System Model – A Testbench Library Component Aided for Emulating User Interaction

Hussain Wadia

Advanced Micro Devices, Inc.

2485 Augustine Dr, Santa Clara, CA 95054, USA

Hussain.wadia@amd.com

1. Abstract

To perform exhaustive verification of CPU cores, we need capability in assembly tests to control external stimuli and monitor internal system events. The existing solutions in the industry are mainly ad-hoc and dedicated for a type of processor and there lacks such an instruction-set-independent testbench component to work within different types of processors.

For this purpose, we have created a universal testbench component, which provides a software interface and acts as a middle man to create multiple scenarios and use cases. The primary use-case of this component is to interpret "special" stimulus events as triggers and inject the corresponding asynchronous events into the processor cores. Its capability has also been extended to monitor internal events and provide feedback to tests.

2. Introduction

In this paper, we demonstrate the proposal of such a testbench component with an implementation for X86 type of processor which is easily extendable to other projects. The component known as System Model has the capability of driving events such as interrupts, reset, error injection, etc. It provides end of test events with appropriate status which serve as self-checks for tests. It can monitor system events (e.g., Halt, C-State changes) and memory/IO accesses and provide count and cycle information to the test. It has been developed in C++ language and designed to fit in various environments, such as standalone instruction set simulator, VCS co-simulation, emulation, and so on.

The paper proposes a generic component which the testbench can interface to through a well-defined set of APIs. This helps make the component truly ISA-independent and easily integrated in several environments. It also explains the specification for the components which serves as a programmer's manual for the test authors.

3. System Model – Component

System Model was developed to allow assembly test patterns with a mechanism for injecting desired stimulus at programmable times or intervals. The idea is to create an abstract testbench component which can be ported in multiple environments and used with variety of processors.

System Model owns a portion of the memory map. Assembly tests communicate through read and write accesses to the System Model owned memory. The System Model as such does not have knowledge of the nature and location of the memory. It creates what we refer to as ports/registers against address offsets. The addresses are governed by the specification agreement of the System Model. Each project can choose to define the location and type of System Model memory. For our implementation used in X86 processors, we chose to place the System

Model in the memory-mapped I/O (MMIO) region.

In a nutshell, System Model's working can be described as follows,

Assembly tests programs the System Model registers. This causes the System Model to generate an output event after a specified system event and/or delay. The output event could represent variety of external events to a processor for example interrupts, reset, error injection. In addition to this System Model can store information of system events and provide feedback to the assembly tests. It can also be used as a more generic 'middle man' to pass messages back and forth between assembly tests and other testbench components through scratch ports.

The block level diagram representing the System Model is shown in *Figure 1*.

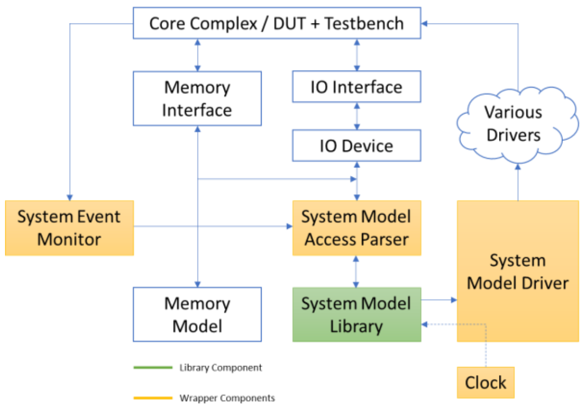


Figure 1

There are two parts of the System Model. The central library component (highlighted in green) specifies the registers as per the System Model specification, schedules and generates events. This library component is portable and can be extended for more than one type of specification. The wrapper components (highlighted in orange) are per testbench where the library is used. These are expected to be thin layers which detect the System Model register accesses and other special events, interface with the library through a set of defined APIs and drive the events accordingly in the testbench.

4. System Model Library

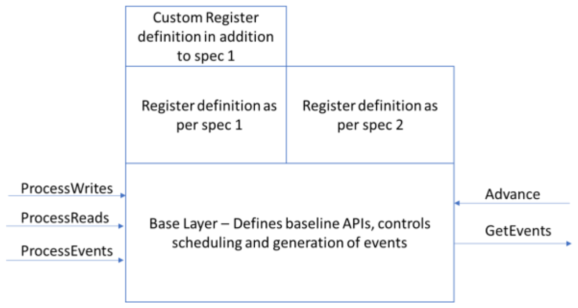
The System Model library is the main crux of this verification infrastructure. It defines all the registers which are owned by the System Model, has knowledge of how to handle all the triggers and what events to generate. The registers are defined as per the specification and the library can be easily extended to multiple specifications. The library has capability to schedule the events internally.

The library works as follow:

1. Assembly tests configure the registers. System Model library identifies the new configuration and waits for specific triggers.
2. Wrappers notifies the triggers to the System Model library.
3. Library reacts to these and schedules internal events.
4. State machine of System Model is "advanced" by the wrapper.
5. System Model generates events at appropriate time ticks.
6. Wrapper drives these events back to testbench.

The triggers to the System Model could be a variety of events happening in the system, for example Halt, C-State changes, memory/IO accesses.

The library is structured as a hierarchy with base class defining the interface APIs and the state machine while the children implement the actual specification. The skeletal architecture is shown in *Figure 2*.



Custom Register definition in addition to spec 1
Register definition as per spec 1
Register definition as per spec 2
ProcessWrites
ProcessReads
ProcessEvents
Base Layer – Defines baseline APIs, controls scheduling and generation of events
Advance
GetEvents

Figure 2

The base class defines the interface APIs to the consumer. It consists of a list of registers which will contain all the registers defined by the child classes. It will perform the basic Read/ Write functionality, determine if it is a System Model or memory access and maintain the state machine to schedule events. Based on the triggers, internal events are created with timers and queued up. Each Advance of the System Model ticks these counters and determines when these internal events are ready to send an output.

The APIs proposed in the implementation are as follows

- int ProcessWrites(uintptr_t address, uintptr_t data, uint8_t accSize);
- int ProcessReads(uintptr_t address, uintptr_t data);
- int ProcessEvents(uint8_t event);
- void Advance()

The children of this class will define the actual registers and handler functions associated with each of the registers. As shown in the diagram, multiple children can exist per specification. One type of processors can have one baseline specification which can be further extended, if required for newer revisions. The design is kept modular to pick and choose as per requirements.

The register classes encapsulate attributes - WriteData, ReadData, Readable, Writable, Address, and Name. The library defines various behaviors as well such as maintain different data for writes and reads, clear on reads, storing a stack of data, etc.

The output of the System Model is a simple packet like object which contains the relevant information required to be driven into the testbench. The fields are as follows

- Event type: The library will define all the types of events which are to be supported. The important events in our implementation were Interrupts, Reset, Debug Request (DbReq), Error Injection, etc.
- Event namespace: The Event types are decided by various derived child classes. This info will allow the child classes to freely define the event types. The consumer / driver will have to appropriately decode the event type based on this field.
- Target CPUs: All the CPUs to which the event is to be targeted.
- Source CPU: This will represent the source of the event.
- Additional Data: This will be event specific list of values.

5. System Model Wrapper components

There are 3 wrapper components need to be implemented by testbench to interface with the System Model library. These will be project dependent as they need to understand the project specifics.

The Access Parser is meant to snoop all data traffic coming out of the testbench and pass it to the library. For our implementation and since we chose the registers to be IO ports, we have an IO device as an Access Parser which sends requests to the library.

The System Events Monitor will have hooks across the testbench and DUT to observe a variety of events and notify the library.

The Driver will need to Advance and collect the events generated by the Library and appropriately drive the events back in the testbench.

6. System Model – Specification

System Model is developed for a specification which acts as a programmer's guide to configuring the System Model library. As mentioned before assembly tests configures registers. These registers are mostly 32-bit wide for convenience, but they can be of any size. Our implementation of System Model has defined various type of registers. This paper does not describe the exact specification but provides an idea of the various types of registers. The most important of the registers are Interrupt Control Registers (ICRs) which provide capability for tests to generate external events after specific trigger and delay. These are explained in detail in the next section.

7. ICR registers

ICR or Interrupt Control registers generate specific events which are driven into the core complex. The common list of ICRs are (but not limited to) INTR, NMI, SMI, INIT, DbReq, Reset, etc. These events are generated based on certain triggers which are defined by Trigger Management Control register (TMCR). The TMCR are configured with these triggers which could be either in form of a memory access or a system event (e.g., Halt, C-State change). Whenever the trigger/event is received, the corresponding TCMR will fire the ICRs configured for that TMCR. Accordingly, the event of the ICR will be queued up.

A TMCR is defined by a group of 3 registers. In our implementation we defined 7 sets of TMCRs for ease of test writers.

Trigger_Control: This register will map an event to this TMCR. The fields are as follows

31-9	**8**	**7-0**
Reserved	Enable	Trigger ID

- Trigger ID - A 8-bit id which defines the trigger.
- Enable - Whether this TMCR is enabled.
- Reserved bits can provide flexibility to add more control bits to this register in the future.

Trigger_Address: This register will allow the test to configure a physical address for a memory/IO access trigger.

Trigger_Count: This register will hold the count of the times the trigger/event occurs. This provides useful feedback to the test.

An ICR is defined by a group of 3 registers

ICRControl_{Name}:

31-15	**14-12**	**11**	**10-8**	**7-3**	**2**	**1**	**0**
	End Trigger		Start Trigger		Rand	Repeat	Active

ICRDelay_{Name}:

31-0
Delay

ICRTarget_{Name}:

31-0
Target

- Active – Defines whether ICR is enabled.
- Repeat - ICR fires in cyclic manner. If it is zero, ICR will be de-activated after firing once. Stimulus will have to re-enable it. If it is repeat mode, it will generate an event and then wait for trigger & delay to generate once again.
- Rand - When this bit is set delay is chosen randomly.
- Start Trigger - Indicates which TMCR to wait for before generating the corresponding event. 0 indicates no trigger, the "delay" counter is started immediately.

- End Trigger - Indicates which TMCR to wait for before clearing the corresponding event. This may not be applicable for all ICRs (e.g., interrupts). 0 indicates after certain defined time units.
- ICR delay - Number of time units to wait after the TMCR has fired to generate the corresponding event.
- ICR target – Configures the CPU target of the ICR event. Essential if test wants capability of targeting events in multi-threaded or multi-processor events.

8. Evaluation and Use-cases

At AMD, the System Model is extensively used to verify several complex architectural and micro-architectural features.

The concept has been existing for long, but the implementation proposed in this paper abstracts out the common functionalities from the test bench into a library component. The wrapper–library style design allows the component to be easily integrated in various environments. It helps make it project agnostic as its implementation is dependent on the specification. In our current implementation, we chose to place the System Model in the MMIO space while project could simply move it to a non-cacheable memory region. This design also allows project to extend the specification to suit their needs.

This design of the System Model library allows to write unit-test which aids in quick development time. In our implementation we created a basic shell around the System Model library which invokes the APIs in an appropriate fashion which would help test various scenarios.

At AMD the common cases where System Model is being used are as follows.

- Wakeup processor from halt.
- Wakeup processor at various C-States. Helps in verifying several power management scenarios.
- Random injection of interrupts and reset. This is particularly useful in random tests, where a specific ICR is programmed in repeat and random mode at the beginning. This helps cross interrupts with several instructions/scenarios.
- Precise error injection.
- Self-checks on complex instructions, for example Cache-Management instruction evicted a cache line. Test would setup a TMCR to monitor writes to an address, execute the instruction and then query the System Model for the number of times the trigger occurred.
- Provide means for assembly tests to end with a status code.

9. Conclusion

This paper attempts to describe a generic testbench component which provides an aid to assembly tests to control external events. We implemented this component to suit our X86 microprocessor projects. However, it was developed to scale and be compatible for any type of microprocessor. The wrapper–library style design allows the component to be easily integrated in various environments. The concept of System Model has been used for various generations of processors at AMD. It has been useful to verify difficult to check stimulus as the component sits at a higher level in the testbench and is capable of monitoring events happening in the system. It has been used to verify various power management, interrupts, and error injection scenarios.