International Journal of Research in Engineering, Science and Management
Volume 5, Issue 10, October 2022
https://www.ijresm.com | ISSN (Online): 2581-5792

Design and Verification of a Bit Movement Engine

Krishnakanth Katteri Mahadeva Murthy*
Department of Electrical Engineering, San Jose State University, San Jose, USA

***Abstract*: An SoC (System on Chip) comprises many functional units. These functional units consist of many lower-level components such as comparators, shifters, adders, etc. In this paper, we talk about one of these functional units called bit movement engine. A bit movement engine moves chunks of data from one memory location to another. Bit movement engine is designed to move bits starting from any location in the memory block and it can move a variable number of bits. At the core of the bit movement engine is a funnel shifter. Bit movement engine communicates to memory through an AHB (AMBA High-Performance Bus) fabric. Any functional unit developed requires a thorough verification to certify that the functional unit behaves as expected. The functionality of the bit movement engine is modeled in the test bench and the behavior of the bit movement engine is checked. The bit movement engine is verified using UVM (Universal Verification Methodology).**

***Keywords*: SoC, AMBA, AHB, UVM, Bit movement engine, IP (Intellectual property), GPUs (Graphics Processing Unit).**

1. Introduction

SoC consists of many IPs. A bit movement engine is one such IP that moves bits of data on a word-oriented bus. Bit movement engines can move any number of bits to any non-overlapping location in the memory. Since this engine can operate on a bit level and can move variable lengths of data, the design is quite complex. GPUs that do a massive amount of data manipulation and memory alteration have IPs that do a similar job as the bit movement engine [1].

On many occasions, IPs have interfaces that are different from the interfaces of the IPs they talk to. In these cases, bridges are used to convert one protocol to another. This enables the reuse of IPs. Bit movement engine uses a bridge to convert its native interface protocol to AHB protocol [2]. AHB bridge further talks to a fabric. Fabric is typically connected to multiple masters and slaves. Fabric is responsible for arbitrating requests from masters and slaves. In this paper, Fabric acts as a placeholder for connecting masters and slaves.

Bit movement engine is a complex design and requires high-quality verification. High-quality verification means driving all possible types of input, checking the bit movement engine's output against the checker's expected data, collecting coverage data to analyze holes in the stimulus, and filling the holes. Doing this would give confidence that the design is solid.

Verification is carried out using UVM [3]. UVM is a framework based on System Verilog. It provides a standard way to build a test bench that's reusable.

2. Design Under Test

Bit movement engine implements a state machine [4]. This state machine has 10 states and each state performs a unique set of operations. Further, the bit movement engine has two interfaces: master interface and slave interface. The master interface is used to talk to the memory slave for reading and writing data. The slave interface is used by the test bench to write data to registers that instruct the bit movement engine about the next operation that it has to carry. Table 1 talks about the bit movement engine's interface signals.

Table 1
Bit movement engine interface

sRW	Read write register interface for slave
sSEL	Indicates slave selected
sADDR	Indicates slave address
sWDATA	Slave write data
sRDATA	Slave read data
mADDR	30 bits master word address
mWDATA	Write data from master interface
mRDATA	Read data to the master interface
mRW	Read write register interface for master
mREQ	Data transfer request from the master
mHOLD	Hold the data transfer in the pipeline
DONE	Indicates operation is completed

Table 2 talks about the control registers in the bit movement engine. These registers are written by the testbench through the AHB fabric, AHB bridge, and the slave interface of the bit movement engine. The source address is decoded from register 0 and register 1. Block length is decoded from register 1. The destination address is decoded from register 2 and register 3. Register 4 is written by the testbench to start the bit movement engine.

Figure 1 shows the state diagram implemented in the bit movement engine. In the first state which is the 'address decode' state, the testbench writes to the bit movement engine's registers. This state waits for the start bit to go high. One start bit is written, the state machine enters the 'address computation' state where the source address, destination address, and block length are decoded. In the 'read to FIFO' (First In First Out) state, 4 double words are read starting from the double word source address is pointing to. Next, the state machine moves to the 'compare offset state' where the source address and destination offsets are analyzed to determine the need for shifting the source data to align with the destination offset. Also, the block length with the destination offset is

*Corresponding author: krishnakanthkm@gmail.com

Table 2
Bit movement engine registers

Addr	Function	Comments
0	Source addr low	Lower 32 bits of a 'bit' address
1	Source addr high, Block length	Upper bits of 'source bit address' (5 bits) Block length (27 bits) in 'bits'
2	Destination addr low	Lower 32 bits of a 'bits' address
3	Destination addr high	Upper 5 bits of 'destination bit address'. Lower 27 bits are unused.
4	Start and status	Upper 29 bits unused. Error, busy, START. Write START bit to a '1' to enable operation

Table 3
List of AHB signals

Signal	Functionality
HCLK	Clock signal for the bus
HRESET	Active low reset signal for the bus
HTRANS	2-bit signal indicating type of transfer
HWRITE	Indicates whether it's a read (high) or write (low) operation
HBUSREQ	Request to access the bus
HGRANT	Indicates ownership of the bus
HREADY	Indicates status of the transfer
HWDATA	Transfers data from master to slave during the write operation
HADDR	32-bit address bus
HRDATA	Transfers data from slave to master during the read operation
HSEL	Indicates which slave is selected from the current transfer

analyzed to determine if the source data fits in a single double word. Based on these analyses state machine's next state is determined.

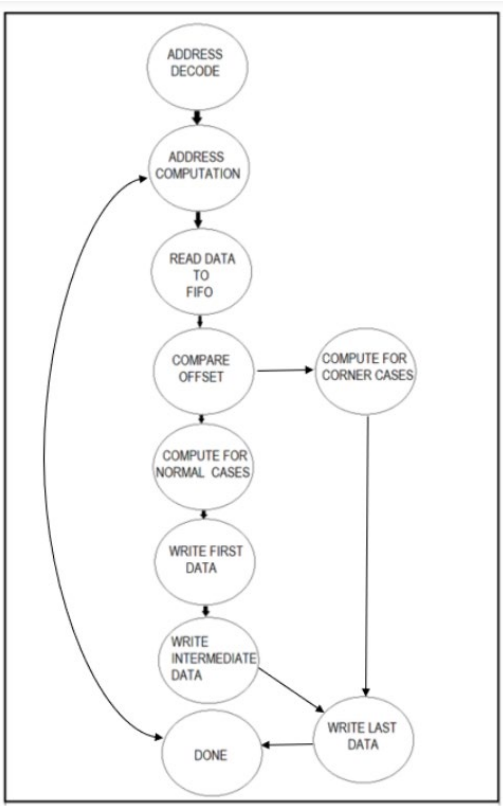

Fig. 1. Bit movement engine state diagram

The state machine moves to the 'compute for corner case' state for cases where the bits to be moved fit in a single double word in the destination. This requires creating three masks and applying them to the data before the destination address offset, after the destination address offset until the block length, and beyond this. The State machine moves to the "compute for normal cases" state for cases where the bits to be moved do not fit in a single double word. In this state, the bits to be moved are properly aligned to the destination offset and write control signals are set up. 'Write first data' state carries the second cycle of write. "Write intermediate data" is the state where all the bits present between the first double word and the last double word are written. This state uses funnel shifters to align the data. The last double word is written in the "write last data" state. Once the last data is written 'done' signal is asserted for a cycle to indicate that the bit movement is complete.

AHB bridge plays the role of translating AHB requests to requests that the bit movement block can sample [5]. Table 3 shows the list of AHB signals. AHB transfer starts with HTRANS being NONSEQ. In this paper AHB bridge and fabric only supports NONSEQ transfers. NONSEQ is a single transfer of read or write. Read and write transfers have two phases: control phase and data phase. HADDR along with other control signals are sampled in the control phase. In the following phase, HRDATA and HWDATA are sampled. Slave can choose to delay the transfer by de-asserting the HREADY signal.

AHB fabric is designed to be a placeholder for an arbiter to arbitrate requests from different masters and slaves. With the bit movement block being the only slave connected to the fabric there is no need for an arbiter within the scope of this paper.

Fabric is designed to receive a pointer to the memory location from where it can fetch the registers for the bit movement block and a pointer to the next set of registers. It has a simple state machine to do this operation. Fabric is responsible for writing the control registers in the bit movement engine and starting the operation. After the engine is done fabric provides new control registers for the engine.

3. Testbench

The testbench is used to drive stimulus to the design and mimic the memory slave. The testbench writes the first pointer to the fabric and responds to the read and write requests to the memory slave. The UVM-based testbench consists of all the components needed to make a fully functional testbench: test,

environment, scoreboard, agent, driver, monitor, sequencer, and sequence [6]. Although UVM provides a number of phases that the test bench can use, this test bench uses the absolutely essential phases for building a fully functional UVM testbench: build phase, connect phase, and run phase. Figure 2 shows a simple block diagram of how each block discussed in this paper is connected.

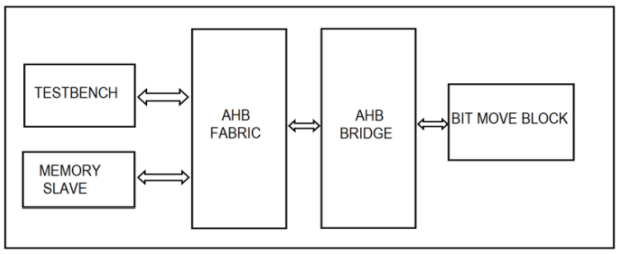

Fig. 2. Block diagram to show the connections

One of the main advantages of using UVM is reusability. For example, the scoreboard used in the verification of the bit movement engine can be used in the chip-level test bench. When simulations are done on the chip level, the scoreboard of the bit movement block will be active and checks the correct behavior of the bit movement engine. Each component of UVM can be reused like a plug-and-play. Sometimes an entire agent can be picked and instantiated in an environment of a different test bench. Communication between these components is established by connecting TLM/analysis ports in one component to imp/export in another component.

The sequence plays a vital role in creating test cases. In this case, generating bit movement block registers, link registers, and memory blocks for the addresses around the source and destination addresses. Once the memory blocks are generated it is passed to the driver and the scoreboard. The driver acts as a memory slave responding to all the read and write requests of the bit movement engine. The scoreboard uses the link register to initiate its reference memory manipulation. Once the bit movement engine is done scoreboard uses its reference memory to check against the memory blocks sent by the driver. If the data doesn't match an error is thrown.

The bit movement block exercises different logic depending on the block length, source address offset, and destination address value. The test cases generated by the sequence should cover all the possible scenarios. Covering all scenarios gives the confidence that the bit movement engine is robust and can move variable bits of data from any address offset to any other address offset. Further, code coverage is run to determine which logic is not exercised. Test stimulus is improved based on the code coverage report. Dead code from the design is removed if the logic is unreachable.

Along with verifying the correct functionality of the bit movement engine, verifying the performance is important [7]. Without performance verification, IP works slow and in turn, the SoC works slow. Bit movement is allowed (block length)/16+10 clocks for completion of data manipulation. This check proves that the bit movement engine is performing the data manipulation well within the provided cycles.

4. Result

The UVM-based testbench generated 20 different test cases covering different block lengths, source address offset and destination address offset. The scoreboard verified all the modified memory blocks for their correctness. Figure 3 shows the UVM report of the simulation. The bit movement block passed the performance checks for all the test cases. Figure 4 shows the performance report of the bit movement block for various test cases.

```
addr[ 108a7d14 ] = dddf2709 expected dddf271f
addr[ 16e988a0 ] = a9442450 expected a944219a
addr[ 24fafe58 ] = 8a8fb8ba expected 8afb8ba9
addr[ 2beceb04 ] = 7ce70451 expected 7ce707e6
addr[ 32177054 ] = f1f41d00 expected f1fbd41d
addr[ 38d40480 ] = 1b7540fa expected 1b7540fe
addr[ 3922b9e0 ] = 7c75e577 expected 7c7795de
addr[ 3af97980 ] = 78c0fcb6 expected 78c0fcac
addr[ 427ec648 ] = 514d284a expected 5534a129
addr[ 4fa060b4 ] = 3e1fa651 expected 3e1fa640
addr[ 66f7fd0c ] = 11146000 expected 1144c8a3
addr[ 6dd6eec0 ] = 97a5bc80 expected 97a5e6f2
addr[ 6f6e1748 ] = 4efae519 expected 4efae50e
UVM_INFO /home/      /uvm-1.2/src/base/uvm_objection.svh(1271) @ 627000: reporter [TEST_DONE] 'run' phase is rea
UVM_INFO /home/      /uvm-1.2/src/base/uvm_report_server.svh(847) @ 627000: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    5
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
[wait]     2

$finish called from file "/home/morris/uvm-1.2/src/base/uvm_root.svh", line 517.
$finish at simulation time              627000
                 V C S   S i m u l a t i o n   R e p o r t
Time: 6270000 ps
```

Fig. 3. UVM report

```
################################################################
Maximum number of clocks to be taken for the operation          18
Number of clocks taken for the operation                        15
################################################################

################################################################
Maximum number of clocks to be taken for the operation          21
Number of clocks taken for the operation                        16
################################################################

################################################################
Maximum number of clocks to be taken for the operation          21
Number of clocks taken for the operation                        19
################################################################

################################################################
Maximum number of clocks to be taken for the operation          31
Number of clocks taken for the operation                        27
################################################################

UVM_INFO bmtx-1.sv(509) @ 609000: uvm_test_top.agnt.scoreboard [Success]


 Test has been completed succesfully!!!
```

Fig. 4. Performance report

5. Conclusion

This paper presented design and verification of a bit movement engine.

Acknowledgment

This project was done under the guidance of Professor Morris Jones at San Jose State University. I am thankful to Professor Morris Jones for his guidance and support and to San Jose State University for providing all the facilities that were necessary for the project.

References

[1] H. Jin, D. Jeong, T. Park, J. H. Ko and J. Kim, "Multi-Prediction Compression: An Efficient and Scalable Memory Compression Framework for GP-GPU," in *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 37-40, 1 July-Dec. 2022.

[2] AMBA specification (rev 2.0), 1999.

[3] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in *IEEE Std 1800.2-2017*, vol., no., pp.1-472, 26 May 2017.

[4] E. Clifford and Cummings, "Coding and Scripting Techniques for FSM Designs with Synthesis-Optimized Glitch-Free Outputs," *SNUG (Synopsys Users Group Boston MA 2000) Proceedings*, September 2000.

[5] V. T. Mahendra, D. S. Shylu Sam, A. J. Hernisha and A. J. Atchaya, "Design of highly reusable interface for AHB verification module," *2022 6th International Conference on Devices, Circuits and Systems (ICDCS)*, 2022, pp. 357-359.

[6] W. Ni and J. Zhang, "Research of reusability based on UVM verification," *2015 IEEE 11th International Conference on ASIC (ASICON)*, 2015, pp. 1-4.

[7] P. Bose and J. A. Abraham, "Performance and functional verification of microprocessors," *VLSI Design 2000. Wireless and Digital Imaging in the Millennium. Proceedings of 13th International Conference on VLSI Design*, 2000, pp. 58-63.