APPROACH FOR FORMAL VERIFICATION OF A BIT-SERIAL PIPELINED ARCHITECTURE

Henning Zabel, Achim Rettberg and Alexander Krupp
University of Paderborn/C-LAB, Germany
{henning.zabel, achim.rettberg, alexander.krupp}@c-lab.de

Abstract: This paper presents a formal verification for a bit-serial hardware architecture. The developed architecture bases on the combination of different design paradigms and requires sophisticated design optimizations. The recently patented synchronous bit-serial pipelined architecture, which we investigate in this paper, is comprised of synchronous and systematic bit-serial processing operators without a central controlling instance. We add timing constraints at the boundaris of the architectures operators. To prove the validity of data synchronization we apply formal verification of the constraints on a cycle accurate representation of the implementation. The results of the formal verification is back annotated to the high-level model.

1. INTRODUCTION

Bit serial architectures have the advantage of a low number of input and output lines leading to a low number of required pins. In synchronous design, the performance of those architectures is affected by the long lines, which are used to control the operators and the potential gated clocks. The long control lines can be avoided by a local distribution of the control circuitry at the operator level. Nowadays, the wire delay in chip design is close to the gate delay. In asynchronous design an interlocked mechanism is often used to activate the computation on the desired operator and stall the part of the circuitry, which should be inactive. Reactivation is triggered by a short signal embedded in the datastream. While the design of a fully interlocked asynchronous architecture is well understood, realizing a fully synchronous pipeline architecture still remains a difficult task. By folding the one-hot implementation of the central control engine into the data path, and the use of a shift register, we realized a synchronous fully self-timed bit-serial and fully interlocked pipeline archi-

Zabel, H., Rettberg, A., Krupp, A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 47–56.

tecture called MACT[1]. Synchronizers of MACT ensure the synchronization of data. Furthermore, routers allow the selection of different paths.

The implementation of the synchronizing elements is an error-prone task. Therefore, we add CTL properties to each MACT operator and apply formal verification to ensure their correctness. We use CTL to describe timing constraints for the I/O of operators. For synthesis, we can fall back on an already realized high level synthesis for MACT [4, 7], which automatically generates MACT implementations in VHDL out of data flow graphs. Additionally, we generate a cycle accurate representation for the model checker from the data flow graph. We investigated how to use the results from the formal verification within the high-level synthesis. Therefore, we identify operators which potentially belongs to a deadlock by tracing violated CTL properties via outputs towards inputs of the MACT model.

The paper is organized as follows. In section 2 the related work is presented followed by the description of the MACT architecture, see section 3. Section 4 presents the formal verification tool RAVEN we used for our approach. The high-level synthesis it self is presented in section 5, followed by a description of the RAVEN specification model, see 6. The verification of the MACT properties are described in 7. Section 8 sums up with a conclusion and gives an outlook.

2. RELATED WORK

Symbolic model checking is well established in the formal verification of small systems, in particular in the field of electronic system design [3]. Compared to classical simulation, the most notable benefit of model checking is the completely automated ellaboration of the complete state space of a system.

The primary input for model checking is a system model given by a set of finite state machines. As a second input, model checking needs a formal specification of required properties. The requirements are typically given as formulae based on temporal logics. Once having specified a property as a temporal logic formula f_i, a model checker can directly answer the question „Does a given model satisfy property f_i?" by either true or false. On request, a model checker typically generates counter examples when the model does not satisfy a specified property. A counter example demonstrates an execution sequence that leads to a situation that falsifies the property, which is very helpful for detailed error analysis.

Recently, Hardware Verification Languages like PSL, SystemVerilog, and *e* have been standardized, which offer assertion flavours for property definition for an underlying Hardware Design Language[6, 5]. Several verification en-

[1] MACT = Mauro, Achim, Christophe and Tom (Inventors of the architecture)

vironments exist for simulation, and for formal property checking from, e.g., Synopsys, Mentor, Cadence, AxiomDA. The current methodology enables the creation and reuse of so-called Verification-IP.[2, 1] Verification-IP represents a library of assertions for system building blocks, basically. The focus of standardized temporal assertions is on linear temporal logic, but branching temporal logic is supported as well, e.g., with PSL OBE.

In our work we automatically generate code from our model for formal verification by the model checker RAVEN. Similar to the Verification-IP-approach, we apply generalized CTL assertions with our intermediate format and use them as formulae for the model checker.

3. MACT ARCHITECTURE

MACT is an architecture, that breaks with classical design paradigms. Its development came in combination with a design paradigm shift to adapt to market requirements. The architecture is based on small and distributed local control units instead of a global control instance. The communication of the components is realized by a special handshake mechanism driven by the local control units. MACT is a synchronous, de-centralized and self-controlling architecture. Data and control information are combined into one packet and are shifted through a network of operators using one single wire only (refer to Figure 1).

The controlling operates locally only based on arriving data. Thus, there exist no long control wires, which would limit the operating speed due to possible wire delays. This is similar to several approaches of asynchronous architectures and enables high operation frequency. Yet, the architecture operates synchronous, thus enabling accurate estimation of latency, etc. a priori.

MACT is a bit-serial architecture. Bit-serial operators are more area efficient than their parallel counterparts. The drawback of bit-serial processing is the increase in latency. Therefore, MACT uses pipelining, i. e., there exist no buffers, operators are placed following each other immediately.

Implementations of MACT are based on data flow graphs. The nodes of these graphs are directly connected, similar to a shift register. Synchronization of different path lengths at nodes with multiple input ports is resolved by a stall mechanism, i. e., the shorter paths, whose data arrives earlier, will be stalled until all data is available (refer to Figure 1). The necessary stall signals run in opposite to the processing direction and are locally limited, in order to avoid a complete stall in the preceding pipeline. The limitation is realized by a so called *block stall* signal, which is locally tapped in a well defined distance.

We consider the flow of data through the operator network as processing in waves, i. e., valid data alternates with gaps. Due to a sophisticated interlock mechanism adapted from asynchronous design, the gap will not fall below

an individual lower bound. Thus, the MACT implements a fully interlocked pipeline. The corresponding signal is the so called *free previous section* signal, which is generated by small logic in each synchronizing control block (see Figure 1). These control blocks are found at each multiple input operator and synchronize packets arriving at different instances of time. The architecture is described in more detail in [8, 9, 7].

4. RAVEN

For formal verification of a MACT model, we apply the RAVEN model checker [10] that supports verification of real-time systems. In the following, we give a brief introduction to RAVEN.

In RAVEN, a model is given by a set of time-annotated extended state machines called I/O-interval structures. I/O-interval structures are based on Kripke structures with [min,max]-time intervals and additional input conditions at their state transitions. A more detailed and formal specification of I/O-interval structures is presented in [11].

For property specifications on a given set of I/O-interval structures, the Clocked Computation Tree Logic (CCTL) is applied [12]. CCTL formulae are composed from atomic propositions in combination with boolean connectives, and time-annotated temporal operators. Timing interval specification is a significant benefit of CCTL when being compared to the classical Computation Tree Logic (CTL) [3].

In the context of RAVEN, I/O-interval structures and a set of CCTL formulae are specified by means of the textual RAVEN Input Language (RIL). A RIL

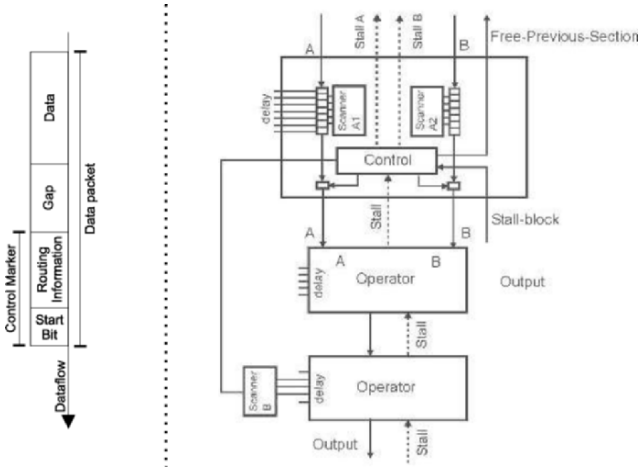

Figure 1. Example data packet (left) and Synchronisation (right)

specification contains (a) a set of global definitions, e.g., fixed time bounds or frequently used formulae, (b) the definition of parallel running modules, i.e., a textual specification of I/O-interval structures, communicating via signals and (c) a set of CCTL formulae, representing required properties of the model.

5. THE MACT HIGH LEVEL SYNTHESIS (MHLS)

MACT models are created upon a predefined set of MACT low level modules. This are operators, delays and synchronisation elements. A MACT model is designed with the graphical MHLS editor. The developer defines the input/output lines and the bit-serial operators. Out of this the MHLS can generate VHDL code for synthesis. For operators the input data packets must arrive synchronous, so that the corresponding data bits are processed at the same clock cylce. For this the MHLS adds automatically synchronisation elements like *Delay* or *DataSync* to the model. Additionally the operators are combined in different sections. These sections prevent the stall conditions from beeing propagated through the complete model (see section 3). Adding this synchronisation elements is a hard task. Wrong placed delays or programming mistakes escpecially in wireing the synchronisation lines can hold off the model from working correctly. In our approach we use model checking techniques to verify that the model conforms to the specification and most important that we don't introduce deadlocks with the stalling mechanism.

To verify this we use the model checker RAVEN. For this we translated the low level MACT modules to a cycle accurate representation in RIL. Withit our code generator composes a complete RIL model description upon the compositional description of the used low level MACT modules. The code generator depends on a XML description. Each module has its own XML file, that contains module descriptions i.e., the names of input and output signals, and a description of the composition of several low level modules or for low level modules the name for the RIL file. This description is very similar to a component description in VHDL and can be generated by the MHLS editor directly. The XML files also contains CCTL formulae with properties over local input and outputs signals, which are automatically added to the RIL file for further processing.

6. SPECIFICATION OF MACT MODULES IN RIL

The lower level MACT modules like *Add*, *Sync* and *Delay* are originally given as a VHDL file. There exist no more than 20 modules and by reason that MACT is a bit-serial architecture these VHDL files are very small too. The effort to tanslate these modules to RIL is manageable, so we have done it by hand.

When mapping these modules to RIL, we have to find a representation for signals, processes and the hierarchie in components. Signals can be devided in clock dependent and clock independent signals.

We translated each VHDL process as RIL module. Figure 2 shows an example of VHDL code and its corresponding RIL code. By instanziating the

```
stall_out <= stall_in;

and : process (CLK,RESET)
begin
  if RESET = '1' then
    summ  <= '0';
  elsif CLK'event and CLK = '1' then
    summ  <= line_0_in & line_1_in;
  elsif
    summ <= summ;
  endif
end
```

```
MODULE and:
  SIGNAL
    rv  :BOOL
  INPUT
    line_0_in := ?
    line_1_in := ?
    stall_in  := ?
    reset     := ?
  DEFINE
    // -- outputs --
    line_out  := summ
    stall_out := stall_in

    // -- boolean term for rv --
    rv_cond := !reset /\
           (line_0_in /\ line_1_in)
  TRANS
    |- TRUE -- :1 --> rv := rv_cond
END
```

Figure 2. VHDL module (left) and its representation in RIL (right)

processes the hierarchie can be flattenend. We extended the signal names with the process names to keep unique identifiers for the signals.

In RIL all signals change their value only within a transition step. In our specification one transition step in RIL corresponds to one clock cycle in the VHDL design. Thus clocked signals can be represented as RIL signals in their corresponding RIL module. Clock independent signals are defined as boolean terms over other signals. It is always true, that their value is present at the next rising edge of the clock. Otherwise the design is overclocked in relation to the longest path. If all the clock independent signals belongs to clocked signals as source, then these signals can be represented as a DEFINE in RIL. Because in this case they are only placeholders for their source signals with an additional boolean term.

This representation is only true, if there are no logic cycles within clock independent signals. But only the *Sync* module has logic cycles without a clock to model a RS-flipflop. All other clock independent signals are defined over clocked ones, but generally not within the same module.

7. VERIFICATION OF MACT PROPERTIES

MACT models are triggered by a leading 1 within the datastream. Usually, MACT is used for pipelined data processing, so every valid input packet to a MACT model should lead to a valid output packet. If the model gets no input, then it shouldn't produce output. For synchronisation, data packets can

be stalled within a MACT model. To avoid deadlocks it is important to prove that these stalls always disappear.

7.1 CTL formulae

Most of the MACT lowlevel modules only pass through the stall informations. Stalls itself are only generated in the *DataSync* and *Sync* modules. The following CCTL properties have to be true for a deadlock free operation of a specific model. In general all MACT modules must hold these properties, but specially the synchonisation modules.

1 We always get data, eventually: $AG(!line\_in \rightarrow AF(line\_in))$

2 We never stall our inputs forever: $AG(stall\_out \rightarrow AF(line\_in \rightarrow AF(!stall\_out)))$

3 We never get stalled forever: $AG(stall\_in \rightarrow AF(!stall\_in))$

4 We always output data, eventually: $AG(!line\_out \rightarrow AF(line\_out))$

There are some relations between these formulae: At first Property 1 and 4 are true, if valid data arrives or leaves. But this is also true if the module is stalled, because the formulae only grants that data arrives (leading 1), not disappears. At second, Property 2 only makes sense, if property 1 is true. This is because the second implication is also true even if no data arrive and thus the stall will never disappear.

7.2 Detecting stalling modules

The codegenerator adds synchronisation elements to the implementation of the MHLS model (see figure 3). All these added elements can be associated with one module in the MHLS model. Because of this, we can detect stalling elements in the MHLS model by detected stalling modules in the implementation. Although stalling elements are introduced during codegeneration, this is a usefull benefit to localise the deadlock.

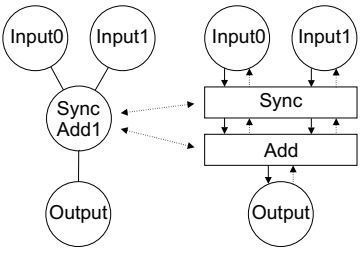

Figure 3. MHLS Model and implementation model

At first the above mentiond formulae should be checked for all inputs and outputs. If they are all true, then we know that data passes the MACT model and we can stop here. If we encounter a problem, the model checker can produce a counter example. For big models it is very hard to localise the source of a deadlock simply by looking at the waveforms (counter example).

An easier way is to check the above formulas step by step on every module from the outputs up to the inputs by repeating the following steps. They will detect and mark all modules in the MHLS model, that are involved in a deadlock.

1 if the module is already involved in a deadlock (marked) then return

2 check all formulas of the local module

3 if the module itself is stalled by its successor (formula 3) then this module can be marked and the algorithm returns

4 if one of the inputs gets stalled (formula 2) or sends no data (formula 1), then mark them and proceed recursive with it.

This algorithm can be realised by the use of RAVENs interactive mode. It allows to start RAVEN with a complete model and different CCTL formulae. The verification of a specific formula can be triggered via a command line interface.

7.3 Example

To prove our method, we use a MACT model that adds four numbers like shown in figure 4. The inputs get valid MACT packets in well defined intervals, expcept input three: The decision wether a packet is send or not is nondeterministic, that means there exists execution pathes where no packet will be send.

We generated RIL out of the model and invoked RAVEN. RAVEN needs 40ms to verify this model on an Pentium4 with 3GHz and creates a transition table with 1841 BDD Nodes. The result of the formulae is shown in figure 5.

All checks for receiving and sending data are false for the successors of input three up to the output. If we apply our algorithm from 7.2, the grayed operators in figure 5 are marked. These are exactly the modules involved in the deadlock. Of course *add0* and *add1* stalls their other inputs to wait for data on the other line, but anyway formulae „don't stall input forever" is true. This is because the second implication in the formulae, which requires a valid input. In our case this doesn't occur, so the implication is always true.

We also tested this apporach on a much bigger model of a discrete cosinus transformation (DCT), but RAVEN breaks the limit of 4GB memory, when it generates it's internal data structure. Thus, for larger models an abstraction is

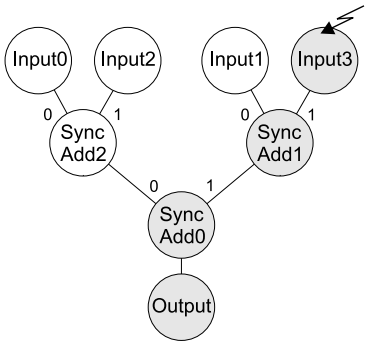

Figure 4. Example Model

Formulae	add0		add1		add2	
	line 0	line 1	line 0	line 1	line 0	line 1
always get data	true	false	true	false	true	true
never stall input forever	true	true	true	true	false	false
never get stalled forever	true	true	true	true	false	false
always output	false	false	false	false	true	true

Figure 5. Model checking results

needed. For example the operators can be abstracted as simple delays, because the operation itself has no direct influence on the stalling mechanism.

8. CONCLUSION AND OUTLOOK

We presented an approach to generate a cycle accurate representation of VHDL implementation of the bit-serial architetcture MACT in RIL. It is composed of small sub-modules and we automatically generate CTL formulae for them to check local data flow properties. Based on this formulae we presented a depth search algorithm to detect stalling modules.

We showed that this approach is applicable for small models and it finds the stalling path. For larger models an abstracted is needed, that only represents the cycle accurate stalling mechanism.

We plan to integrate this algorithm in our MHLS editor. With it we can visualise the modules that are involved in the deadlock. This should help the developer for the MHLS code generator to detect programming errors. It also helps the developer of a MACT model to integrate sufficient delays to its model.

The automated generation of CCTL formulae for each module can also be applied to verify timing constraints of the MACT model too. Exspecially timing that belongs to the length of a data packet is important and could be in-

tegrated during the high level syhntheses. RAVEN has special extensions for such timing constraints.

REFERENCES

[1] M. Bartley, D. Galpin, and T. Blackmore. A comparison of three verification techniques: Directed testing, pseudo-random testing and property checking. In *Proceedings of DAC 2002*, New Orleans, June 2002. ACM.

[2] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer, 2006.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[4] Florian Dittmann, Achim Rettberg, Thomas Lehmann, and Mauro C. Zanella. Invariants for distributed local control elements of a new synchronous bit-serial architecture. In *Second IEEE International Workshop on Electronic Desing, Test and Applications (DELTA 2004)*, pages 245–250, Perth, Western Australia, 28 - 30 January 2004.

[5] IEEE. *IEEE Std.1800-2005 - Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language*, November 2005.

[6] IEEE. *IEEE Std.1850-2005 - IEEE Standard for Property Specification Language (PSL)*, September 2005.

[7] Achim Rettberg, Florian Dittmann, Mauro C. Zanella, and Thomas Lehmann. Towards a high-level synthesis of reconfigurable bit-serial architectures. In *Proceedings of the 16th Symposium on Integrated Circuits and System Design (SBCCI)*, Sao Paulo, Brazil, 8 - 11 September 2003.

[8] Achim Rettberg, Thomas Lehmann, Mauro C. Zanella, and Christophe Bobda. Selbststeuernde rekonfigurierbare bit-serielle pipelinearchitektur. Deutsches Patent- und Markenamt, December 2004. Patent-No. 10308510.

[9] Achim Rettberg, Mauro C. Zanella, Christophe Bobda, and Thomas Lehmann. A fully self-timed bit-serial pipeline architecture for embedded systems. In *Proceedings of the Design Automation and Test Conference (DATE)*, Messe Munich, Munich, Germany, 3 - 7 March 2003.

[10] J. Ruf. "RAVEN: Real-Time Analyzing and Verification Environment". *Journal on Universal Computer Science (J.UCS), Springer, Heidelberg*, February 2001.

[11] J. Ruf and T. Kropf. "Modeling and Checking Networks of Communicating Real-Time Processes". In *Int. Conf. on Correct Hardware Design and Verification Methods*, Bad Herrenalb, Germany, 1999. Springer-Verlag.

[12] Jurgen Ruf and Thomas Kropf. "Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals". In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, Montreal, Canada, October 1997.