of the controller is the *core* module, which can be controlled at the command level by methods such as *precharge*, *activate* and *read*.

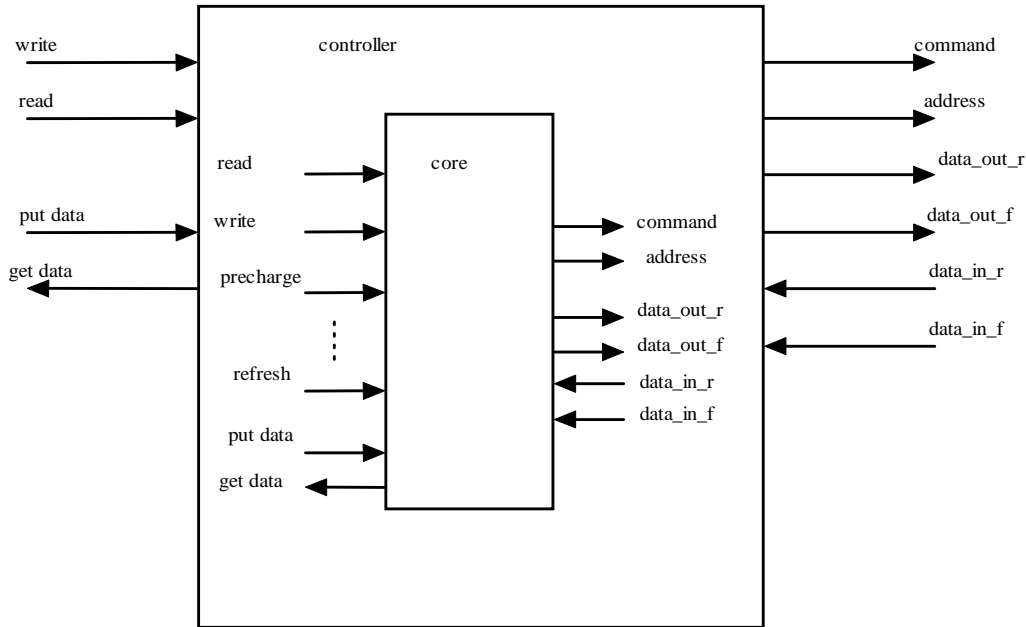


FIGURE 4. Controller hierarchy

In Bluespec SystemVerilog the interfaces can be described separately from the modules and in the next part of this section the interfaces of the *controller* and *core* modules will be described.

BSV INTERFACES

This section describes the Bluespec SystemVerilog interfaces. The *core* module (see Figure 4) is accessed using the interface called *DDR2_intfc*, which is described as follows:

```
interface DDR2_intfc#(type row_addr_type, type col_addr_type,
    type data_type);
    interface Put#(Tuple2#(data_type, data_type)) inport;
    interface Get#(Tuple2#(data_type, data_type)) outport;
    method Bool data_available;
    method Action read(Bit#(3) ba, col_addr_type a);
    method Action write(Bit#(3) ba, col_addr_type a);
    method Action enable();
    method Action disable();
    method Action nop();
    method Action activate(Bit#(3) bank, row_addr_type row);
    method Action load_mode(Bit#(3) r, row_addr_type m);
    method Action refresh();
    method Action precharge_all();
    method Action precharge(Bit#(3) bank);
```

DDR2 interface level

```
method Action d_in_rising(data_type r);
method Action d_in_falling(data_type f);
method row_addr_type address();
method Bit#(3) bank_address();
method data_type d_rising();
method data_type d_falling();
method Bit#(1) d_en();
method Bit#(1) d_sample();
method Bit#(1) dqs_en;
method Bit#(1) dqs;
method Bit#(1) cke;
method Bit#(1) cs;
method Bit#(1) ras;
method Bit#(1) cas;
method Bit#(1) we;
endinterface
```

The first part of this interface, from the *data put* interface to the *precharge* method are the methods through which the controller accesses the functionality inside the core module, whereas the last part of the interface, from the method *d_in_rising* to *we*, forms the signals that are transformed by the *io* block to the signals that are connected to the external memory (see Figure 1).

The interface was defined polymorphic, using *addr_type* and *data_type*, which makes it easier to change, but because these types relate to external signals, the benefit of the polymorphism in this particular case is limited. Furthermore, all data signals come in two versions: one that, on the external interface, is valid on the rising edge of the clock and one that on the external interface is valid on the falling edge of the clock.

```
interface DDR2_ctrl_intf#(
    type addr_type,
    type data_type,
    type row_addr_type,
    type col_addr_type);
    interface Put#(Tuple2#(data_type, data_type)) inport;
    interface Get#(Tuple2#(data_type, data_type)) outport;
    method Bool data_available;
    method Action write(addr_type a);
    method Action read(addr_type a);
    method Bool write_available;
    method Bool read_available;
    method row_addr_type address();
    method Bit#(3) bank_address();
    method data_type d_rising();
    method data_type d_falling();
    method Action d_in_rising(data_type r);
    method Action d_in_falling(data_type f);
    method Bit#(1) d_en;
    method Bit#(1) d_sample;
    method Bit#(1) dqs_en;
    method Bit#(1) dqs;
    method Bit#(1) cke;
    method Bit#(1) cs;
    method Bit#(1) ras;
    method Bit#(1) cas;
    method Bit#(1) we;
endinterface
```

BSV MODULES

The implementation of the *DDR2_core* module has been kept simple, most methods translate directly into command issued to the memory. The command bits have been defines as registers, e.g.:

```
Reg#(Bit#(1)) cs_r <- mkReg(1);
```

And for example the *activate* method was described as:

```
method Action activate(Bit#(3) bank,
  Bit#(row_address_width) row);
  cs_r <= 1'b1;
  ras_r <= 1'b1;
  cas_r <= 1'b0;
  we_r <= 1'b0;
  ba_r <= bank;
  a_r <= row;
endmethod
```

Because the *read* and *write* methods require a number actions that have to be performed after some delay, they were implemented by using a shift register for the delay and a set of rules to implement the actions (for the complete source: [Nport BSV]).

The implementation of the controller (*ddr2_ctrl_parametric*) has a bit more complexity, as it contains the following functionality:

- initialization sequence
- read burst
- write burst
- refresh

The initialization sequence is required by the DDR2 specification [JESD79-2A] and was implemented by using a counter and a set of rules that perform the required actions at specific counter values, e.g.:

```
rule init_lm_1_rule(initializing && (counter == 2406));
  ddr2_if.load_mode(1, 'b0_0_1_000_0_000_1_0_0);
endrule
```

The above rule specifies that when the counter value reaches 2406, the *load_mode* command of the *DDR2_core* module is called. After the initialization sequence is performed, a register bit is set and read and write actions can be performed. To implement the read and write actions, shift registers were used to delay actions and to allow overlapping requests (see Figures 2 and 3).

For the purpose of this evaluation, the implementation of the refresh was done at the end and the other functions (initialization, write and read bursts) were implemented without any provisions for the refresh. Using a conventional RTL implementation flow, this would require to a lot of changes to code already written and tested, or result in an inefficient implementation. For a design in Bluespec SystemVerilog this should be a perfectly acceptable approach as the Bluespec compiler can deal with it effectively.

The refresh was implemented using a counter to keep track of time. When the counter reaches the value for the refresh period, any currently active read or write burst is finished and thereafter the refresh is started. During the refresh, no other read or write methods can be called.

PORT LEVEL

Basically, the controller described in the previous sections, can be seen as a one-read and one-write port memory controller. To extend that functionality to more read and write ports, only a set of FIFO's and the functionality to connect those FIFO's to the controllers read and write ports is necessary. In the next parts of this section the two main modules that implement the functions mentioned above will be described:

- the *scheduler* module which contains the FIFO's and the ability to select one of the FIFO's.
- the *nport* module which contains the controller, the scheduler and the main control functionality.

In the remainder of this section, first the interfaces will be described and then some of the design choices in the modules will be explained.

BSV INTERFACES FOR THE PORT LEVEL

As mentioned before, the *scheduler* module contains the FIFO's in which the addresses and data for the separate read and write ports are stored. The Bluespec SystemVerilog *Vector* interface class was used to access those FIFO's. The FIFO's are implemented using the FIFO modules that are standard with the Bluespec compiler. The FIFOF interface class was used to access those FIFO's because that interface has methods for inspecting if the FIFO is full or empty. This resulted in the following interface:

```
interface Scheduler_intf#(type addr_type, type data_type,
    type num_a_infifo, type num_a_outfifo);
    interface Vector#(num_a_infifo, FIFOF#(addr_type))
        chai_fifo;
    interface Vector#(num_a_outfifo, FIFOF#(addr_type))
        chao_fifo;
    interface Vector#(num_a_infifo,
        FIFO#(Tuple2#(data_type, data_type))) chi_fifo;
    interface Vector#(num_a_outfifo,
        FIFO#(Tuple2#(data_type, data_type))) cho_fifo;
    method Bool notEmpty();
    method ActionValue#(Tuple2#(Bit#(1), addr_type)) get_a();
    method ActionValue#(Tuple2#(data_type, data_type)) get_d();
    method Action put_d(Tuple2#(data_type, data_type) d);
    method Action next();
    method Action switch_datapath();
endinterface
```

As can be seen in this interface, the FIFO's for the addresses and data are separate to allow for a more effective implementation. Furthermore, the width of the data FIFO's is twice the width of a single data item and therefore twice the width of the external memory interface, so that the full data rate of the external memory interface can be supported.

External control of is provided by the methods:

- *get_a*, to pop an address value from the selected FIFO, whereby an extra bit is added to indicate if it is a read or a write burst.
- *get_d*, to pop a data value from the selected data FIFO (in case of a write burst).

- *put_d*, to push a data value onto the selected data FIFO (in case of a read burst).
- *next*, to indicate to the *scheduler* module that the next port can be serviced.
- *switch_data_path*, to indicate to the *scheduler* module that the data path can be switched to the port that has to be serviced (Note that this scheme allows an overlap period in which commands are issued belonging to one port and data is read or written belonging to another port).

The interface to the top-level of the multi-port memory controller can be described reasonably compact:

```
interface Nport_parametric_intf#(
    type addr_type,
    type row_addr_type,
    type col_addr_type,
    type data_type,
    type a_inports, // address ports for input
    type d_inports, // data input ports
    type a_outports, // address ports for output
    type d_outports); // data output ports
    interface Vector#(a_inports, Put#(addr_type)) chai_port;
    interface Vector#(d_inports, Put#(Tuple2#(data_type,
        data_type))) chi_port;
    interface Vector#(a_outports, Put#(addr_type)) chao_port;
    interface Vector#(d_outports, Get#(Tuple2#(data_type,
        data_type))) cho_port;
    interface DDR2_ext_intf#(row_addr_type, data_type)
        ddr2_intf;
endinterface
```

Again, the *Vector* interface class is used, but now *put* and *get* functions are used to access the address and data FIFO's, which simplifies the interface. The *DDR2_ext_intf* contains the separate methods that correspond to the external memory signals (identical to the second part of the *DDR2_core* interface, see page 6). The main reason to add a level of hierarchy to the interface was that it allows the assignment of synthesis pragmas to this sub-interface.

BSV MODULES FOR THE PORT LEVEL

Most of the functionality is contained in the *scheduler* module. It contains the FIFO's and the functionality for switching between them. Furthermore, it should accommodate for an static schedule that can be specified at the top-level. As FIFO's are standard components for the Bluespec compiler they can be instantiated in a compact manner, e.g. in the case of the address FIFO's:

```
for (Integer i=0; i<valueof(chai_fifos); i=i+1)
begin
    FIFO#(Bit#(address_width)) chai_port_fifo <-
        mkSizedFIFO(div(chi_port_depth, burst_size));
    chai_fifo_vector[i] = chai_port_fifo;
end
for (Integer i=0; i<valueof(chao_fifos); i=i+1)
begin
    FIFO#(Bit#(address_width)) chao_port_fifo <-
        mkSizedFIFO(div(cho_port_depth, burst_size));
    chao_fifo_vector[i] = chao_port_fifo;
end
```

Those FIFO's can be made accessible to a higher level module by using transparent interface assignments, such as:

```
interface chai_fifo = chai_fifo_vector;  
interface chao_fifo = chao_fifo_vector;
```

The main control methods, *get_a*, *get_d*, *put_d*, *next* and *switch_datapath*, are implemented using registers that contains the number of the port that is selected and the methods select the appropriate FIFO's accordingly.

A vector of integers is passed to the top-level of the *scheduler* module, which indicates the order in which the ports are serviced. When the *next* method is called, the next item in the vector is serviced.

The *nport* module instantiates the *controller* and *scheduling* modules and contains a finite state machine that implements the main control for the *scheduler* module:

```
Stmt main_loop_stmt = seq  
  while (True) seq  
    if (sched_if.notEmpty) seq  
      action  
        Tuple2#(Bit#(1), Bit#(address_width)) t <-  
          sched_if.get_a();  
        t_r._write(t);  
      endaction  
      if (tpl_1(t_r._read) == 1'b0)  
        seq  
          while (!ddr2_ctrl_if.write_available)  
            noAction;  
          ddr2_ctrl_if.write(tpl_2(t_r));  
          sched_if.switch_datapath;  
          for (acc_loop <= 0; acc_loop <= fromInteger(  
            burst_size-1); acc_loop <= acc_loop+1) seq  
            action  
              Tuple2#(Bit#(data_width), Bit#(data_width))  
                d <- sched_if.get_d();  
              ddr2_ctrl_if.inport.put(d);  
            endaction  
          endseq  
          sched_if.next();  
        endseq  
      else  
        seq  
          while (!ddr2_ctrl_if.read_available)  
            noAction;  
          ddr2_ctrl_if.read(tpl_2(t_r));  
          noAction;  
          sched_if.switch_datapath;  
          sched_if.next();  
        endseq  
      endseq  
    endseq  
  endseq;  
endseq;
```

In the code shown above, *ddr2_ctrl_if* is the interface to the controller, whereas *sched_if* is the interface to the *scheduler* module.

The *nport_parametric* module is parametric and therefore cannot be synthesized. A fixed toplevel is necessary to make a synthesizable module.

Verification and Synthesis

This section describes the verification and synthesis of the multi-port memory controller.

VERIFICATION

As the Bluespec compiler generates an executable, a lot of verification can be done using Bluespec SystemVerilog. However, for a full system verification a behavioral and timing model for the external memory should be used. A Micron Verilog model for a 1Gbit DDR2 memory was used for this [Micron DDR2]. In Figure 5, the verification environment is shown.

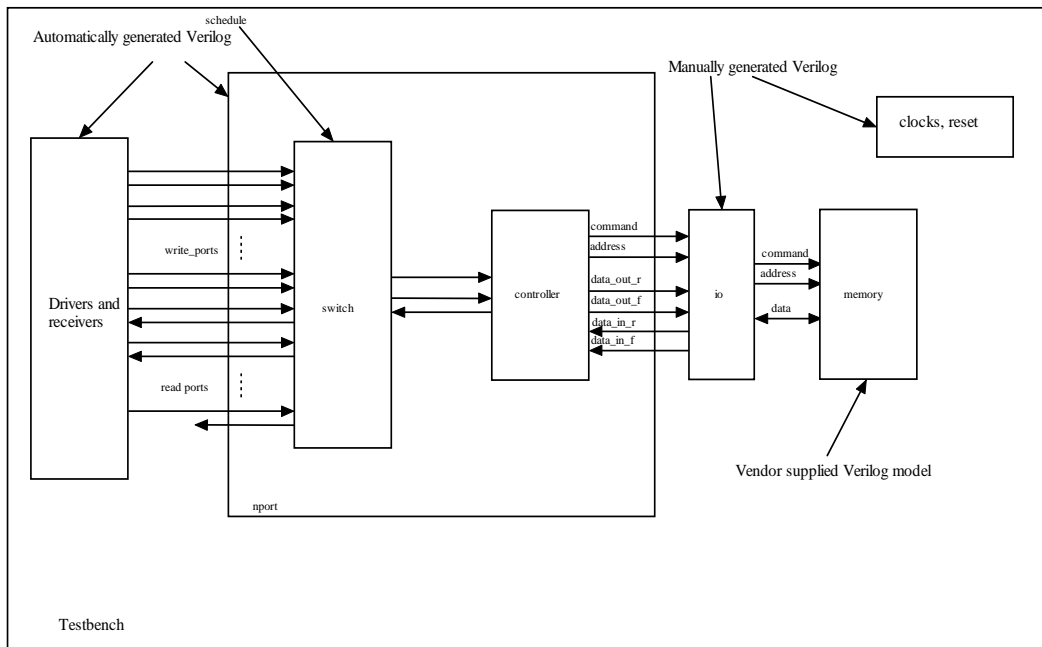


FIGURE 5. Verification environment

The Bluespec compiler generates standard Verilog code, which can be used without any problem in a Verilog based verification environment. A standard verification methodology, e.g. such as described in [Bergeron 2000] can thus be used.

SYNTHESIS

The design is synthesized in a TSMC 130 nm technology using Synopsys Design Compiler. A straightforward command file was used (see Appendix A) and the timing constraint for the main clock was varied for a clock frequency of 400 MHz to 1000 MHz. In Table the results are shown. A 1.3GHz Pentium III running RedHat 8 was used for this synthesis experiments.

TABLE 1. Synthesis results

Constraint	Resulting delay	Resulting area	Execution time
2.50 ns (400 MHz)	1.86 ns	441438.5	1010 s
1.67 ns (600 MHz)	1.56 ns	447134.8	1120 s
1.25 ns (800 MHz)	1.05 ns	480747.2	1226 s
1.0 ns (1000 MHz)	0.86 ns	490168.4	1258 s

As can be seen in Table 1, the timing requirement (to be able to run at a clock speed of 400MHz) was met without any problem. Furthermore it can be seen that the register transfer level Verilog generated by the Bluespec compiler can be optimized very well by Design Compiler, so even higher clock frequencies can be obtained at the expense of more area.

Comparisons with a multi-port memory controller written in RTL Verilog or VHDL was difficult, because the original design targeted an FPGA and especially the FIFO modules are different. However, preliminary experiments, replacing the FIFO's with FPGA specific FIFO's show that the area is within 10% of the original design.

Conclusions and recommendations

In this report the design of a multi-port memory controller was described for the purpose of evaluating the Bluespec SystemVerilog language and the Bluespec compiler. This design was chosen because it contains both low-level aspects and high-level aspects. Furthermore, a mixed verification environment was used to accommodate for the Verilog memory model.

The Bluespec SystemVerilog language proved effective for this design and the Bluespec compiler fitted very well in the verification and synthesis flow and the synthesis result exceeded the timing requirement. As claimed by Bluespec, the generated code was 100% guaranteed synthesizable RTL code and comparable in area and speed to hand coded RTL. The library containing complex structures, such as FIFOs, that increase productivity without compromising the RTL quality also proved a considerable benefit.

Rule and rule based interfaces semantics enforce a more formal check on the code, eliminating conflicts between shared resources. Therefore the verification does not have to check for data path multiplexing errors, reducing verification time.

The BSV simulator proved very fast, and it also has features not used in this evaluation, such as cosimulation with Verilog, VHDL and SystemC, making it very easy to integrate in any design flow.

One of the main advantages of Bluespec SystemVerilog is that the resulting design can be changed with less effort than a corresponding RTL design in Verilog or VHDL, resulting in re-use of components and a large portfolio of Intellectual Property that can be used in a “plug-and-play” like fashion. During the design it is not

References

necessary to commit to a fixed micro-architecture, it can easily be changed allowing the designer to experiment and choose the best possible micro-architecture.

The Bluespec methodology is quite different from the traditional RTL methodology, therefore the introduction training proved invaluable for this evaluation and also during the evaluation the support from the technical staff at Bluespec was very effective.

The complete code is available on request [Nport BSV].

References

[BSV User Guide] “Bluespec SystemVerilog User Guide, Revision: 22 July 2005”, Bluespec, Inc., 2005

[JESD79-2A] “JESD79-2A, DDR2 SDRAM SPECIFICATION”, JEDEC, *January 2004*

[Micron DDR2] <http://www.micron.com/products/dram/ddr2sdram/>

[Bergeron 2000] “Writing Testbenches”, Janick Bergeron, 2000

[Nport BSV] “Multi port DDR2 memory controller Bluespec SystemVerilog source code”, Gert-Jan Tromp, 2005, e-mail: G.J.Tromp@d-sync.com

Appendix A compile.tcl

```
set target_library [list tsmc13_fast.db ]
set synthetic_library [list standard.sldb ]
set link_library [concat "*" $target_library $synthetic_library ]

read_verilog SizedFIFO.v
read_verilog FIFO1.v
read_verilog FIFO2.v
read_verilog nport.v
#set_fpga_target_device 2V1000FF896

create_clock CLK -period 1.00

uniquify

compile

write -format verilog -hier -out nport_gt.v
report_timing
report_area
exit
```