Effective Verification of ARM SoCs

Ron Larson, Macrocad Development Inc.
Dave Von Bank, Posedge Software Inc.
Jason Andrews, Axis Systems Inc.

Overview

System-on-chip (SoC) products are becoming more common, but design and verification of these chips is becoming more difficult. While the effect of Moore's Law has resulted in increased circuit density and speed, the development risks are increasing at an alarming rate. Even in the face of such risk, the benefits of increased integration are necessary to remain competitive in many markets, such as consumer electronics, to drive down system cost. For today's design engineers, ten million gate designs have turned Moore's Law into Moore's Curse.

Schedules for chip development have not significantly increased to account for larger, faster, and more asynchronous designs. Traditional verification tools for ASIC development have proven insufficient for complex chip design. The inclusion of microprocessors into designs requires engineers to do more verification at the system level. This necessitates the inclusion of software into the design verification process. While the execution of embedded software is necessary as a verification step it is also not sufficient as a verification method.

To address the growing verification requirements multiple techniques must be deployed to address specific verification concerns. A verification environment with a mix of tools and technologies for performing mixed-language logic simulation, simulation acceleration, in-circuit emulation, HW/SW co-verification, transaction-level testbenches, and assertions is required to successfully minimize risk and optimize project schedule. A complete verification methodology was proposed in [1]. This paper provides more detail by illustrating the combined value of co-verification and constrained random testing to maximize verification coverage for designs using AMBA AHB. It also demonstrates a unique verification flow that can be achieved by cooperation between co-verification and testbench.

Managing the Iterative Loop

Engineers verify their design by traveling around an Iterative Loop as shown in Figure 1.

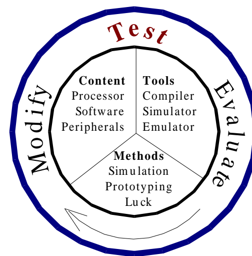

Figure 1
The SoC Iterative Loop

This Iterative Loop consists of three sequential tasks; Modify, Test and Evaluate. Managing size

1 See "Transaction-based methodology supports HW/SW co-verification" by Jason Andrews, Axis Systems http://www.eedesign.com/features/exclusive/OEG20030228S0061

of the Iterative Loop Circumference, measured in time, is the key to managing throughput. Traveling around the Loop, the designer uses tools and methods applied to the design content to determine if a function being evaluated meets the requirements of the design specification, or if the designer's assumptions need to be adjusted. The more automated the verification process becomes the shorter the time to complete the Loop. This can take the form of assertions, reference models, and formal methods. The inclusion of specification details can point out problems earlier in the test phase, but it can also make the Iterative Loop Circumference larger.

The Test phase of the Loop provides the largest contribution of overall time. If the content increases, the Loop Circumference increases. If simulation tools are slow the Loop Circumference will increase. If verification methods are inflexible and not well suited to the particular task, the Loop Circumference will increase. The Test phase is where tools and methods can have the biggest impact on the Iterative Loop Circumference.

The concepts of the Iterative Loop are best illustrated by example. Figure 2 shows a block diagram of a design in which an ARM processor communicates with memory and Ethernet MACs using an AHB bus interface.

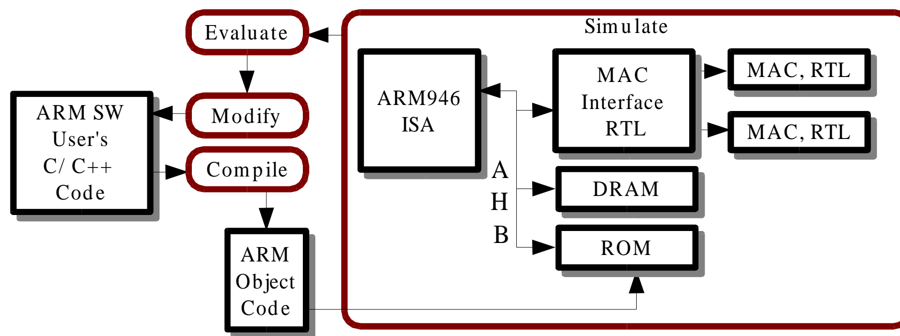

Figure 2
ARM946 Design Example

The processor model is a design-signoff model (DSM) for the ARM946E-S and all the peripherals are included as RTL models. Running software programs on the DSM is practical only for small programs and short tests. The resulting performance will make this method impractical for software designers desiring higher performance and better debugging. Software designers are primarily interested in the processor and its execution of instructions. Hardware engineers also suffer from the overhead associated with a full-functional model when they are mainly interested in the bus protocol and bus transactions. Clearly, better solutions are needed for both hardware and software engineers to be productive.

Co-Verification and Test Generation

The solution for the software engineers is HW/SW co-verification. Co-verification allows software engineers to achieve higher performance, source-level debugging, and a method to test hardware/software interaction. Early software testing improves confidence in the hardware and software quality and saves time in the project schedule. One of the key metrics of hardware design quality is its ability to run the embedded system software. Once projects discover the benefits of co-verification they artificially believe that if the design runs all of the diagnostics, boots the operating system, and runs applications, it must be bug free. Unfortunately, software can change at any time. There is also no guarantee that software exercised all of the features and functions of hardware.

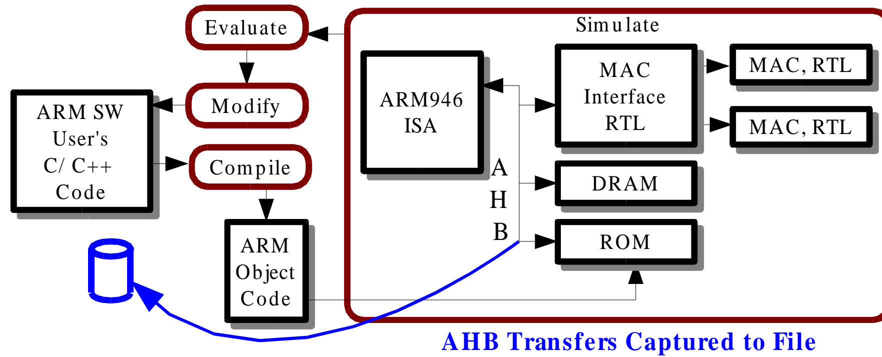

FIGURE 4
EXAMPLE OF AHB TRANSFER SEQUENCE EXTRACTION

The best verification is provided by a combination of running software tests augmented by traditional testbenches used to verify the complete bus interface. One of the major stress points in hardware verification is the manual work required for testbench creation and stimulus generation. This paper shows how co-verification combined with stimulus generation provides increased verification value.

Using the ARM example and co-verification it is possible to run software programs on an ARM946 co-verification model and capture the resulting sequences of AHB transfers created by these software programs. Figure 4 shows the extraction of AHB transfers using a co-verification model.

The AHB transactions are written in the form of a command file that is can be used to drive a synthesizable bus functional model of the AHB protocol. Bus functional models have long been used in functional verification to abstract microprocessor operation and for other I/O and communication interfaces such as PCI and Ethernet. Synthesizable models are crucial for simulation acceleration and emulation applications to obtain the highest performance verification environment.

The AHB command file contains AMBA AHB transaction commands, sideband commands, and delay commands shown in Figure 5.

```
    # AHB commands output from mdi7 at Thu Oct 23 11:43:06 2003

# Commands with an asterisk ('*') following CMD field were not
# specified explicitly, but rather implied by other commands.

# Command Formats:
          # DELAY                                  TICKS                #    CMD
# SET/GET                     INDEX   RDATA/WDATA  DMASK    #     CMD        DATA READ   ABNORM MISCOMP
# HWRITE HBURST HSIZE HPROT    HADDR  HRDATA/HWDATA DELAY   #     CMD  BEAT  DATA READ   ABNORM MISCOMP
     1      0      1      8  00000410    41451b05     0     #    1399    -          -       0       -
     0      0      2      8  00000400    26918153     0     #    1400    -   00000000       0       0
    d0                                   00000002           #    1401    -          -       0       -
     0      0      2      1  00000400    26918153     0     #    1402    -   00000000       0       0
    d0                                   0000000f           #    1403    -          -       0       -
     1      0      1      8  00000420    770c6f4a     0     #    1404    -          -       0       -
```

FIGURE 5
EXTRACTED AHB TRANSFERS IN AHB COMMAND FILE

The AHB transfer descriptions in this file have all the information needed to reproduce the sequence of operations as seen by the hardware design without requiring a full processor model.

In this application co-verification is used to provide stimulus generation for a bus functional model.

If a software test results in a possible hardware problem the hardware designer can "replay" the transactions and debug without the use of co-verification and software debuggers. The sequence of AHB transfers will also run faster because the ARM core is replaced by a bus functional model. Figure 6 shows the processor model replaced with the synthesizable AHB BFM with the stimulus coming from the AHB command file.

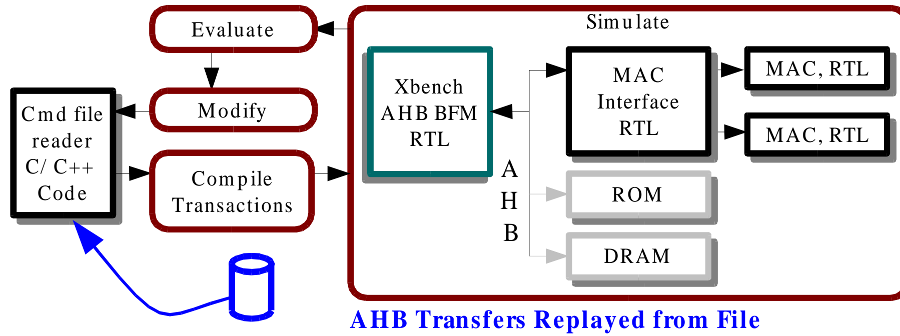

FIGURE 6
TRANSFER REPLAY ON ARM946 EXAMPLE

The key to such a methodology is to maintain signal compatible Verilog modules for each representation of the ARM core. The synthesizable BFM provides an AHB master interface, but also the other supporting signals so that it fits seamlessly into the simulation environment.

For hardware engineers, AHB transfers are an important point of reference. Once the full functional processor model is replaced with the BFM the Iterative Loop is manageable once again. The hardware designer replicates the sequence of AHB transfers and is able to analyze the results. This is done without the hardware designer leaving the logic simulation environment.

Random Test Generation

Easily replicating transaction sequences for software tests is useful, but the hardware designer has a larger problem. In order to obtain increased confidence more verification must be done and functional coverage must be measured. The goal is to easily generate stimulus and run the stimulus at high speed. Again, transaction based verification is the best alternative to do this. Transactions encapsulate the necessary address, data, and control information.

Figure 7 shows transactions generated using a C program. With this architecture tests can be directed, random, or a blend of both. The transaction files produced by software tests can be combined with random test generation to construct a comprehensive stress test for the hardware design. The solution benefits from synthesizable bus functional models that can also be used for simulation acceleration and emulation. The ARM subsystem can be combined with other synthesizable models and in-circuit interfaces to create a complete verification environment.

A common way to increase design confidence is to run random test patterns. This can be done in addition to directed tests or interlaced with the directed testing for a more realistic simulation. Random test generation must be automated and make use of constraints to keep the tests relevant. Random test generation is only useful if the effectiveness of the tests can be measured and quantified.

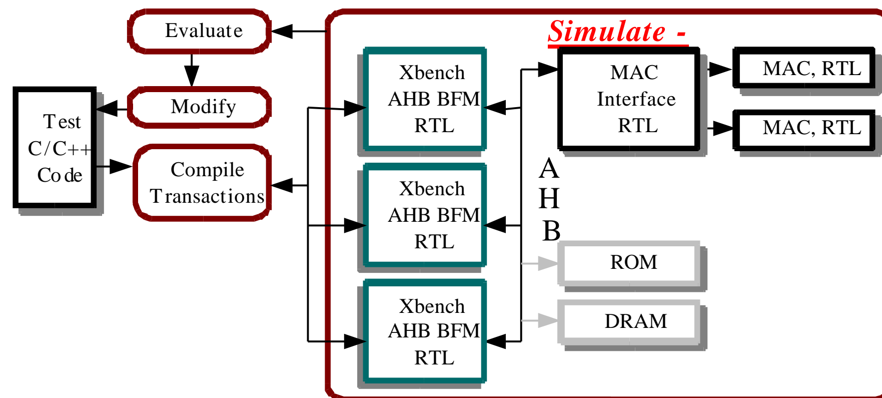

FIGURE 7
EXAMPLE OF AHB ACCELERATED ITERATIVE LOOP

The C-API of the synthesizable AHB model can provides the necessary measurement. Engineers can use these metrics to adjust the random generation constraints and rerun the simulation to check the coverage improvement. The tests can be written as multi-threaded C or C++ programs.

Coverage Analysis

Looking back at the ARM example it is necessary to examine how effective these methods are in exercising the AHB bus protocol. To do this analysis an AHB transaction model, an AHB coverage model, constraints applied to the AHB transaction model, and a reference model are introduced.

AHB Transaction Model

AHB axis	Description
HWRITE	1 bit
HBURST	3 bits
length	10 bits
HSIZE	2 bits (byte, halfword, and word accesses supported)
HPROT	4 bits
HADDR	32 bits
HDATA	32 bits (only 32 bit data buses supported)
pre delay	32 bits
beat delay	64 bits (4 bits for each of 16 data beats in a burst)
post delay	32 bits
total	212 bits

TABLE 1
TRANSACTION MODEL

The AHB transaction model represents a single AHB transaction. To represent all possible AHB transactions and the possible delays before, during, and after each transaction requires a ten-dimensional model consisting of the axes shown in Table 1. Since the ARM946 uses only a 32-bit data bus 212 bits total are required. 84 of these bits are specific to the protocol and the other 128 bits are specific to the delay values.

AHB Coverage Model

The AHB coverage model is a subset of the AHB transaction model. The coverage model identifies the portions of the transaction model of most interest for specific hardware tests. Four axes of the transaction model are used directly: HWRITE, HBURST, HSIZE, and HPROT. We

collapse the three axes associated delay possibilities into a single axis with a single bit of data. The resulting five dimensional model uses eleven bits and is shown in Table 2.

AHB axis	Description
HWRITE	1 bit
HBURST	3 bits
HSIZE	2 bits
HPROT	4 bits
any delay	1 bit
total	11 bits

TABLE 2
COVERAGE MODEL

AHB Transaction Constraints

Before generating random tests the axes of the AHB transaction model are constrained to deliver more focused tests. The transaction constraints used for the ARM946 example are shown in Table 3.

Hwrite	Hburst	Hsize	length	Hprot	Haddr	Hdata	Pre Delay	beat Delay	post Delay	weight
0-1:1	0,1:1 2-7:4	0:2 1:1 2:1	1-256:1	0-f:1	0-ffff:1	0-ffffffff:1	0-1:1	0:1	0:1	1
0-1:1	0:1	0,2:1	0:1	0:1	fffe0000-fffe2000:1	0-ffffffff:1	0-1:1	0:1	0:1	1

TABLE 3
ARM946 EXAMPLE CONSTRAINTS

There are two sets of constraints, indicated by the two rows in the table. The first row corresponds to the axes of the AHB transaction model and the last column provides a weight for each set of constraints. Each element in the table is a constraint specifying how a random value is chosen from the domain for an axis. Each constraint takes the form of one or more fixed values, ranges, or sets – followed by a weight (indicated by the integer following the colon). The weights are used to pick a particular random value from the domain for the axis. Hence, there are two levels to the randomization. First, a set of constraints (a row in the table) is chosen based on the weights in the last column. Second, a random value is chosen for each axis (the first ten columns in the table) based on constraints supplied for that axis. This two-level constraint mechanism allows, for example, the random values generated for one range of addresses to be different than those for another range of addresses.

In Table 3 shows two sets of constraints, each having a weight of 1, as indicated by the last column. This means that they are each equally likely to be chosen when generating a random transaction. The table shows fixed value, range, and set constraints. All three of these are shown in the HBURST column. In the first row, there is a set of two values "0, 1" with a weight of "1" and a range of six values "2-7" with a weight of "4". Thus, a value is chosen from the range four times as often as one is chosen from the set. When the set is chosen, "0", for a single transfer, is selected with the same probability as is "1", for an unspecified-length burst. When the range is chosen, one of "2", "3", "4", "5", "6", and "7", for the different flavors of fixed-length incrementing and wrapping bursts, is selected. Each value in the range is selected with the same probability. In the second row, "0" with a weight of "1", indicates that a single transfer is always used when the set of constraints for the second row is chosen.

SoC Data Reference Model

Self checking tests are the best way to automate verification. The simplest way to do this is to use system data to check the validity of the design. To accomplish this, a reference model for the

design is used. When a read occurs, the data returned from the reference model is sent to the BFM as the expected data during the read. The comparison is done in the BFM to maximize performance in simulation acceleration and emulation applications. A single bit reflecting the comparison is returned to the test program. If no reference model is available, the comparison can be turned off.

Coverage Results

Next, the results are analyzed for two ARM SoC designs. We will examine both the ARM946 example previously presented and an ARM926 example shown in Figure 8. The ARM926EJ-S CPU core uses two AHB interfaces, one for instructions and one for data.

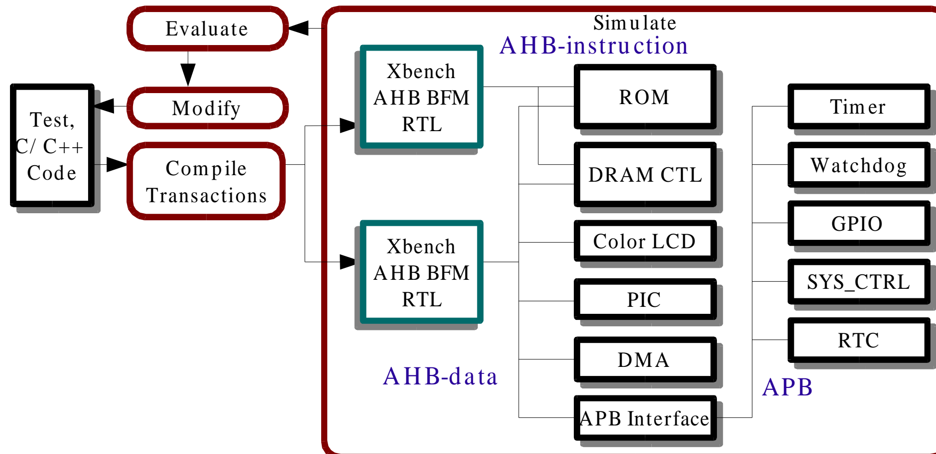

FIGURE 8
ARM926 EXAMPLE

Table 4 details the coverage metrics achieved for each of the designs. For each AHB interface a software program was run using co-verification, and a test program generated using random methods was run with using the synthesizable AHB BFM. Each method has its own set of AHB transactions that are generated. There is a row in the table for each sequence of commands. The columns provide information about the sequence: the number of commands in the sequence, the percentage of 1-transaction sequences covered, and the percentage of 2-transaction sequences covered. For our AHB coverage model there are 1,536 possible 1-transaction sequences (1536 different transactions) and 1,536 * 1,536 = 2,359,296 possible 2-transaction sequences.

SoC Design AHB Interface	Tool	Commands	1-Transaction Sequence Coverage	2-Transaction Sequence Coverage
ARM946	Co-verification	665,389	2.018%	0.006%
	AHB BFM	6,500	85.221%	0.198%
ARM926 I	Co-verification	1,388,933	0.260%	> 0.000%
	AHB BFM	10,000	94.140%	0.294%
ARM926 D	Co-verification	1,394,093	0.977%	0.003%
	AHB BFM	12,000	96.549%	0.347%

TABLE 4
ARM946 AND ARM926 COVERAGE RESULTS

The results are dramatic. The AHB BFM random sequences accomplish at least 40 times the 1-Transaction Sequence coverage using only 1% of the transactions for each of the designs. It shows that software diagnostics are useful for testing specific hardware operations, but are not

focused on exercising the AHB protocol.

AHB protocol coverage is an important measure of verification effectiveness. For example, if our AHB coverage for the ARM946 indicates that there is low coverage due to the burst length not being thoroughly exercised, there might be some packet lengths for the Mac which is not well tested either. In the ARM926 example, a split bus is used. The instruction AHB bus in the ARM926 has reduced coverage compared to the data AHB bus. Clearly, designers can benefit from using a BFM to drive the AHB busses rather than relying completely on software diagnostics.

The coverage for 2-transaction sequences is much lower in all cases. This is because there are over three orders of magnitude more possible 2-transaction sequences than 1-transaction sequences. Measuring the sequences of transactions is important because of latencies, recovery times, data availability, previous state machine conditions, and the interaction with bus stalls. Each functional block interacts with internal and external events. Exercising these interactions is critical. With randomly generated tests, having a measure of this coverage is a key to effective verification. Though it isn't shown in the table, 2-transaction sequence coverage of over 10% was achieved for the ARM946 design using a one million transaction sequence.

In addition to the summarizing coverage information presented in the table, the C-API for the AHB BFM provides raw coverage information that can be accessed by the C test program. This program can quickly look for important transactions or crucial sequences. If they are not covered directed tests can be easily added.

Conclusion

Improved system verification approaches are needed to better address increasing complexity and decrease risk. Mitigating risk is best accomplished by increased verification. This requires each engineer to be effective in maintaining a small circumference on the Iterative Loop. To do this, complementary design methods must be used. Interoperable tools in the area of HW/SW co-verification and test generation are needed to improve overall verification.

In this context, transaction sequence coverage was demonstrated as a useful measure that applies to both co-verification used by software engineers and test generation used by hardware engineers. The coverage achieved from both methods can be aggregated to ensure thorough verification and to eliminate unnecessary duplication.

Constrained random test sequences can be quickly created by understanding the transaction model and constraint specification. By defining a coverage model verification goals can be established.

This research has concluded that the use of software diagnostics and other ARM test code is not sufficient to verify ARM SoC designs. While co-verification is useful for development and debugging of functional tests, more is needed to increase design confidence. The interoperability between ARM co-verification and a synthesizable AHB BFM was demonstrated and impressive coverage results for random test generation using the C-API for the AHB BFM were presented.