Cycle-accurate RTL Modeling with Multi-Cycled and Pipelined Components

Rainer Dömer, Andreas Gerstlauer, Dongwan Shin
{doemer,gerstl,dongwans}@cecs.uci.edu
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract — Despite extensive research efforts for a number of years, modeling of RTL designs has still not reached a satisfactory state. Behavioral RTL design models still lack cycle-accuracy when multi-cycle and/or pipelined components are used. With such components, cycle-accuracy is only reached at the end of the RTL design flow when a complex structural netlist is obtained. Observation, debugging and modification efforts, however, are very tedious and difficult in such a model due to its complexity. This paper provides a simple yet powerful solution to this problem. An easy-to-understand RTL model is proposed that supports clock-cycle accuracy in a behavioral description even in the presence of multi-cycled and/or pipelined components. Experiments show the effectiveness of the approach for specification, simulation, and synthesis.

1 Introduction

While more and more research recently already focuses at levels of abstraction above the algorithm and register-transfer level (RTL), it is still RTL where most designs are specified today. While the benefits of entering designs at higher abstraction levels are clear and well-understood, designers still have trouble accepting this and putting it to use for their real-world designs.

The two main arguments used by designers are the following: First, the performance of automatically synthesized designs is typically lower than for hand-optimized implementations, and second, designers are loosing control over the decisions made during the automated synthesis process.

The authors of this work believe that these two issues are actually related to each other. More specifically, the loss of control over the implementation decisions is actually the main reason for the lower performance of an automatically generated design. In other words, if the designers would be able to control all the critical design decisions in the synthesis process, then the resulting implementation would reach the same quality level as a manually optimized design.

Hence, the *controllability* of the automated design process is of critical importance, and, if achieved, this will enable the move to higher levels of abstraction with significantly higher productivity and efficiency. This paper addresses the modeling of RTL designs at the behavioral level. While many details of the design implementation are intended to be abstracted away, critical aspects that will significantly affect the performance and quality of the result should be able to be specified explicitly. In other words, as soon as a design decision is made, it should be reflected in the design model such that it can be evaluated, for example, through simulation of the model.

The core tasks of behavioral synthesis are performed, namely scheduling, allocation and binding. During these tasks, a finite state machine with data (FSMD) model is typically used to represent the design. The behavioral description of the design are now scheduled into actual clock-cycles and represented by separate states. Thus, the model itself is now clock-cycle accurate and will exhibit the actual timing of the design when simulated. However, complex components at the RTL are functional units whose execution time is longer than one clock cycle. Specifically, the use of multi-cycle and pipelined units cannot accurately be described. The components that have a delay longer than a single clock cycle or are pipelined, pose a real problem as their behavior cannot be expressed in any current hardware description languages (HDL) such as VHDL [6], Verilog [7], and SystemC [5]. The reason for this problem is simply that there is no construct available in these languages that describes a function being executed over a duration of multiple cycles.

The rest of this paper is organized as follows: Section 2 shows related work and Section 3 introduces the concept of *delayed assignment statements* and how these can be used to simulate the behavior of complex components in a cycle-accurate manner. For compilation and synthesis, an algorithm is given in Section 4 which automatically inserts delayed assignments into a model. Experimental results are listed in Section 5, and Section 6 finally concludes this paper.

2 Related work

Issues in RTL modeling, RTL design and behavioral synthesis, aka. High-Level Synthesis (HLS), have been studied for more than a decade now [4],[10]. Countless research results have been published. Due to space limitations, however, only a few approaches can be mentioned here.

Many automatic synthesis tools (also known as *push-button* synthesis) have been developed, including Synopsys Behavioral Compiler [13], Cyber [14], and OSCAR [11]. However, these tools provide no means to access the intermediate de-

sign models that are created during the synthesis process. The only models accessible to the designer are the behavioral input model and the structural output model.

On the other hand, interactive synthesis tools including Amical [8] and ISE [9] allow the designer to inspect and manipulate the design model at different stages in the synthesis process, typically via a graphical user interface. However, a cycle-accurate simulation model with complex components is not available for the intermediate stages.

Recently, a new interactive high-level synthesis system has been developed [12], that is based on the Accellera RTL standard [1]. Accellera modeling semantics define simulatable and synthesizable intermediate models at different stages in the synthesis process. Description of partial design decisions is possible as well. However, multi-cycle and pipelined functional units are not supported.

HY-C [15] models hardware behavior as parametric time-interval extended FSMs. Cycle-accurate, cycle-free (untimed) and mixed models are supported, but require in-depth understanding of the underlying complex timing formalism.

3 Cycle-accurate Behavioral RTL

In this section, we will solve the problem of describing the use of multi-cycle and pipelined components *cycle-accurately* in a behavioral RTL model. What we need is basically a new construct that computes a function (i.e. the behavior of the component) over a period of time. Specifically, for multi-cycle components, control and arguments need to be supplied over a sequence of clock cycles, only then the result can be read at the end. Pipelined components are similar, except that control and arguments only need to be supplied in the very first cycle since they will be stored internally in the component over the execution of the pipeline.

Essentially, we want to supply arguments as required by the component, compute the function (only once!), and then obtain the result after the specified period of time for the component. The idea is to use *delayed assignments* for this purpose.

3.1 Delayed assignment statements

A delayed assignment statement is an assignment that takes place only after a specified number of clock cycles.

The specific semantics are defined as follows: The right-hand side (RHS) of the assignment statement (i.e. the function with its arguments) is evaluated in the same clock cycle the delayed assignment is specified. However, the left-hand side (LHS), the target of the assignment, is evaluated only after the specified number of cycles. At the same time, the actual assignment of the result to the target then takes place.

Syntactically, a delayed assignment is specified by use of a keyword (**after** or **piped** clause, see below), and a positive integer indicating the delay in terms of number of clock cycles.

3.2 Multi-cycle components

Fig. 1(a) shows an example of modeling multi-cycled components by use of **after** clauses. In state `s1`, a multiplier starts computing the product of `RF[0]` and `RF[1]`. Since the multiplier needs two cycles for this computation, as specified by the **after** clause, the result will only be available in the target register `RF[0]` after two cycles, i.e. in state `s3`. It is an error to read the result from register `RF[0]` earlier.

Figure 1: Modeling multi-cycle components using **after** clauses: (a) specification, (b) simulation.

One possible way to implement this delayed assignment in a simulator is shown in Fig. 1(b). The result of the multiplication is simply stored in a temporary variable `RF_0_tmp` in state `s1`, and then assigned to the target in state `s3`.

3.3 Pipelined components

Pipelined components can be handled in a very similar way. Fig. 2(a) shows the use of a 3-stage pipelined multiplier for the two multiplications starting in state `s1` and `s2`. The multiplier is assumed to have two internal registers as indicated by the **piped** clauses. Thus, the multiplication results are available only in states `s3` and `s4`, respectively.

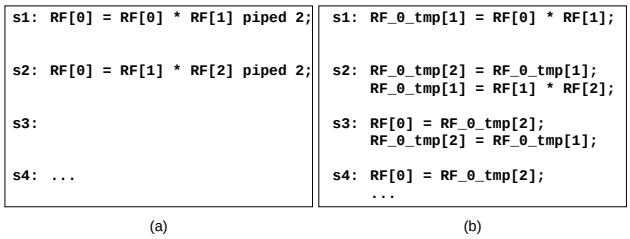

Figure 2: Modeling pipelined components using **piped** clauses: (a) specification, (b) simulation.

Note that for simulation of **piped** clauses an array of temporary variables is required, as shown in Fig. 2(b). Then, to mimic the pipeline behavior, the contents of the array are shifted in pipeline fashion, as shown.

It should be emphasized that the use of such temporary variables is not only simple, it is also very efficient. In the presence of multiple delayed assignments in the design model, the temporary variables can be easily shared. It is the number of different targets that determines the number of temporary variables (not the number of delayed assignments!). In other words, only one temporary variable is needed for every target register.

4 Compilation and Synthesis

While the examples discussed in the previous section seem to be straightforward, things become significantly more complex in real design models. For instance, taking conditions and loops into account, the control flow in a FSMD may be arbitrary. This requires to insert temporary assignments on every

possible path in the FSMD. As we will see in Section 4.2, this not only can lead to duplication of temporary assignments, but also to conflicting assignments to the same target.

Another fact ignored so far, is that delayed assignments may be conditional. In this case, the delayed statement must only be executed if the specified condition is true. As a result, the same specified condition must be applied also to all temporary assignments.

These and other problems need to be solved when delayed assignments are processed by compilers (for simulation), as well as by synthesis tools (for actual implementation). The following section addresses these issues by providing an efficient algorithm that automates the handling of **piped** and **after** clauses.

4.1 Automation of delayed assignments

Fig. 3 shows the pseudo code for an algorithm that can be used in a compiler to automatically insert temporary variables and corresponding assignments for delayed assignment statements in the model. With minor modifications, the same algorithm can also be used in synthesis tools to create the correct control words in every state.

```
algorithm CompileDelayedAssignments(fsmd):

foreach s in States(fsmd) do
  foreach d in DelayedAssignments(s) do
    c = Condition(d)
    n = Cycles(d)
    if Type(d)='after' then
      v = NewTmpVar(fsmd,lhs(d),1)
      foreach ns in NextStates(s,n) do
        AddTmpAssignment(ns,c,lhs(d),v,1)
    else
      v = NewTmpVar(fsmd,lhs(d),n)
      for i=1 to n-1 do
        foreach ns in NextStates(s,i) do
          AddTmpShift(ns,c,v,i)
      foreach ns in NextStates(s,n) do
        AddTmpAssignment(ns,c,lhs(d),v,n)
    Replace(d,v,rhs(d))
```

Figure 3: Algorithm for compilation of delayed assignments.

The algorithm essentially traverses all states of the given FSMD and replaces any delayed assignment statements with a set of temporary variable assignments that are inserted into the following next states.

- $States(fsmd)$ returns a list of the states in the $fsmd$
- $DelayedAssignments(s)$ returns a list of the delayed assignment statements in a state s
- $Condition(d)$ computes the condition under which a statement d is executed
- $Cycles(d)$ returns the number of cycles a statement d is to be delayed
- $Type(d)$ returns the type of a delayed assignment d, i.e. **piped** or **after**
- $NewTmpVar(fsmd, lhs, s)$ creates a new temporary variable in the $fsmd$ corresponding to the target lhs with array size s; note that $s = 1$ in the case of **after**; if the variable for lhs already exists, its $size$ is set to $max(size, s)$; this enables the sharing of temporary variables as discussed earlier
- $NextStates(s, n)$ returns the list of next states reachable from state s within n transitions
- $AddTmpAssignments(s, c, lhs, v, i)$ adds an assignment to state s under condition c; lhs is the target of the assignment; $v[i]$ is the source
- $AddTmpShift(s, c, v, i)$ adds a shift statement to state s where $v[i + 1]$ is set to $v[i]$
- $Replace(d, v, rhs)$ replaces the delayed assignment d with an assignment $v[1] = rhs$

It should be noted that the function $NextStates(s, n)$ is usually part of the static reachability analysis that every compiler or synthesizer performs in order to detect non-reachable states and to optimize the state transitions in the FSMD. As such, it is not further outlined in this paper.

Since the algorithm visits every state and from there possibly all next states, the complexity grows linear with the number of state transitions, or quadratic in terms of the number of states. Thus, the complexity is $O(n^2)$ where n is the number of states.

4.2 Conflicting delayed assignments

As mentioned earlier, there exists a possibility that conflicting assignments are being created by the algorithm. A conflict occurs if and only if multiple assignments to the same target variable exist in the same state under the same condition. This, of course, indicates a real problem since no register can be loaded with values from multiple sources at the same time.

Careful analysis of the situations, which can lead to such a condition, shows that in all cases an actual resource conflict has been specified in the model. For example, a multi-cycle component is used for different operations in the same cycle, or a pipelined component writes to the same target register as another component.

Fortunately, such resource conflicts can be easily detected by the compiler and synthesizer, and can then be reported to the designer as an error condition.

In fact, this checking can be implemented in the functions $AddTmpAssignments()$ and $AddTmpShift()$ in the algorithm shown in Fig. 3. Before adding the requested assignment statement, the two functions check if an assignment to the same target already exists (in the same state, under the same condition). If not, the functions can go ahead and do their work.

If an assignment to the same target variable is already present, the functions will check if the source is the same as given in their arguments. This case happens naturally if multi-cycle and pipelined components are used within loops, and, of course, is perfectly ok. If the sources are different, however, then an actual resource conflict exists and needs to be reported to the designer as an error message.

5 Experiments and Results

The technique and the algorithm described in this paper have been implemented in a compiler and a synthesizer [12] as an extension to the SpecC language [3].

In order to demonstrate the effectiveness of this approach, the code-book search algorithm specified in the voice encoder

ALU_16 ALU_32	1 cycle 1 cycle	1 cycle 2 cycles	2 cycles 2 cycles	1 stages 2 stages	2 stages 2 stages	2 stages 2 cycles	3 stages 3 cycles	4 stages 4 cycles
States / cycles:								
`cor_h_x`	27 / 3843	33 / 4793	37 / 4876	33 / 4793	36 / 4835	36 / 4835	46 / 5833	56 / 6831
`set_sign`	53 / 902	59 / 1103	73 / 1334	59 / 1103	70 / 1246	70 / 1246	88 / 1591	107 / 1976
`cor_h`	56 / 11152	58 / 9674	70 / 12140	58 / 9674	69 / 12099	69 / 12099	84 / 13126	101 / 15713
`search_10i40`	312 / 18577	355 / 23425	434 / 27577	335 / 21021	365 / 24669	386 / 27105	448 / 29689	536 / 36689
`build_code`	111 / 1725	123 / 1940	131 / 1987	123 / 1940	130 / 1985	130 / 1985	149 / 2245	168 / 2505
`q_p`	10 / 59	10 / 59	10 / 59	10 / 59	10 / 59	10 / 59	10 / 59	10 / 59
`code_10i40`	569 / 36258	638 / 40994	755 / 47973	618 / 38590	680 / 44893	701 / 47329	825 / 52543	978 / 63773
piped clauses	0	0	0	104	279	175	175	175
after clauses	0	104	279	0	0	104	104	104
Tmp. variables	0	39	61	36	57	71	70	80
Tmp. assignments	0	208	558	208	558	558	654	717
Simulation time (s)	106.96	111.40	131.48	105.45	120.81	132.87	136.10	159.88

Figure 4: Experimental results for code-book search example using pipelined and multi-cycled ALUs.

of the GSM standard for telecommunication has been chosen as design example. The code-book search algorithm consists of six filter functions which operate on sub-frames of speech data, each of which consists of 40 samples. While the complexity of these filter functions varies widely, all of them use saturated arithmetic operations with 16 and 32 bit results.

For our experiments, we have chosen different allocations of ALUs, varying in delay and number of pipeline stages. Fig. 4. list the experimental results for different multi-cycle ALUs, different pipelined ALUs, and different combinations of multi-cycle and pipelined ALUs.

For each experiment, synthesis and simulation have successfully been performed. The resulting number of states and executed clock-cycles are listed for each filter function. The tables also list the number of **after** and **piped** clauses used in the model, as well as the number of temporary variables and temporary assignments in the simulation model. Note that the latter two are large numbers in many cases, which emphasizes the benefit of automatic insertion.

The simulation times in the tables are measured over a total of 652 sub-frames each and include a number of other functions around the code-book search. Nevertheless, the times increase linear with the increased number of cycles, indicating a minimal overhead introduced by the delayed assignments.

6 Summary and Conclusion

Push-button synthesis is not accepted by most designers because of the lack of control. Interactive synthesis using graphical user interfaces goes into the right direction. However, it is still desirable to provide an actual HDL description to the designer that can be simulated and freely manipulated. Furthermore, this description should be a behavioral model, since structural models are simply too complex.

This paper provides an easy and straightforward modeling solution that allows behavioral RTL models to be cycle-accurate, even in the presence of multi-cycle and pipelined components. This approach uses delayed assignment statements that are specified by use of simple **after** and **piped** clauses, modeling multi-cycling and pipelining, respectively.

Finally, the listed experimental results demonstrate that the approach is not only feasible, but also applicable and practical towards simulation and synthesis of real-world designs.

References

[1] Accellera C/C++ Working Group of the Architectural Language Committee. *RTL Semantics, Draft Specification*. Accellera, February, 2001. `http://www.eda.org/alc-cwg/cwg-open.pdf`.

[2] R. Dömer. *SpecC Reference Compiler and Simulator*, available at `http://www.cecs.uci.edu/~specc/reference/`.

[3] R. Dömer, A. Gerstlauer, D. Gajski. *The SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, Japan, Dec. 2002.

[4] D. Gajski, N. Dutt, C. Wu, Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1991.

[5] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] IEEE. *IEEE Standard VHDL Language Reference Manual, Revision 1993*. IEEE Std. 1076-1993, IEEE, 1993.

[7] IEEE. *Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Std. 1364-1996, IEEE, 1996.

[8] A. Jerraya, I. Park, K. O Brien. "AMICAL: An Interactive High Level Synthesis Environment". In *Proc. of EDAC*, 1993.

[9] H. Juan, D. Gajski, V. Chaiyakul. "Clock-driven performance optimization in interactive behavioral synthesis". In *Proc. of ICCAD*, Nov. 1996.

[10] D. Ku, G. De Micheli. *High-level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[11] B. Landwehr, P. Marwedel, R. Dömer. "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming". In *Proc. of EDAC*, 1994.

[12] D. Shin, A. Gerstlauer, R. Dömer and D. Gajski. *"Interactive C-based RTL Design Methodology"*. Technical Report CECS-TR-03-42, University of California, Irvine, Jan. 2004.

[13] Synopsys, Inc. *Behavioral Compiler*, available at `http://www.synopsys.com/`.

[14] K. Wakabayashi, T. Okamoto. "C-based SoC Design Flow and EDA tools: An ASIC and System Vendor Perspective". In *IEEE Transactions on CAD*, Dec. 2000.

[15] V. Chaiyakul, T. Hadley, A. Nakata, T. Tanimoto. *HY-C LRM 1.2 Rev 1.1*. Y Explorations Inc., 2004.