

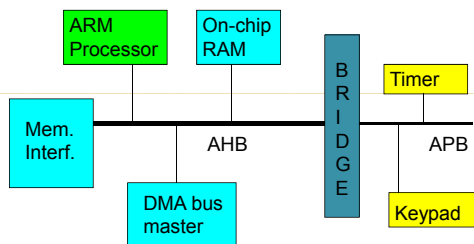
## Experience in using SMV

Abhik Roychoudhury  
CS 427I

## A real-life application

- Verifying the AMBA Advanced High-performance (AHB) bus protocol in ARM processors.
- Cannot discuss all the details, but it gives a feel of:
  - the practical usage of model checking
  - Simplifications involved in modeling
    - The original AMBA AHB document runs to 60 pages, see
    - [http://www.comp.nus.edu.sg/~abhik/CS427I/AMBA\\_SPEC.pdf](http://www.comp.nus.edu.sg/~abhik/CS427I/AMBA_SPEC.pdf)
    - Look at chapter 3 (AMBA AHB)
  - How subtle errors can creep into real designs !!

## Bus-based SoC design



## Bus Protocols

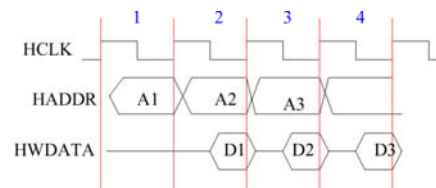
- Popularity of bus-based SoC designs necessitate the verification of bus protocols.
- Different from testing/validating the cores.
- SoC Bus Protocols often involve **advanced features** for high speed data transfer, leading to **corner cases**
  - **Pipelining**
  - **Wait Cycles**
  - **Split Transfers**
- **Case study:** AMBA AHB protocol from ARM.

## Bus architecture

- Several masters and slaves are connected to AHB.
- An **arbiter** decides which **master** will transfer data.
- Data is transferred from a master to a slave in **bursts**.
- Any burst involves read/write of a sequence of addresses.
- The **slave** to service a burst is chosen depending on the addresses (decided by a **decoder**).
- AHB is connected to APB via a bus bridge.
- Let us study the transfer features of AHB protocol

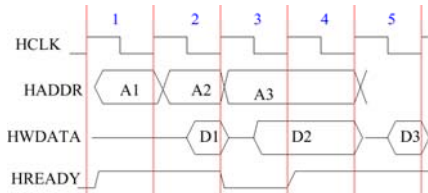
## Pipelining within a burst

Address and data of consecutive transfers are transmitted in same clock cycle,



## Wait cycles

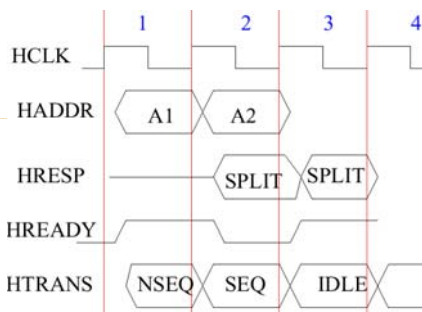
Slave may not be ready to service request.  
Inserts Wait cycle(s) by de-asserting HREADY



## Split response

- If the slave thinks it may take too long to service a request
  - Insert too many wait cycles, or better ...
  - Suspend the request (via SPLIT response)
    - Arbiter informed via bus
    - Corresponding master is suspended
    - Other masters can access the bus
    - Later slave informs arbiter that it is ready.

## Transfer Cancellation



## Split response

- Cycle i
  - Master M drives address A on bus
- Cycle i+1
  - Slave S thinks it can take too long to service A, issues SPLIT response
  - Arbiter snoops on SPLIT response, records **current master**.
  - Issued in (i+1, i+2) to kill already initiated transfers
- Cycle i+2
  - Arbiter disables bus access to **current master**. Others can now access the bus.

## Mechanism of split

- Slave issues split to master m.
- Arbiter records that m has been split
  - Even if m requests bus access during this period, it is not even considered by the arbitration algorithm
- Slave informs arbiter later that it can now service m.
- Arbiter now enables potential access
  - m may still not get bus access immediately.

## Model Checking

- Developed a formal specification of the protocol.
- Various kinds of components
  - > 1 Masters
  - Slave(s)
  - Arbiter
- Bus interface of each component modeled as a finite state machine.
- Protocol = Synchronous composition of these FSMs
- Model check temporal properties (e.g. non-starvation) using **Cadence SMV** tool.

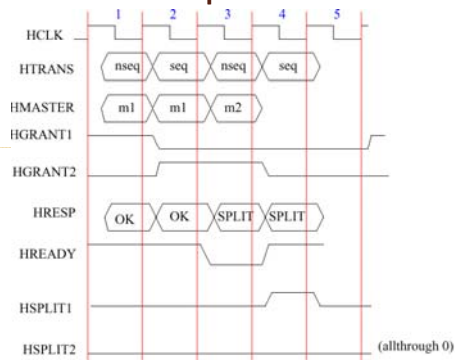
## Checking for no-starvation

- Property in Temporal Logic LTL
  - $G (HBUSREQ_m \Rightarrow F HGRANT_m)$
  - $HBUSREQ_m$ : Master m requests bus access
  - $HGRANT_m$ : Master m granted access by arbiter
- Do not consider starvation introduced by incorrect implementation of the cores, e.g.
  - The arbiter is unfair.
  - A slave is not guaranteed to service every split transfer eventually.

## Checking for no-starvation

- Check for starvation introduced by corner cases of the protocol
- However, do not fix any implementation of the cores
  - Do not code up a fair arbiter.
  - Do not code up a slave which is guaranteed to eventually service split responses.
- Use assertions to denote these features
  - using fair, slave\_live prove no\_starve;
  - assume fair, slave\_live;
- Restrict model checking to executions satisfying these assertions (possible in Cadence SMV).

## Counter-example



## Explanation of counterexample

- Cycle 1: Master m1 transmits, slave gives OK response later.
- Cycle 2: Master m1 transmits again, slave's SPLIT response in cycle 3. Master m1 is done.
- Cycle 3: Master m2 starts to transmit, slave's SPLIT response reaches. Slave records that m1 has been split. Arbiter snoops on bus, records current master m2 and marks m2 as split.
- Cycle 4 onwards:  
Arbiter waits for slave to inform that m2 can resume participation in bus contention. But slave never informs so since it does not have any record of m2 being split.  
**Master m2 is starved from bus access forever.**

## Conclusion

- Case study in model checking of a bus protocol with non-trivial data transfer features
- Interaction of these features (Pipelining + Splits) leads to a corner case starvation scenario.
  - Essentially an incompleteness in the spec.
- Source of starvation was suspected via human understanding of the protocol.

## Conclusion

- Model checking effort was taken up to:
  - verify our suspicion.
  - Find a detailed counter-example trace.
- References
  - SMV Code (feel free to take a sneak peek)
    - <http://www.comp.nus.edu.sg/~abhik/software/amba/ahb.smv>
  - A relevant paper
    - <http://www.comp.nus.edu.sg/~abhik/pdf/date03.pdf>

## Exercises on Past Lecture

- *Transition Systems and Kripke Structures*
  - Ex 1. Consider the following program with two processes, which are composed **asynchronously**. Assume that initially  $x = y = 0$ , and each assignment is executed atomically.
    - $x := 1$                        $a := y$
    - $y := 1$                        $b := x$
  - What are the possible values of  $a$  and  $b$  when the program terminates? For each of these possible values construct a trace that will generate it.

## Exercises on Past Lecture

- *Transition Systems and Kripke Structures*
  - Ex 2. Two students are taking the CS4271 exam. We must ensure that they cannot leave the exam hall at the same time. To prevent this, each student reads a shared token  $n$  before leaving the hall. The shared token is an arbitrary natural number. The global state of the system is given by  $\langle s1, s2, n \rangle$  where  $s1$  and  $s2$  are the local states of students 1 and 2 respectively;  $s1 \in \{in, out\}$ ,  $s2 \in \{in, out\}$ . The pseudo-code executed by the two students is given below. The two student processes are executed asynchronously. Every time one process is scheduled, it atomically executes one iteration of its loop. Initially  $s1 = in$  and  $s2 = in$ .
 

```

do forever{
  if (s1 = in & n is odd) {s := out}
  else if (s1 = out) {s1 := in; n := 3*n + 1}
  else {do nothing}
}
          
```

```

do forever{
  if (s2 = in & n is even) {s2 := out}
  else if (s2 = out & n is even) {s2 := in; n := n/2}
  else {do nothing}
}
          
```

## Exercises on Past Lecture

- *Transition Systems and Kripke Structures*
  - Ex 2 (Continued)
  - To represent the above program as a Kripke Structure, we need to introduce atomic propositions corresponding to  $s1, s2$  and maintain approximate information about  $n$ . I suggest using an atomic proposition  $pn$  which is true when  $n$  is even and false otherwise. Thus, the global state will be given by the valuation of three atomic propositions. Draw the Kripke Structure for the above program.