Model Checking of S3C2400X Industrial Embedded SOC Product

Hoon Choi, Byeongwhee Yun, Yuntae Lee, and Hyunglae Roh
SOC Development Group, System LSI, Samsung Electronics, Yongin-City, Kyunggi-Do, Korea
hchoi@ieee.org, {bwyun, yuntaelee, shlroh}@samsung.co.kr

ABSTRACT

This paper describes our experience and methodology used in the model checking of S3C2400X industrial embedded SOC product. We employed model checking to verify the RTL implementation. We describe how to model the multiple clocks, gated clocks, unsynchronized clocks, and synchronization logics in model checking. Detailed case studies of real designs show the application of the proposed modeling techniques, environment modeling, and the properties we checked. The verification results validate the proposed techniques by finding real bugs.

1. INTRODUCTION

This paper describes our experience and methodology used in the intensive use of formal verification in the design of S3C2400X embedded SOC product. The overview of S3C2400X is shown in Fig. 1. It is composed of an ARM920T processor and 16 function modules, i.e., IPs.

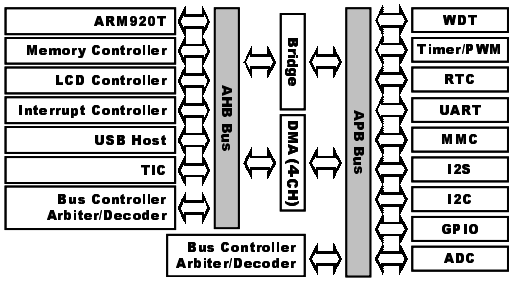


Fig. 1: Block diagram of S3C2400X SOC product

We can classify the IPs into three groups in the verification point of view. First, many IPs have been used in the previous products and verified in silicon, e.g., memory controller, UART, I2S, etc. We just changed the interface logic for new bus systems, i.e., from SSB/SPB to AHB/APB [8]. In this case, those bus systems were so similar to each other that we could verify the interface logic easily with just a simulation. Second, some IPs were newly designed for the new bus systems, e.g., bus controllers, DMA, etc. We used model checking to verify the correctness of those IPs. Last, we bought USB host controller (UHOST) as an IP [9] and designed interface logic to attach it to the AHB bus. In this case, the interface protocol of UHOST is significantly different from that of AHB system. Specifically, the former one uses FIFO based protocol (HCI protocol), while the latter one uses a pipelined bus protocol. Furthermore, they run at different unsynchronized clocks. This led us to use model checking to guarantee the correct operation of the designed interface in all the possible cases, e.g., various combinations of different clocks.

In this paper, we describe the model checking techniques used for the second and the third classes. We employed model checking to verify the RTL implementation. The rest of this paper is organized as follows: In Section 2, we briefly present the selection of a model checker and a modeling language and the overview of our design/verification flow. The modeling details, i.e., how to model the multiple clocks, gated clocks, unsynchronized clocks, and synchronization logics are covered in Section 3. Section 4 describes the details of actual verifications, i.e., case studies, with verification results. Discussions and conclusions are given in Section 5 and Section 6, respectively.

2. MODEL CHECKER AND LANGUAGE

We used SMV [1] as our model checker because it has many good features to support real designs and there are many success stories from the industry [2][3][4][5][6][7]. SMV supports various features to reduce the problem size, i.e., the *scalarset* data type for symmetric reduction, the *ordset* data type for induction, the *subclass* structure for case-splitting, the *layer* structure for the compositional assume-guarantee verification, and the property based reduction capability.

SMV supports SMV language (SMVL) and Verilog (actually it is translated into SMVL before verification) as the modeling language. SMVL is better than Verilog in controlling and exploiting the full power of SMV, especially in problem size reduction. In addition, its macro definition capability is very useful in handling multiple clocks conveniently. Hence, we used SMVL as our main modeling language. The environment (i.e., abstracted models of other modules needed for the verification of a module) is also modeled in SMVL using non-deterministic values. It can significantly abstract out the details of the environment so that the complexity of the environment as well as the amount of modeling work can be significantly reduced.

Now, we briefly describe our design/verification flow shown in Fig. 2. Each module is designed in either the SMVL or the Verilog. The modules written in Verilog are translated into those in SMVL using vl2smv utility. The modules in SMVL and the properties (to be checked) written in CTL are processed by the model checker. If the model checking is passed without any failure, we become to get the golden models. The SMVL golden model is translated into the Verilog golden model. Since, to the best of our knowledge, there is no available SMVL to Verilog translator, we manually perform this translation. However, since almost all the syntax of SMVL has its corresponding part in Verilog, the translation is relatively an easy task (This is especially true for the subset of Verilog used for the synthesis, and we guided to use only the subset of SMVL that has its corresponding part in that subset of Verilog.). We verify the correctness of the translated code by translating it back to SMVL and then performing the model checking.

After obtaining the golden model, we refine the RTL code for

38th DAC, June 18-22, 2001, Las Vegas, Nevada, USA.

more efficient synthesis. To verify the correctness of these refinements, we use the equivalence checker. The conventional simulation and the synthesis follow the refinement step. Optionally we may perform the final check by model checking the compiled gate-level netlist.

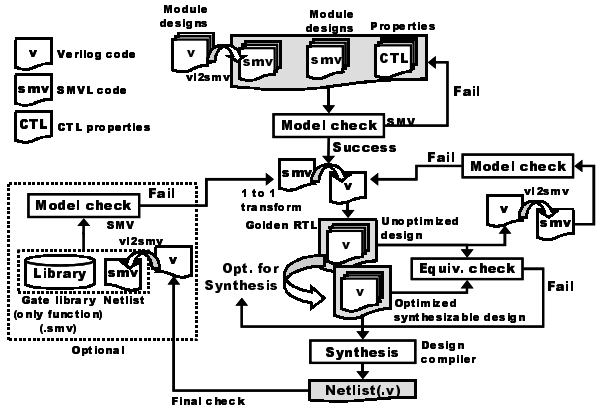


Fig. 2: Overview of design/verification flow

In our design flow, the formal verification is tightly coupled with the RTL design procedure. It is used to get the correct and verified RTL designs by our RTL designers, not for a separate verification procedure performed by a separate verification team. This use of formal verification enabled our designers to detect and eliminate many design errors (even very complex ones) easily at the very early stage of our RTL design. The verification time itself for a module with appropriately abstracted environments was almost comparable to the simulation time using a large number of vectors. In short, the use of the formal verification gave us not only the high confidence on the correctness but also the reduced design time.

3. MODELING DETAILS

In this section, we describe the modeling details: multiple clocks and gated clocks that are very common in RTL designs, unsynchronized clocks and synchronization logics that are used in the UHOST interface.

3.1 Multiple clocks

In the modeling, we have to handle two different clocks, i.e., AHB clock and APB clock. However, SMV supports only one implicit clock and SMVL does not support any syntax to describe explicit multiple clocks. This is also true for the Verilog code translated into SMVL: All the *always* statements using *posedge* and/or *negedge* are converted such that the registers are updated at the same next time step of a single implicit clock.

To circumvent this problem, we use the following techniques. The AHB clock, i.e., HCLK, having both edges is generated as follows: *init(HCLK) := 0; next(HCLK) := ~HCLK*. And we define *PNEXT* macro for the update at the positive edge such that *PNEXT(HCLK, d) := s* is converted into *if(~HCLK) next(d) := s*. Then signal *d* is updated to the value of *s* at the next implicit clock where *HCLK* makes a low-to-high transition. Similarly, we define *NNEXT* macro for the update at the negative edge.

For the APB clock (i.e., *PCLK*) that runs at a half frequency of *HCLK*, we cannot use the same approach. If we generate *PCLK* as *init(PCLK) := 0; PNEXT(HCLK, PCLK) := ~PCLK*, the *PCLK* has two cycles of high and two cycles of low as shown in the Fig. 3-(a). However, in this case we cannot use *PNEXT(PCLK, d) := s* because there are two implicit time steps, i.e., t_a and t_b, in the low phase of *PCLK*, which makes *d* updated twice at t_a and t_b while we actually want it to happen only once at t_b. This may result in incorrect results (e.g., *PNEXT(PCLK, s) := s + 1* may increase *s* twice in one cycle). To handle this problem, we use an asymmetric *PCLK*, i.e., three cycles of high and one cycle of low for the positive edge of *PCLK* as shown in Fig. 3-(b), and one cycle of high and three cycles of low for the negative edge. Then the signal *s* is updated only at t_b as we want. (Note that this method is possible partially because the AHB modules in our design get the data from APB modules only at the rising edge of HCLK.)

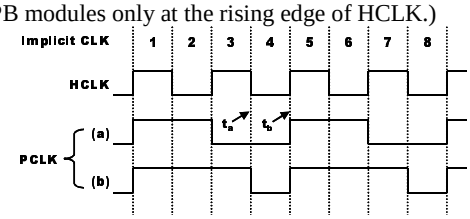


Fig. 3: Handling multiple clocks

3.2 Gated clocks

The use of gated clocks also causes similar problems as the multiple clocks. For example, if we use *PNEXT* macro for the gated clock shown in Fig. 4, data will be latched not only at t_e where we actually want but also at t_a~t_d because the gated clock is low at those time instances. We have to solve this problem for the correct verification.

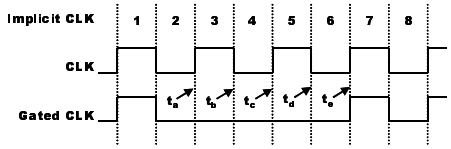


Fig. 4: Problem of gated clocks

In our design, gated clocks are used for two purposes. First, we use gated clocks for low-power consumption. In this case, the clock gating is performed at a module-granularity, i.e., entire blocks of a module are clock gated, and we usually keep the gating logic in a separate module, i.e., the gating logic does not reside in the module under test. Furthermore, in the functional verification we mainly concern the normal operation mode, not the power down mode. Therefore, we could verify those clock-gated modules without considering the clock gating effect.

Second, the gated clock is used for data transfers, e.g., MMC controller. In the transfer of data to the MMC card, MMC controller uses gated clocks to indicate the time instances for the card to get the data. Here, the MMC controller gates out the clock edge if there is no valid data on the data bus. For example, in Fig. 5 the rising edges of GCLK1 (generated by MMC controller) indicate the time instances at which the valid data can be obtained from the bus. The missed rising edge of GCLK1 means that data is not available at that time instance, i.e., time 5 and 6.

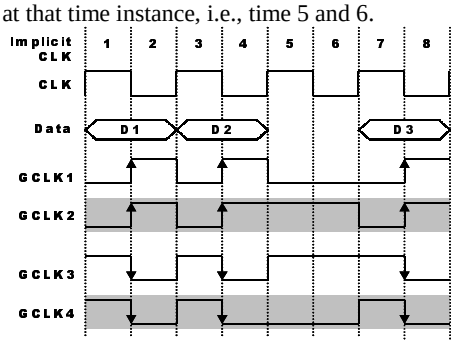


Fig. 5: Handling gated clocks

However, we cannot use the GCLK1 as it is in SMV because

the low value of GCLK1 at time 5 makes the data latched at time instance between 5 and 6. To circumvent this problem, we changed MMC controller to generate GCLK2 instead of GCLK1. As we see, the high value of GCLK2 at time 5 and 6 solve the problem of GCLK1. Similarly, we use GCLK4 in place of GCLK3 for the falling edge. Note that such a change requires not only the AND gate to OR gate replacement in the gating logic but also the change of gating timing. In our case, such a change was possible, thus we could use this approach.

3.3 Unsynchronized clocks

The two unsynchronized clocks of UHOST interface have to be modeled in such a way that all the possible cases are covered. The modeling should not restrict the covering of possible cases.

We first considered the use of a fine clock and two counters. For example, given HCLK running at 100MHz and UHOST clock (UCLK) running at 12MHz, two counters counting 3 and 25, respectively, can mimic those clocks: When each counter reaches its own limit, it inverts its output. However, this approach has two problems: First, these predefined clocks cannot guarantee the covering of all the possible cases such as the speed change of HCLK in different operation modes. Second, the two counters increase the number of state variables, thus slow down the verification. Thus, we use a different approach.

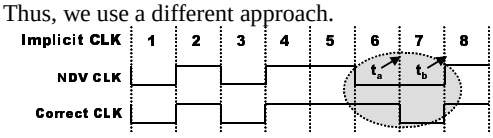


Fig. 6: Clock modeling using non-deterministic values

We use non-deterministic values in modeling clocks. However, we do not use the non-deterministic value directly as a clock. For example, *NDV CLK* in Fig. 6 shows the clock modeled directly by a non-deterministic value. In time 6 and 7, *NDV CLK* is consecutively low, thus problems happen at those points as the gated clocks. To solve this problem, we have to force *NDV CLK* to have only one cycle of consecutive lows at the maximum as shown in *Correct CLK*. To make such a clock, we use a FSM shown in Fig. 7 where *ndv* represents a non-deterministic value. The fairness constraint prevents *CLK* from staying at 1 forever. We use two copies of this FSM, one for each clock.

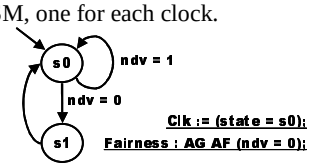


Fig. 7: FSM modeling an unsynchronized clock

3.4 Synchronization logic

We use double synchronization FFs (DS-FFs) shown in Fig. 8 in the UHOST interface. If *In* meets the setup time of the first FF, the *Out'* and *Out* become stabilized as shown in (a). However, if it is not the case, *Out'* and *Out* are delayed by one clock cycle as shown in (b).

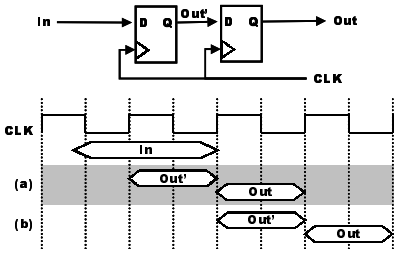


Fig. 8: Double synchronization flip-flops

To model this effect, we first considered the use of a sequence of internal FFs (running at a fine clock) and some gates: if and only if all the internal FFs have the same value, the setup time is regarded as being met, and the output reflects the input. However, this model increases the number of FFs significantly, which degrades the verification performance. In addition, this model requires the *fine clock* (mentioned in Section 3.3). Hence, we use a different approach using non-deterministic values.

We use a FSM shown in Fig. 9 where *Get* is a non-deterministic value. This FSM is based on the fact that *Out'* gets the input data at the maximum of one clock cycle delay, i.e., Fig. 8-(b) case. Note that we do not use this FSM for each communication signal. Instead, we use only two, one for signals from HCLK to UCLK, and the other for those from UCLK to HCLK. This is because if we use separate FSMs for each signal, some signals may reach receiver's domain while others may not, even if all of them have waited at the input of DS-FFs concurrently. This is not the case in real circuits.

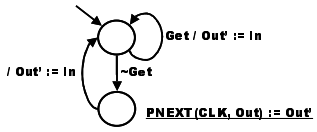


Fig. 9: FSM modeling DS-FFs

4. CASE STUDY

4.1 Verification of DMA

DMA contains AHB slave part (for accessing control registers, CRs), AHB master part, and APB master part. It performs AHB to APB, APB to AHB, and AHB to AHB DMA operations. We divided the verification into two modes for the reduction of problem size: One was the *ARM to CR access mode* and the other was the *DMA operation mode*. In *ARM to CR* access mode, we checked only the CR read/write accesses of the ARM core. On the other hand, in the *DMA operation mode*, we checked the DMA operation itself with assuming some fixed CR values.

Among the seven CRs, three can be written only by the ARM core, other three only by the DMA core, and another one by both of them, while all the CRs can be read by the ARM core. Hence, the write to the first three could be thoroughly verified in the *ARM to CR access mode*. On the other hand, the write access to the other three could only be verified in the *DMA operation mode*, and the last one partially in both of the verification modes.

The verification environment for *ARM to CR access mode* is shown in Fig. 10. It is composed of a simple model of ARM core (a simple FSM to test only the CR accesses), a simple AHB arbiter, and a simple AHB decoder. The shaded area represents the blocks that are not verified in this mode. Those blocks were modeled very simply using non-deterministic values. The clear block boundary between the ARM to CR access blocks and the DMA operation blocks were very helpful in this abstraction, thus we believe that we should consider this kind of verification requirement from the start of the design.

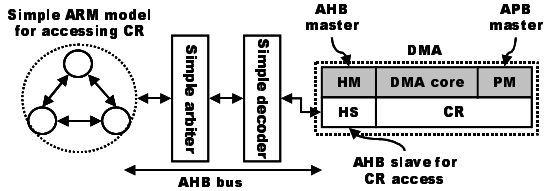


Fig. 10: Environment for ARM to CR access mode

In this environment, we tested 12 CTLs including 5 vacuous

checks. The execution time was about two minutes. Here we found one critical design bug that could occur during the burst access of CRs. The change of allowed access modes during the design caused this bug.

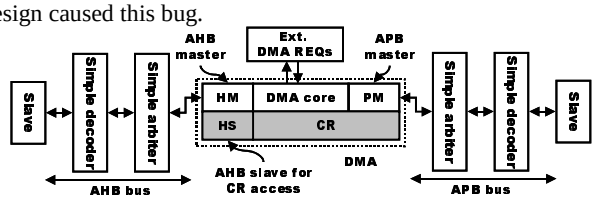


Fig. 11: Environment for DMA operation mode

The verification of DMA operation mode started with assuming CRs to have some fixed values for a specific DMA operation mode. Here we applied the case-splitting technique. In short, we tested each of DMA operation modes, separately. In addition, we also applied the compositional verification: For each mode of the operation, we assumed the correctness of AHB arbiter, AHB decoder, APB arbiter, and APB decoder. This enabled us to use a simple verification environment shown in Fig. 11. The slave was used to check the correctness of the transfer. Though we could do this by directly looking at the bus signals, to know the bus signal sampling time we needed a FSM knowing the bus protocol. The slave was used as the FSM in our verification. The shaded CR access blocks were modeled using non-deterministic values such that all the possible cases were covered.

The properties were also written in such a way that each different mode of operation was verified separately. The problem here was the large number of variables (i.e., 16) whose combinations decide different modes of operations. To solve this problem, we exploited the fact that operations initiated by some variables do not depend on those initiated by other variables. For example, interrupt requesting at the end of transfers initiated by the *interrupt/polling* variable has nothing to do with specific bus transfers initiated by other variables such as the *source selector*. For such variables, we do not need to test all the possible combinations of them. We built a graph showing such a relation among the variables, and then elicited the minimum set of properties to test.

In this verification, we verified 67 properties (including 6 vacuous checks) each of which have about 100 state variables. The execution time was about 7 hours and 20 minutes. We found one critical design error (i.e., HTRANS was not returned to IDLE after a burst transfer in some situations) and fixed it.

One of the interesting verification was the deadlock checking. The operations of DMA and bridge use both of busses (i.e., AHB and APB) as masters, thus there is a possibility of a deadlock. Hence, to prevent such a deadlock we use some mechanisms in the design and have to verify them.

The environment for this checking requires both the DMA and the bridge. However, we could use a very simplified version of the bridge instead of the full complexity one by modeling the bridge as a simple FSM having four states. This FSM models the bus requesting behavior of the bridge that is relevant to the deadlock verification. It requests AHB bus, and if granted it requests APB bus. It uses three non-deterministic values to model the various kinds of transfers and to decide the bus release time.

The AHB arbiter was also modeled using a simple FSM having five stages. It models the behavior of the arbiter only for the two request sources, i.e., DMA and bridge. It gives a grant to one of DMA or bridge, and waits for the release signal from the granted master to model the bus ownership. The APB arbiter was also modeled similarly. Fig. 12 shows the verification environment.

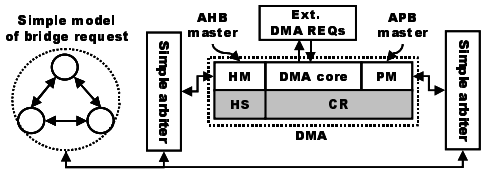


Fig. 12: Verification environment for deadlock

In this environment, we verified 13 properties (including 4 vacuous checks) having about 90 state variables. CPU time was about 23 minutes, and we could verify the correctness of the deadlock preventing mechanisms.

4.2 Verification of UHOST interface

4.2.1 Overview of interface logic

Fig. 13 shows a part of AHB system, interface logic, and USB host (UHOST). UHOST is composed of a core, a HCI master interface (read/write), and a HCI slave interface (read/write). Our interface logic performs interfacing between the HCI I/F of UHOST and the AHB bus. In the *master write operation* of UHOST, master WR I/F of our interface receives a sequence of data from HCI I/F and then write them to the system memory via AHB bus. Similarly, in the *master read operation* our interface reads in a sequence of data requested by the UHOST HCI I/F from the system memory, and then gives it to UHOST. In slave operation, ARM920T writes control-words to the UHOST control registers via our interface's slave WR I/F. In read, ARM920T reads control-words via our slave RD I/F.

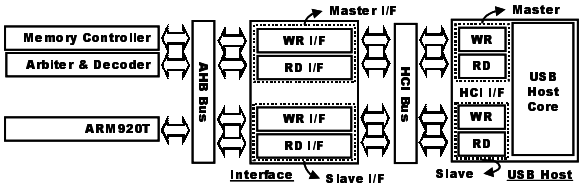


Fig. 13: Interface logic and its environment

4.2.2 Design partitioning for verification

Partitioning of a design for the verification is important in model checking because the design size is one of the most important factors deciding the success of model checking. In our previous designs, modules designed without a proper partitioning caused model checking very hard and inefficient, and at the end we had to redesign those modules considering partitioning for verification. This experience led us to consider the appropriate partitioning as soon as the building blocks of our interface were determined (before the actual coding). The partitioning was mainly for the ease of verification, not for the ease of design.

The result of partitioning is shown in Fig. 13. The partitioning was performed to exploit the case splitting technique. For example, we divided the interface into two, i.e., one for HCI master operation and the other for HCI slave operation. Furthermore, we divided the interface for the master operation further into two, i.e., read and write. Similarly, that for the slave operation was also divided. This partitioning is based on the fact that we can check the correctness of our interface by checking those four partitions separately. This partitioning caused the interface logic to have some duplicated logics, i.e., not an optimal design in area, but the redundancy was very marginal and the verification became much easier and efficient. Thus, we believe that such a partitioning for verification is very important for model checking.

4.2.3 Environment modeling

We used two different verification environments, one for mas-

ter operation and the other for slave operation. The verification environment for the master operation is shown in Fig. 14. Specifically, it shows that for master writing operation. Here we model the memory controller and memory, arbiter, decoder, and HCI I/F writing master.

Memory controller and memory were modeled using a simple AHB slave with a depth four buffer. The model was abstracted to handle only the transfer types generated by our interface logic. HREADY signal from the memory controller was modeled using a non-deterministic value and a fairness constraint to mimic the various different delays of possible different memories.

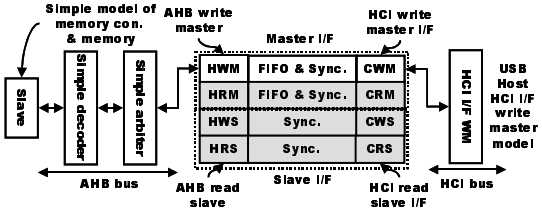


Fig. 14: Environment for master writing operation

AHB arbiter was modeled using a two state FSM and a non-deterministic value (NDV). In state *s1*, if a bus request comes in, it goes to *s2* depending on NDV. If and only if NDV is 1, it goes to *s2*. In *s2*, it asserts a grant signal until the bus request is de-asserted. Though simple, it can mimic the situation where the grant is not asserted due to other bus masters. A fairness constraint is used to prevent the starving case.

We modeled the HCI I/F writing master efficiently also using non-deterministic values. For example, the number of data to transfer, read or write, and byte enables were modeled using NDVs. Furthermore, the latency between requests was also modeled using NDV. This covered all the possible different delays between requests that actually depend on the kind of data to be transferred, e.g., data request requires three cycles of delay while control-data requires no delay.

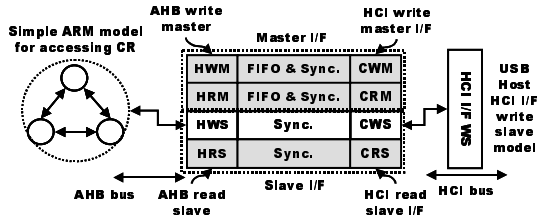


Fig. 15: Environment for slave writing operation

The environment for verifying the slave operation is shown in Fig. 15. Specifically, it shows that for slave writing. Here we need to model the ARM920T that sets the control registers of USB host. We modeled the write operation of ARM920T using a simple FSM. HCI I/F writing slave was also modeled very simply to check just the correct data and address arrivals.

4.2.4 Verified properties

Properties that we checked can be classified into four groups.

Data and address movement checking

This group checked the correct data and address movements. In writing CTLs, we used case splitting, e.g., we divided cases depending on the number of data to transfer, various byte enables, etc. This reduced the number of state variables related to each CTL. In addition, we did not check the correct transfer from one end to the other end (i.e., from HCI I/F to memory, and vice versa) because it involved large number of state variables. Instead, we checked each consecutive fine step of the transfer separately using *assume and guarantee* technique.

HCI bus protocol checking

The USB host IP was delivered with a HCI bus monitor. It was a set of Verilog modules and originally intended to check whether the user designed interface complies the HCI I/F protocol or not.

We translated the monitor into CTLs for model checking. Monitor modules checking simple relations among signals were translated into the corresponding simple CTLs. On the other hand, complex monitor modules implemented in FSMs, e.g., checking whether the number of data pushed into the data FIFO is same to *HCI_MBstCntr*, were translated into one of the two forms. First, in some cases, we could translate it into two or three consecutive CTLs. Second, in many other cases, the Verilog FSM was translated into a combination of a SMVL FSM and a CTL. Here, the SMVL FSM monitors the error condition and sets an error flag, while the CTL says that the error flag never becomes true, i.e., *assert AG ~(error_flag)*. In writing the FSM, *abstract variables* [1] were used not to interfere with the design. This group of properties checked not only our interface design but also the abstracted model of HCI interface that we assumed to be correct.

Stable signal checking

In our interface design, we use double synchronization FFs for the communications between FSMs running at unsynchronized clocks. However, for some signals (mainly data signals) we do not use such FFs to reduce the number of FFs. Thus, for those signals we have to guarantee that those signals are stable, i.e., have no setup time problem, when the receiver latches them.

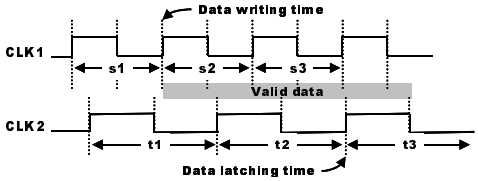
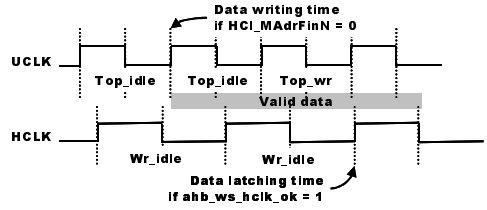


Fig. 16: Stable signal condition

The stable signal condition is shown in Fig. 16. We assume that 1) FSM1 running at CLK1 writes data at the transition from state *s1* to *s2*, and FSM2 running at CLK2 gets the data at the transition from state *t2* to *t3*. Then, the data is stable in FSM2's point of view if and only if the state of FSM1 at one cycle before the latching time, i.e., between *t1* and *t2* of CLK2, is *s2* or later ones and the data is not changed until the latching time.

For example, Fig. 17 shows the stable signal checking of *n_valid_wf* signal coming from UCLK domain to HCLK domain. *HCI_MadrFinN = 0* in *top_idle* state means that a data was already written at the previous rising edge, and *ahb_ws_hclk_ok = 1* in *wr_idle* state describes that data will be latched at the next rising edge. The CTL to check the stability is shown at the bottom of Fig. 17. Note that the state of FSM running at UCLK is sampled at the rising edge of HCLK in the CTL.




```
PNEXT(HCLK, sample) := (top_state = Top_idle) & ~HCI_MAdrFinN |
                       (top_state = Top_wr);
assert AG ((wr_state = Wr_idle) & ahb_ws_hclk_ok -> sample);
```

Fig. 17: Example of stable signal checking

Overrun check

In Fig. 18, *a1* of FSM1 fires FSM2 waiting at *b1*. Assume that the two FSMs run at different clocks and FSM2 runs very fast so that it finishes its job, sends the done signal to FSM1, and comes back to *b1*. If the start signal of FSM1 that fired FSM2 is not cleared yet, FSM2 starts its operation again. We call this situation *overrun* and have to verify that it never happens in all the cases.

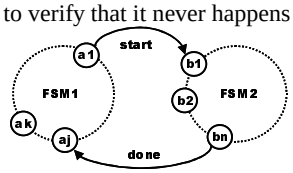


Fig. 18: Overrun condition

This can be stated as follows: Given *b1* receiving a communication signal, in all the state paths of FSM2 back to *b1*, there should exist a state where the communication signal is deasserted in FSM2's point of view. Note that we cannot simply use *U* operator of CTL since the start signal can be deasserted and then reasserted before *b1*, which is also a valid situation.

To facilitate this verification, we restricted our design to use *flip* strategy in which all the communication signals are flipped over (toggled) at the same time of assertion. The receiver also flips over the decision value as soon as it receives an asserted communication signal. This reduced the overrun check to verifying only the correct flipping after every communication state in both the sender and the receiver, which was easily stated and checked in CTL.

4.2.5 Verification results

We first verified our interface with setting both of clocks, i.e., HCLK and UCLK, to a single same clock to detect bugs not related to the unsynchronized clocks, and then the non-deterministic clocks to find bugs related to the unsynchronized clocks.

In the verification of the master writing operation, we checked 102 properties for data and address movement checking, 4 for overrun checking, and 5 for stable signal checking. About a half of them were used to check the vacuous properties. The verification time was about 9 hours for the same single clock and 43 hours for the proposed non-deterministic clock covering all the possible cases. In this testing we found no bugs.

For the verification of the master reading operation, we used 36 properties for data and address movement, 4 properties for overrun check, and 2 for stable value check. It took about 2 hours for the same clock, and about 6 hours for the non-deterministic clock. Here we found no bugs in the same clock environment. However, when we used the proposed non-deterministic clock, we could find one real bug caused by the unsynchronized clocks, i.e., control of buffers between unsynchronized clocks. It would be impossible or at least very hard to find this bug if we used only the same clock or some predefined two clocks. The non-deterministic clock model found this bug.

For the slave reading and writing operations, we tested total 22 properties: 14 for data and address movement and 8 for overrun checks. It took about 2 minutes. The HCI monitor was translated into 23 CTLs and 5 FSMs. Here 15 simple monitor modules were translated into 18 CTLs, and 5 complex ones into 5 FSMs and 5 CTLs. The verification time was about 3 hours. In this verification, we could find one bug in our HCI I/F model: It did not set the *HCI_MBstCntr* signal correctly during the burst-writing mode.

5. DISCUSSIONS

- In many cases, SMV verified not only the design but also the property itself. The incremental design and verification (i.e., both the design and the property grow incrementally by adding new features one by one, and the model checker checks both of them) played an important role in our design.
- At the early stage of our design, we could find real design errors relatively easily using very simple environments. As we began to use more complicated environment to verify the complex properties, many errors detected by the model checker were not the real design errors but the errors in the environment modeling and the property writing. As the environment and the properties were settled, we could find a few but crucial remaining real design errors. The last one is the well-known reason of using model checking. However, we believe that the first one, the early stage use of model checking, is also very important and useful because it enabled us to find bugs early in the design, thus to reduce the design time. In addition, the more efficient ways to model the environment will be very much helpful for the efficient verification.
- The design and verification time was considerably reduced as we got more experiences in model checking. It is now almost equal to or even lesser than the conventional Verilog and simulation based design. Such a speed up was possible due to the reuse of verified SMVL templates for designs and CTLs.
- Though we use formal verification much intensively than the past, yet we have to rely on the conventional system-level simulation to detect the possible bugs in the abstracted environments, interfaces between partitions, and the specification itself. For that purpose, we translate the CTLs into Verilog monitor modules that run in the simulation with other modules and inform us if there is any violation.

6. CONCLUSIONS

This paper described our experience and methodology used in the model checking of our embedded SOC product. We explained how to model the explicit multiple clocks, gated clocks, unsynchronized clocks, and synchronization logics. Detailed case studies showed the actual application of the modeling techniques, environment modeling, and the properties we used. The verification results validated the proposed methods by finding real bugs.

Thanks to the proposed methodology, we could get the functionally verified designs in a reasonable time. We now believe that the model checking is a very useful, mature, and affordable technology that industries can use for SOC products.

REFERENCES

[1] K. L. McMillan, *Cadence SMV*, available at *http://www-cad.eecs.berkeley.edu/~kenmcmil*.

[2] B. Chen, M. Yamazaki, and M. Fujita, "Bug Identification of a Real Chip Design by Symbolic Model Checking," in *ED&TC*, pp. 132-136, 1994.

[3] J. Bormann, J. Lohse, M. Payer, and G. Venzl, "Model Checking in Industrial Hardware Design," in *32nd DAC*, pp. 298-303, 1995.

[4] J. Lu, S. Tahar, D. Voicu, and X. Song, "Model Checking of a Real ATM Switch," in *ICCD*, pp. 195-198, 1998.

[5] A. Th. Eiriksson, "Integrating Formal Verification Methods with A Conventional Project Design Flow," in *33rd DAC*, pp. 666-671, 1996.

[6] K. Takayama, T. Satoh, T. Nakata, and F. Hirose, "An Approach to Verify a Large Scale System-on-a-chip Using Symbolic Model Checking," in *ICCD*, pp. 308-313, 1998.

[7] H. Choi, M.K. Yim, J.Y. Lee, B.W. Yun, and Y.T. Lee, "Formal Verification of an Industrial System-on-a-chip," in *ICCD*, pp. 453-458, 2000.

[8] *AMBA Specification manual*, England: ARM, 1999.

[9] *USB Host Controller User's Manual*, Phoenix Tech., 1999.