A Methodology for Verifying Memory Access Protocols in Behavioral Synthesis

Gernot Koch† Taewhan Kim‡ Reiner Genevriere†

†Synopsys Inc.
700 E. Middlefield Rd.
Mountain View, CA 94043 USA

‡Dept. of Electrical Engineering & Computer Science
and Advanced Information Technology Research Center
KAIST, Taejon, 305-701 KOREA

Abstract— Memory is one of the most important components to be optimized in the several phases of the synthesis process. In behavioral synthesis, a memory is viewed as an abstract construct which hides the detail implementations of the memory. Consequently, for a vendor's memory, behavioral synthesis should create a clean model of the memory wrapper which abstracts the properties of the memory that are required to interface to the rest of the circuit. However, this wrapping process invariably demands the verification problem of the memory access protocols in order to be safely used in behavioral synthesis environment. In this paper, we propose a systematic methodology of verifying the correctness of the memory wrapper. Specifically, we analyze the complexity of the problem, and derive an effective solution which is not only practically efficient but also highly reliable. For designers who use memories as design components in behavioral synthesis, automating our solution shortens the verification time significantly in contrast of simulating memory accesses in the context of full design, which is a quite complex and time-consuming process, especially for designs with many memory access operations.

1 Introduction

As the design complexity is growing very rapidly, using behavioral synthesis[1, 2] starting from a high-level abstraction of design description is becoming inevitable for more and more design projects. Moreover, high-throughput on-chip memories are widely used in designing chips today. This trend turns out that one of the strongest arguments for behavioral synthesis is the ease of incorporating memories into a design.

Memory accesses are simply specified as accesses to an array variable in the HDL source code. The behavioral synthesis then takes care of the details of the memory access protocols and, if desired, even optimizes the order of memory accesses, exploiting possibilities for pipelined accesses or parallel accesses of multi-port memories. To enable the synthesis to infer memories for array variables, it requires an existence of an *HDL wrapper* (encapsulator) which instantiates the memory and takes care of the setup and hold time requirements of the memory. In addition, the synthesis requires separate information about the cycle-by-cycle protocol of this wrapper.

Until recently, this information and the wrapper had to be created manually by the designer[2]. Consequently, automating this process can save a considerable design effort in behavioral synthesis and increase the productivity further. The automation requires a wrapping utility which uses memory properties entered by the designer to generate memory wrappers to be used in behavioral synthesis. However, this memory wrapping invariablely induces the verification problem of the memory access protocols. That is, the correctness of everything the wrapping utility generates depends directly on the correctness of the properties entered by the designer. Earlier, with the hand-generated memory wrappers, users would create a small behavioral design solely to verify these wrappers. As memories are not visible in the pre-synthesis simulation, this design had to be synthesized and a post-synthesis simulation had to be performed to verify both the HDL wrapper and the protocol information. Since there are many possible configurations of parallel and inter-leaved memory accesses, e.g. for multi-port memories, creating a sample design and a testbench that perform a complete check rather than just a sanity check is very tedious, but very important. Fortunately, as the wrapping utility contains all the information it needs to generate a memory wrapper, it is in an ideal position to generate everything necessary to verify the wrapper as well.

In this paper, we propose a systematic approach to the problem of (1) generating a sample behavioral design, (2) a synthesis script constraining the design appropriately to push it through the synthesis and (3) a self-checking simulation testbench to verify the correctness of the memory wrapper. More specifically, we analyze the complexity of the problem of generating the sample design, script and testbench for a full coverage of verifying the memory access protocols in the memory wrapper, and propose an efficient and highly reliable solution to the problem.

2 Behavioral-level Memory Verification

2.1 Verification Structure

Testing the correctness of the memory wrapper in the context of behavioral synthesis requires a *sample behav-*

ioral design which must be designed carefully to fully test any combination of the pipeline/parallel memory accesses. Once a sample design is generated, a behavioral synthesis *script file* which controls the schedule of the memory operations in the design is needed. In addition, we are required to generate a self-checking *testbench file* to simulate the sample design before and after scheduling.

Figure 1 shows the design hierarchy for testing memory wrapper. The memory wrapper encapsulates the implementation of the memory. Behavioral synthesis uses the information in the memory wrapper for scheduling memory operations in the sample design. The correctness of the memory wrapper is verified by executing three tasks: (a) simulating the pre-scheduled behavioral-level sample design, (b) scheduling/allocating the sample design according to the schedule constraints, and (c) simulating the post-scheduled RTL design. Completing task (a) confirms the correct behavior of the sample design, and is not related to the memory wrapper itself. Task (b) indicates that the behavioral synthesis tool follows the protocols of the memory provided by the memory wrapper. Finally, task (c) shows that the memory wrapper in the RTL design produced by the behavioral synthesis tool functions correctly. Here, the confidence of the correctness of the memory wrapper is established by creating a sample design and scheduling the memory operations in the design in many different ways, which is the main subject of this paper. Note that the memory wrappers we are considering are always synchronous. This means that memory protocols with asynchronous or combinational behavior of memory operations are made to appear synchronous by the memory wrapper, e.g. by registering input signals in the wrapper.

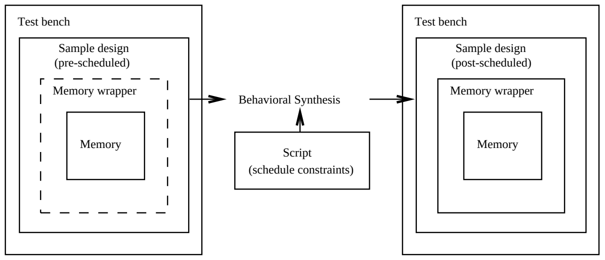

Figure 1: Design hierarchy of behavioral-level memory wrapper testing

2.2 Preliminaries

We first clarify some terms used in our presentation to avoid any confusion.

Logical port: A memory is connected to the outside world through *physical ports* (e.g., address port, data port, write enable port etc.). The throughput of the memory is characterized by the timing synchronization among the waveforms passed to the ports, which determines the *logical ports* of the memory, *i.e.,* read-write port, read-only port, and write-only port. With this context, we call the read and write memory operations *logical port operations* of the memory.

Regular pipeline: A pipeline with initiation interval i and latency l is *regular* if given an operation starting at cycle s another overlapping operation can only start to execute at cycles $s+i$, $s+2i$, $\cdots$, $s+k\cdot i$ where k is an integer such that $k\cdot i \leq l < (k+1)\cdot i$.

Irregular pipeline: A pipeline which is not a regular pipeline, but still allows overlapped execution of operations, is *irregular*.

Conflict flag and Conflict vector: We define a *conflict flag*, $C_{x,y}(i)$, to denote the feasibility of pipelining memory operation y to memory operation x with initiation interval i. $C_{x,y}(i)$ becomes 1 if such a pipeline causes a resource contention, and becomes 0 otherwise. We collectively represent $C_{x,y}(i)$, i = 1, 2, $\cdots$, $l-1$ where l is the latency of x as a vector form, called *conflict vector*, $CV_{x,y}$ = $[C_{x,y}(1), \cdots, C_{x,y}(l-1)]$.

A memory operation can have different cycle-by-cycle input/output connections and resource requirements of the overlapping executions among operations. We can represent the cycle-by-cycle connections of a memory operation using a concept similar to behavioral templates in [3]. For example, Figure 2(b) shows modeling of 3-cycle memory write operation from the RT-level timing relations of the signals for the write operation shown in Figure 2(a). The address, data and write enable inputs are de-coupled in terms of when and how long each input must be stable. We then extract the feasibility of overlapping execution of memory operations from the template modeling of the cycle-by-cycle connections. For example, 2 memory write operations on the left hand of Figure 2(c) cannot be pipelined with initiation interval 1 on the same logical port because the *addr* connection conflicts, but pipelining with initiation interval 2 is possible as shown on the right hand of Figure 2(c). In summary, from Figure 2(c) $C_{x,y}(1)=1$ and $C_{x,y}(2)=0$. That is, $CV_{x,y}$ = [0, 1].

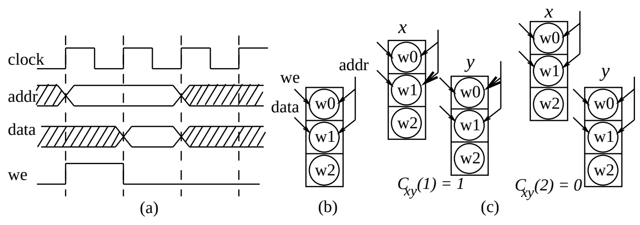

Figure 2: (a) An RT-level timing relation of signals for 3-cycle memory write; (b) The cycle-by-cycle template modeling of (a); (c) Derivation of conflict flags for two memory write operations x and y

Since the verification of memory wrapper is tightly related to the type of the memory and how the behavioral synthesis tool models the memory, we first introduce a set of assumptions that are minimally required by our verification solution:

(a) Memory has at least one port supporting read access, and also has at least one port supporting write access. That is, we do not perform any self-checking for the memory wrappers of read-only memories and write-only memories.

(b) All read accesses of a memory have the same protocol. This applies to all write accesses as well. This assumption simplifies the computation of conflict vectors. We only need to determine the four conflict vectors, $C_{R,R}$, $C_{W,W}$, $C_{R,W}$, and $C_{W,R}$ where R and W represent *any* read and write memory operations. We extract the conflict vectors of the memory wrapper from the waveforms (protocols) of the memory operations.

(c) The designer has no control of binding a memory operation to a particular port of memory. The task of memory port binding is automatically done by the synthesis tool in a way to optimize the overall design.

(d) The synthesis tool can constrain scheduling of memory operations relative to the other operations. Specifically, when we denote sch(x) to the cycle step at which operation x starts the execution, the scheduler supports a schedule-constraining command, fix_cyc(c, x, y), which constrains sch(y) - sch(x) = c.

The memory information that the memory wrapper contains is used to link the memory cell to the data path and to schedule memory operations. The correctness of the information can, in fact, be verified by simulating the scheduled RTL design of the sample behavioral design that is created by our methodology of memory wrapper testing. The memory information necessary for testing in behavioral synthesis is (1) bit-widths of address and data busses, (2) number of each logical port (i.e., read-only, write-only, read-write), (3) control pins used for each logical port operation (read, write), and (4) latency and required control pin settings for each logical port operation.

3 Complexity Analysis

We could write any sample behavioral design which contains memory operations, schedule it and simulate the scheduled RTL design together with the vendor provided simulation model of the used memory. Our goal is, however, to achieve a highly reliable memory wrapper testing systematically. To do this, we could create a sample design with scheduling constraints and a simulation testbench which covers all the possible pipelines and parallelisms of memory operations. However, this would require an excessive amount of design effort and run time in scheduling.

For example, consider a memory with n read/write (logical) ports. When we assume that the latency of each of read and write operations for every logical port is l and a pair of any two operations can be pipelined with any value of initiation interval, the number of possible combinations of pipelines executed over the n logical ports is $2^{l-1} \cdot 2^n$ since there are up to 2^{l-1} different patterns of pipelines using any of read and write operations on a single logical port and there are n logical ports. Thus, the total number of memory operations in a sample design for testing all the pipelines becomes $2^{(l-1)+n} \cdot l \cdot n$ since up to l memory operations are involved in a pipeline on a logical port. We might also take into account the number of possible combinations of parallel accesses among different (logical) ports, which is 2^n. However, this has already been counted when the combinations of pipelines were considered. The solid curve in Figure 3 shows the change of the number of memory operations required with the change of the values of the latency, l, and the number of (logical) ports, n. For example, to fully test a memory with 2 read/write ports and latency 3, a total of 96 memory operations are needed. In contrast, the dotted curve in Figure 3 shows the number of memory operations required by our testing methodology.[1] It reduces the testing complexity significantly, but achieves a highly reliable and systematic testing. The details of our proposed testing methodology are described in Sec. 4.

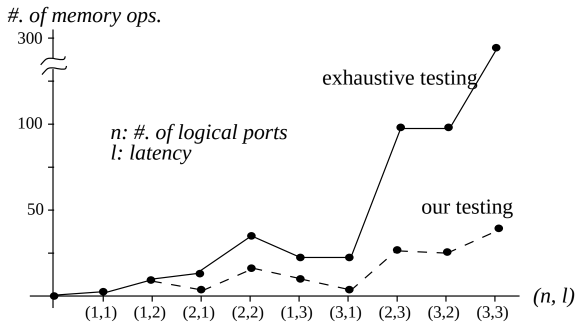

Figure 3: The complexity of memory wrapper testing in terms of the number of operations

4 Behavioral-level Memory Verification

The scope of our memory wrapper verification is purely functional. We generate a sample behavioral design which contains memory accesses, schedule it and save the scheduled RTL design in VHDL or Verilog. This makes it technology independent and allows for a more general testing.[2]

We accomplish the verification by carrying out two classes of memory operations, namely, checking for *intra-port accesses*, i.e., pipelined memory access, and checking for *inter-port accesses*, i.e., parallel memory accesses.

[1]The number of operations is expressed as $4n \cdot l + n$, $n > 1$ and $l > 1$ where the first and second terms correspond to that in checkings in Sec. 4.1 and that in Sec. 4.2, respectively.

[2]While the sample design may also be used as a starting point for static or dynamic timing verification, we exclude this from the scope of this paper.

These checks ensure that the protocols of logical port operations specified in the memory wrapper are correct and the memory can be operated with the maximum throughput according to the number and type of logical ports, the latencies and pipelines. (Each of the checks described in the following is assumed to start at cycle 0. This is to be understood as being relative to the start of the particular check. As these checks are performed one by one sequentially, the real cycle number at which a certain check starts depends on the previously performed checks.)

4.1 Intra-port checkings

For each logical port of memory, every pipeline (read-read, write-write, read-write, write-read) is filled so as to achieve a maximal throughput. Since by the third assumption in Sec. 2.2, there is no way to constrain the synthesis tool to use a certain logical port for a certain memory operation, for each memory operation all the ports that support this operation have to be accessed at the same time to ensure that the pipeline of each port is actually filled. Our testing consists of four types of pipeline.

(1) Read-Read pipeline: Given a memory with n logical ports supporting read operations, each with a regular pipeline with initiation interval i and latency l, the schedule of read operations of the pipeline is shown in Figure 4 where k is an integer such that $k \cdot i < l \leq (k+1) \cdot i$. This corresponds to the case that $C_{R,R}(i) = C_{R,R}(2i) = \cdots = C_{R,R}(ki) = 0$ and $C_{R,R}(j)$ = 1, $j \neq i, 2i, \cdots, ki$.

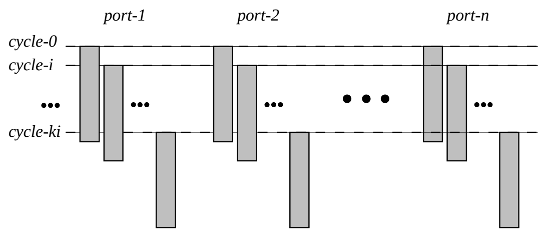

Figure 4: Expected schedule for testing a regular pipeline of read operations

To illustrate when and how an irregular read-read pipeline is tested, let us consider a memory whose read latency is 5 with its condition vector $CV_{R,R}$ = [0, 0, 1, 0]. To effectively test the pipeline at each logical read port we attempt to maximize the overlapping of the executions among the memory operations. Consequently, in Figure 5(a), operation $r2$ is pipelined to operation $r1$ with initiation interval 1 because there is no access conflict between them (i.e., $C_{r1,r2}(1)$ = 0), resulting in the pipeline in Figure 5(b) which is then overlapping with operation $r3$ with initiation interval 1 since $C_{r1,r3}(2)$ = $C_{r2,r3}(1)$ = 0 as shown in Figure 5(c). However, overlapping $r4$ with $r3$ with initiation interval 1 is not possible because there is an access conflict by $C_{r1,r4}(3)$) = 1 as indicated by a block box in Figure 5(d). Furthermore, $r4$ cannot overlap with $r3$ with initiation interval 2 as shown Figure 5(e) since $C_{r2,r4}(3)$ = 1. Consequently, the pipeline we have found so far is the one in Figure 5(c).

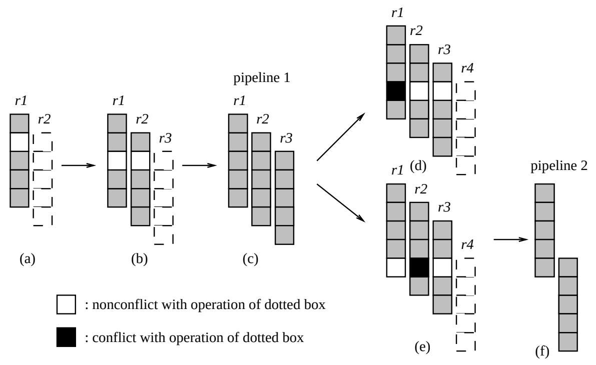
r1
r2
r3
r4
pipeline 1
pipeline 2
(a)
(b)
(c)
(d)
(e)
(f)
: nonconflict with operation of dotted box
: conflict with operation of dotted box

Figure 5: An example of showing the generation of pipelines; (a) check pipelining with $r2$; (b) check pipelining with $r3$; (c) pipeline with three operations; (d) access conflicting when pipelining $r4$ with initiation interval 1; (e) access conflicting when pipelining $r4$ with initiation interval 2; (f) another pipeline derived from (e)

To explore another possibility of pipelining which is totally different from the previously generated one, we utilize the conflict information produced before. Specifically, if we examine the failure of overlapping $r4$ with $r1$ in Figure 5(e), it is due to the conflict with $r2$, not because of that with $r1$. This means that we can start to create a new pipeline with $r1$ and $r4$ such that $r4$ is overlapped with $r1$ with initiation interval 4 as shown in Figure 5(f). If there were more than one cycle step at which such conflicting occurs, a new pipeline, for each conflicting cycle step, is generated with the initiation interval starting with the cycle step. In the following, we summarize the idea of generating pipeline designs with a constructive procedure, which works for irregular pipelines as well as regular ones. It is recursive and starts by calling Gen_RR_Pipelines(1, 0).

Gen_RR_Pipelines(i, K): *Generate read-read pipelines*

```
(1)  • Set $\mathcal{M}$ = {}; /* conflicting operations */
(2)  • Set $\mathcal{S}$ = {}; /* scheduled operations */
(3)  • Schedule an operation (say $r_K$) to cycle step $K$;
(4)  For each $k = K + i, \cdots$, and $K + i + l - 1$ {
(5)     if ($C_{r_K, r_k}(k - K)$=1) continue; /* conflict with $r_K$ */
(6)     else if (there is an $r_j$ in $\mathcal{S}$ such that $C_{r_j, r_k}(k - j)$=1) {
(7)        /* conflict with some of the remaining scheduled ops */
(8)           • $\mathcal{M}$ = $\mathcal{M}$ + {$r_j$};
(9)           • continue;
(10)    } else { /* non-conflicting */
(11)          • Schedule $r_k$ to cycle $k$;
(12)          • $\mathcal{S}$ = $\mathcal{S}$ + {$r_j$};
(13)          • last_cstep = $k$;
(14)    }
(15)  } /* endfor */
(16)  /* Generate new pipelines recursively */
(17)  • start_cstep = $K$+last_cstep+l; /* $D$: defined in Sec.4.3 */
(18)  • Gen_RR_Pipelines($k$, start_cstep);
```

Note that each of the pipelines generated is replicated to the number of read and read-write logical ports of the memory, which are then constrained to be executed in *parallel* to ensure that the scheduler of behavioral synthesis produces these pipelines. For example, in Figure 5 two types of pipeline were extracted to test. Suppose that the corresponding memory has 1 read/write port and 1 read-only port. Then, each type of pipeline is duplicated to be performed in parallel, one on the read-write port, the other on the read-only port.

(2) Write-Write pipeline: This pipelines are treated exactly the same as that of Read-Read pipelines except that all the read operations used in Gen_RR_Pipelines are replaced with write operations. The conflict vectors and latency are changed accordingly. Each type of pipelines produced from Gen_WW_Pipelines(1, 0) is replicated to the number of read/write and write-only logical ports of the memory, and scheduled them in parallel.

(3) Read-Write pipeline: This is to verify that the memory indeed supports overlapping read and write operations on the same read-write logical port correctly as was specified in the memory wrapper. To do this, a read operation is scheduled at cycle 0, and at each cycle k (= 1, $\cdots$, $l-1$), a write operation is started unless there is a conflict. A Read-Write pipeline generation routine, Gen_RW_Pipelines, is constructed from Gen_RR_Pipelines by modifying that all read operations except operation r_K must be replaced with write operations. Specifically, the conflict checking in line 5 is replaced with $C_{r_K,w_k}(k-K) = 1$ since it is to check that a write operation can be pipelined to a read operation with initiation interval ($k-K$), and also the checking in line 6 is replaced with $C_{w_j,w_k}(k-j) = 1$ since it is to check that a write operation can be pipelined with another write operation with initiation interval ($k-j$).

For each type of pipeline obtained, its read operation is replicated to the number of ports supporting read operation, and they are scheduled in parallel. In the same way, each of the write operation is replicated to the number of ports supporting write operation, and they are scheduled in parallel. This ensures that the scheduler indeed generates the desired pipeline.

(4) Write-Read pipeline: This check is performed according to the Read-Write check with the exchanged roles for read and write operations.

4.2 Inter-port checkings

The inter-port check verifies the integrity of the memory model specified in the memory wrapper in the case of parallel execution of operations on different ports. Due to the nature of the intra-port checks, the cases of parallel read operations and parallel write operations have already been taken care of. Not yet taken care of is only the execution of write and read operations scheduled at the same cycle.

Because of the assumption of the inability of the behavioral synthesis to assign an operation to a certain port, intra-port check cannot be performed selectively for each pair of ports. Therefore, only one check is performed where one read operation and one write operation are scheduled at the same cycle. To do this, for a memory with n_r read-only ports, n_w write-only ports and n_{rw} read-write ports, we schedule (n_r+m) read operations and (n_w+n_{rw}-m) write operations where $m = \lceil \frac{n_{rw}}{2} \rceil$ at the same cycle. This causes a maximum parallelism of a mixture of read and write operations and the scheduler will map them onto different ports. The result of this check is assumed to be projectable to each pair of memory ports.

4.3 Pre- and Post-Processing

Tests of the intra-port and inter-port accesses are performed sequentially, one for each type of pipelines and parallel accesses. For each test, we are required to have pre- and post-processing steps to make it work.

Pre-processing: For each test, the sample design must read all the values from a testbench to be used in the memory write operations. Reading these values is guarded with a control signal telling the testbench when it should write values. Then, all the memory addresses used in the read operations must be initialized prior to the test. This initialization can already be part of the checks, e.g., during Write/Write pipeline checking.

Post-processing: After executing the main test of pipelining or parallel accesses, all the addresses used in the write operations must be retrived with memory read operations. A part of these read operations can be executed during the checks already. After the values have been retrived from the memory, they must be written back to the testbench, which is also guarded by a control signal.

All the operations in the pre- and post-processings are constrained to be scheduled so that one operation starts to execute only when the execution of the previous operation is completed. The value of schedule interval D in line 17 of Gen_RR_Pipelines() between the schedules of two pipelines is the sum of the number of cycles required by the operations in the pre-processing of the proceeding pipeline and the number of cycles by the operations in the post-processing of the succeeding pipeline.

4.4 Test Example

We have implemented a utility program, called *MemWrap*, for generating a memory wrapper and testing the wrapper in the environment of the Synopsys Behavioral Compiler [4]. From the memory information provided by the user, *MemWrap* creates a memory wrapper to be used by the Behavioral Compiler. To validate the correctness of the memory wrapper, *MemWrap* also generates a sample design with scheduling constraints and its testbench automatically according to our proposed testing methodology. The sample design and testbench are written in either VHDL or Verilog.

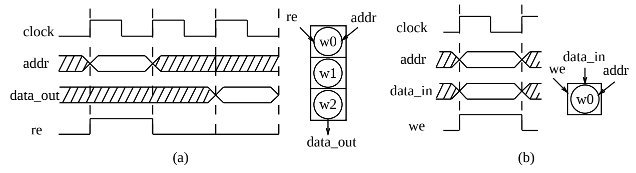

Figure 6: RT-level access timing relations of memory mem_test1 and template modelings

Consider testing memory mem_test1 with two read-write ports. The RT-level timing relations of the read and write access operations and the corresponding template modelings are shown in Figures 6(a) and (b). From the access templates, we derive the conflict vectors. That is, $CV_{R,R} = CV_{R,W} =$ [0, 0], and $CV_{W,W} = CV_{W,R}$ = []. Consequently, we derive read-read and read-write pipelines, and a parallel accesses of 1 read and 1 write. The read-read pipeline of test sample design written in Verilog is shown below.

```
forever begin : test_loop
    //// Design for read-read pipeline with initiation interval 1
    // (i) Get values for testing the pipeline
    ready <= 1;
    dat0 = data_in; // line_label rr_in0
    @(posedge clk); ready <= 0;
    dat1 = data_in; // line_label rr_in1
    @(posedge clk);
    ....
    ready <= 1;
    dat4 = data_in; // line_label rr_in4
    @(posedge clk); ready <= 0;
    dat5 = data_in; // line_label rr_in5
    // (ii) Initialize memory for testing
    cell[0] = dat0; // line_label rr_write_in0;
    ....
    cell[5] = dat5; // line_label rr_write_in5;
    // (iii) Testing read-read pipeline
    out0 = cell[0]; // line_label rr_read0
    ....
    out1 = cell[5]; // line_label rr_read5
    // (iv) Send out values for comparison
    done <= 1;
    data_out <= out0; // line_label rr_out0
    @(posedge clk); done <= 0;
    data_out <= out1; // line_label rr_out1
    @(posedge clk);
    ....
    done <= 1;
    data_out <= out4; // line_label rr_out5
    @(posedge clk); done <= 0;
    data_out <= out5; // line_label rr_out5
    //// Design for read-write pipeline with initiation interval 1
    ....
end // test_loop
```

It consists of four parts. All labeled statements in the parts except step (iii) are executed sequentially. To do this, fix_cyc scheduling constraint is used. The synthesis orders the executions of the operations as constrained[5]. The pipelined memory accesses in step (iii) are generated by constraining as

```
fix_cyc(0, rr_read0, rr_read1); fix_cyc(0, rr_read2, rr_read3);
fix_cyc(0, rr_read4, rr_read5); fix_cyc(1, rr_read0, rr_read2);
fix_cyc(2, rr_read0, rr_read4);
```

where the constraints of the first line ensure that for each pair of operations rr_read0 and rr_read1, rr_read2 and rr_read3, and rr_read4 and rr_read5, the two operations start to execute on different ports at the same time, and the constraints of the second line ensure that rr_read2 is pipelined to rr_read0 with initiation interval 1, and rr_read4 is pipelined to rr_read0 with initiation interval 2.

5 Conclusions

In this paper, we have proposed a systematic approach to the problem of verifying the memory access protocols in behavioral synthesis. As the design complexity is growing very rapidly, using behavioral synthesis as the first step of the synthesis process is becoming increasingly important to meet the productivity needs, and further, high-performance on-chip memories are widely used today in industry. In that context, we proposed a highly reliable as well as practically efficient solution to the verification of the correctness of the memory access protocols in behavioral synthesis.

Acknowledgment: This work was supported by Synopsys Inc., and the work of Taewhan Kim was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

References

[1] D. Gajski, N. Dutt, A. Wu, S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publisher, 1992.

[2] D. W. Knapp, *Behavioral Synthesis*, Prentice Hall, 1996.

[3] T. Ly, D. Knapp, R. Miller, D. MacMillen, “Scheduling using Behavioral Templates,” *DAC*, 1995.

[4] *Behavioral Compiler User Guide*, Synopsys Inc., 1998.

[5] D. Knapp, T. Ly, D. MacMillen, R. Miller, “Behavioral Synthesis Methodology for HDL-Based Specification and Validation,” *DAC*, 1995.