Designing a Custom AXI Master using Bus Functional Models (BFMs)

Version 1.0, May 2015
Rich Griffin (Xilinx Embedded Specialist, Silica EMEA)

Overview

In a previous application note we explored the creation of a custom AXI-lite Slave Peripheral, designed for use in the Xilinx Zynq-7000 All Programmable SoC devices. In this guide we will explore the creation of custom AXI Master, which can be used generate transactions on the AXI interconnect, controlled by custom HDL logic.

As we noted in previous application notes, the AXI4 (Advanced eXtensible Interface) specification has been designed to offer different variants of the interconnect. In the case of our previous example, we created a slave peripheral which implemented the "lite" variant of AXI4. In this application note we will build upon that knowledge by introducing the additional signals and features that are relevant when implementing transactions using the "full" address-mapped variant of AXI4. This means that we will be able to create transactions which generate bursts of up to 256 beats of data on the interconnect, and we will also explore some of the additional signalling that is available to users.

Pre-requisites

In my previous application note, I discussed the use of the AXI4 signalling / handshaking protocol and described each of the signals in detail. This document will not attempt to cover any of that material in any detail, because it will be considered as "assumed knowledge" for the reader. If you are unfamiliar with the AXI4 protocol / signalling / handshaking requirements, or if you feel that you might need to refresh your understanding of it, you are strongly encouraged to refer to the previous application note before proceeding with this one.

AXI4 vs. AXI4-lite; a comparison of features

Burst Modes & Alignment Principles

The first major difference between the two variants of the AXI4 protocol concerns the ability to "burst" data from masters to slaves. In our previous discussion relating to the AXI4-lite protocol, we noted that the interface was very simple and only allowed single beats of data to be transferred. In AXI4 "full" the possibility exists to transfer bursts of data up to 256 data beats in size. AXI4 also permits data beats which vary in size to be transferred, ranging from 1 byte (8 bits) up to 128 bytes (1024 bits) in width. In the AXI4 signal set, there are two signals in the read and write address channels which communicate the size (i.e. "data width") and also the length of the burst to be transferred. When considering bursts of data it is also important to consider how the data will be received. Three options exist here; “incrementing”, “fixed”, and “wrapping” bursts. The first two burst modes are simple to comprehend, and the third is less so. “Incrementing” bursts are perhaps the most obvious style, and imply that the destination address is incremented for each beat in the burst. Taking the example of a burst starting at address 0x20000000 which transfers 32-bit data beats, the address bus for a typical RAM structure would increment by four bytes for each beat of data; 0x20000000, 0x20000004, 0x20000008, 0x2000000C, and so on. “Fixed” bursts are those which send / receive every beat of data to / from the same address, and an example of this is when one is transferring data into a FIFO. The FIFO is likely to have a fixed address in the system address map and therefore does not operate like a traditional RAM structure. For this reason, the value on the address bus never changes during a fixed burst.

The last style of burst, “wrapping”, requires a little more imagination and to understand it properly we must first consider the concept of “alignment” of data values in the address map. If we take an example of a burst transaction comprised of 8 data beats (i.e. 8 transfers) which are each 4 bytes in size (32 bits) and starting at address 0x20000000, then it is easy to understand that the last beat of the burst (the 8th beat) will occur at address 0x2000001C.
Each beat of the burst is aligned to a boundary in the system address map which is applicable to the size of the beat being transferred. In other words, and continuing our example using eight 32-bit data beats, we can see that the addresses ending “0000”, “0004”, “0008”, and “000C” are all aligned addresses for a 32 bit data width. All other addresses would be un-aligned, and examples of this include (but are not limited to) “0001”, “0002”, “0003”, and “0005”. To cement this concept, if we were transferring larger data beats of 8 bytes in size (64 bits), then there would only be two naturally aligned addresses in the same address range; the addresses ending “0000” and “0008”. With that concept in mind, let us now consider the concept of alignment for the burst rather than for each data beat. Using the same example as before, we can see that the start address of the burst is aligned when the start address is 0x20000000, but would be un-aligned if we had started it at address 0x20000004.

	Byte	**Byte**	**Byte**	**Byte**
Aligned word	0x20000000	0x20000001	0x20000002	0x20000003
	0x20000004	0x20000005	0x20000006	0x20000007
Un-aligned word	0x20000008	0x20000009	0x2000000A	0x2000000B
	0x2000000C	0x2000000D	0x2000000E	0x2000000F

Note the key difference here between burst alignment and beat alignment; the start address 0x20000000 offers alignment for both the burst and the beat, but the address 0x20000004 only offers alignment for the <u>beat</u>.

	32 bit Word	**32 bit Word**	**32 bit Word**	**32 bit Word**
Un-aligned 8 beat burst	0x20000000	0x20000004	0x20000008	0x2000000C
	0x20000010	0x20000014	0x20000018	0x2000001C
	0x20000020	0x20000024	0x20000028	0x2000002C
	0x20000030	0x20000034	0x20000038	0x2000003C

	32 bit Word	**32 bit Word**	**32 bit Word**	**32 bit Word**
Aligned 8 beat burst	0x20000000	0x20000004	0x20000008	0x2000000C
	0x20000010	0x20000014	0x20000018	0x2000001C
	0x20000020	0x20000024	0x20000028	0x2000002C
	0x20000030	0x20000034	0x20000038	0x2000003C

Taking this concept one stage further; if we started the same burst four bytes higher in the address map, at address 0x20000004, then we can clearly imagine that the last beat of the transfer would occur at address 0x20000020. This represents an un-aligned burst transfer, even though each beat (i.e. each 32 bit word within the burst) was aligned in the address map. It is this concept of <u>burst</u> alignment which helps us to introduce the concept of “wrapping” bursts. So if we reconsider our last example of the 8 beat burst which starts at address 0x20000004, the address bus would increment by 4 bytes for every beat in the burst.

	32 bit Word	32 bit Word	32 bit Word	32 bit Word
Incrementing Burst	0x20000000	0x20000004 (1st Beat)	0x20000008 (2nd Beat)	0x2000000C (3rd Beat)
	0x20000010 (4th Beat)	0x20000014 (5th Beat)	0x20000018 (6th Beat)	0x2000001C (7th Beat)
	0x20000020 (8th Beat)	0x20000024	0x20000028	0x2000002C
	0x20000030	0x20000034	0x20000038	0x2000003C

However the key difference in this example between an incrementing burst and a wrapping burst lies with the address of the 8th beat of the burst. In the incrementing burst scenario we saw that the last beat would be transferred at address 0x20000020, but with a wrapping burst the last beat of the burst would occur at address <u>0x20000000</u>. This is initially a bizarre concept, because the address of the last beat of the burst is actually 4 bytes <u>below</u> the start address. The reason for this behaviour is because the address bus increments up to the burst alignment boundary relative to the size of that burst transaction, and then wraps around at that boundary point to fill the empty gap between the start address and the burst boundary below it.

	32 bit Word	32 bit Word	32 bit Word	32 bit Word
Wrapping Burst	0x20000000 (8th Beat)	0x20000004 (1st Beat)	0x20000008 (2nd Beat)	0x2000000C (3rd Beat)
	0x20000010 (4th Beat)	0x20000014 (5th Beat)	0x20000018 (6th Beat)	0x2000001C (7th Beat)
	0x20000020	0x20000024	0x20000028	0x2000002C
	0x20000030	0x20000034	0x20000038	0x2000003C

Once again we can cement this concept by considering a similar burst transaction, but one which starts at address 0x2000000C. The first five beats of the burst target the addresses ending “000C”, “0010”, “0014”, “0018”, and “001C”, but the last three beats wrap around at the burst boundary to target the addresses ending “0000”, “0004”, and “0008”.

	32 bit Word	32 bit Word	32 bit Word	32 bit Word
Wrapping Burst	0x20000000 (6th Beat)	0x20000004 (7th Beat)	0x20000008 (8th Beat)	0x2000000C (1st Beat)
	0x20000010 (2nd Beat)	0x20000014 (3rd Beat)	0x20000018 (4th Beat)	0x2000001C (5th Beat)
	0x20000020	0x20000024	0x20000028	0x2000002C
	0x20000030	0x20000034	0x20000038	0x2000003C

Confused? Yes, you might well be! The concept of wrapping bursts can often be regarded as completely incomprehensible to many users, but is considered to be essential and highly useful to others. Depending on your intended use model for your custom AXI master, you will either be celebrating or feeling utterly terrified at this stage. If it is the latter then don’t worry because it suggests that wrapping bursts are not likely to be useful to you, and you can just forget about the whole concept for now. In either case, but principally in the interests of preserving the reader's sanity, we shall not discuss this topic any further at this stage.

Caching Modes

The full AXI4 specification provides a number of powerful features which concern how caches may be used in the system. We are designing a custom AXI4 master which is designed to communicate with slaves in the system, and it is convenient to assume that only the AXI interconnect lies between the two blocks of logic. However in reality it is important to consider that other system blocks might be implemented between the master and the slave, and one example of this could be a cache array. Our custom master can generate an AXI4 transaction but, in the event that a cache array was present in the system, the master must also indicate to the cache controller how the transaction may be treated. For example, if our custom master is generating a transaction which will read from or write to the main DDR memory, then it is possible that caches could be used to accelerate the response time of the transaction. Conversely, if our custom master was writing to a GPIO pin, then it might be highly undesirable for a cache controller to treat the write transaction as potentially cacheable, because the GPIO might not actually be updated until a much later time than that which was intended by the master. For this reason some additional signalling is present on the full AXI4 interconnect which allows the master to indicate how the transaction may be treated by any downstream functional blocks which implement caching functions. The options for caching and buffering are vast and extremely complicated, making them far too lengthy to describe in detail in this document. In our example we will keep things simple for the beginner and our example will control the caching signals to indicate that no caching should occur.

Protection Modes

In the AXI4 variant of the interconnect it is possible to indicate that some transactions are protected in various ways; examples include privileged / unprivileged memory accesses, secure and non-secure memory accesses, and a method to indicate that the transaction is an instruction fetch rather than a data fetch. Once again this topic is considerably beyond the scope of this document and requires a detailed understanding of the ARM Security Extensions. For the purposes of designing our custom master, we will assume that we will be generating the simplest form of transactions; unprivileged, secure, data accesses. This is not a cop-out, but simply reflects the most common requirements for users wishing to design a custom AXI master.

QoS Modes

When designing a system on chip from the ground up, the designer has the freedom to implement all slaves, all masters, and all interconnect structures in between them. With that level of freedom, the designer might choose to tag some transactions to have a greater priority than others, by design. The AXI4 specification supports this requirement by offering Quality of Service (QoS) signalling as part of the signal set. Although this is a very powerful feature, the success of the implementation requires that all parts of the system support the use of such signalling. In our example we are designing just one custom AXI master to be implemented in the programmable logic of a Zynq device, and thus we do not have control over the rest of the system. An example of this lack of control is the implementation of the AXI interconnects which are hard coded as part of the Zynq-7000 silicon architecture, and therefore cannot be changed. Happily, the AXI4 specification provides a way for custom masters to indicate that they do not wish to participate in any QoS scheme, and this is the mode we shall choose when implementing our custom master.

Transaction IDs

Perhaps the biggest difference between the AXI4 "full" and "lite" variants is the concept of transaction IDs. AXI4 masters are permitted to generate multiple transactions in quick succession, but neither the AXI interconnect nor the slaves are required by the specification to process or indeed complete the transactions in the same order that they were generated. The master is also permitted to begin generating one transaction, for example by sending addressing information on the write address channel, and then commence another one intended for a different slave, before sending / receiving any data for the first transaction on the write data channel. To achieve this functionality, each transaction on each of the five AXI channels is tagged with a ID number ranging between zero and fifteen, and the slave / interconnect must reply to each transaction using the same ID number that was provided by the master, so that the responses can be correctly associated with the requests. At first glance this is a feature which might appear to be extremely useful, because it allows the master to operate in the most efficient way possible. However the real world practicalities of implementing this feature in hardware are simply mind-blowing for the designer, because details about all ongoing transactions must be stored by the various parts of the system just in case they are not completed before another transaction arrives. It is therefore common-place that the ID feature of the AXI4 specification is not used, and indeed most implementations always assume that ID=0 will be used. In the interests of simplicity, we shall be doing the same thing when designing our custom master.

Memory Regions

The AXI4 "full" specification provides support for overlapping memory regions. This is very much beyond the scope of most designs, but is used to expand the address map to allow multiple logical address mappings to be presented to a single physical interface. For example, in the case of a slave peripheral, the same addresses could be used for both the configuration registers and also the data input/output registers, simply by assigning each set of registers to a different memory "region". However for most practical designs, users find that there is ample address space for all of the registers and memory space that are required in the system. For this reason most designs will ignore the memory regions and just use the default region, which is region zero.

User Signals

Lastly, the AXI4 "full" specification includes some additional signalling for custom user communication. As the name perhaps suggests, there is no pre-defined use model for this feature and it is left up to the user to decide what they will do with it. Naturally this creates some additional signals (with the suffix "USER") in the specification, but they can be safely ignored if they are not required. Note: Once again, this feature often requires control over the entire System On Chip in order for it to be implemented successfully.

Summary

Despite adding all of these additional features, the underlying AXI4 protocol is identical for both the AXI4-lite and AXI4 "full" variants. However due to the huge number of additional features, we must now explore a significant number of new signals, which control how to use (or to disable!) the features described above. A modular design approach is critical to the success of designing an AXI4 master and, just as we did before with the custom slave, we must split the design up into the five AXI channels and then control them with a top level of hierarchy.

AXI4 Signalling; The differences from AXI4-lite

As we noted above, the AXI4 handshake protocol is identical between the AXI4-lite and AXI4 "full" variants, and therefore the "ready" and "valid" signals should already be familiar to the reader. The five AXI channels that we explored previously are implemented in precisely the same way and perform the same functions, albeit with an enhanced feature set. We will now explore the new signals which enable all of the additional features described above and compare them to those that we used to design the custom AXI4-lite slave in the previous application note. It is very important to remember that we will be designing a master rather than a slave, and this means that a lot of the signals which were previously inputs are now outputs, and vice versa.

Read & Write Address Channels

Starting with the read and write address channels, we can see that the three signals from the AXI4-lite variant ("Address", "Valid" and "Ready") are precisely the same, with the exception that the port direction (input vs output) are swapped because we are now designing a master rather than a slave. The remaining signals refer to the additional features implemented in the "full" variant of the AXI4 specification, and are described in the table. The table shown here is documented for the read address channel, but the write address channel works in precisely the same way.

AXI4 Read Address Channel			
Signal Name	Size	Driven by	Description
M_AXI_ARADDR	32 bits	Master	Address bus output from AXI master device.
M_AXI_ARVALID	1 bit	Master	Valid signal, asserting that the slave peripheral should sample the other signals on this channel.
M_AXI_ARREADY	1 bit	Slave	Ready signal, driven by the slave to indicate that it is ready to accept the values on the address lines and other control signals.
M_AXI_ARID	4 bits	Master	Transaction identifier, used in systems where multiple are issued by one master without waiting for the previous transaction to complete. The use of transaction IDs are beyond the scope of this document so we will use ID="0000".
M_AXI_ARLEN	7 bits	Master	Burst length signal, used to communicate the number of beats in a burst. This is a zero based number and therefore a single beat transaction has the value LEN=0.

			The maximum burst length of 256 beats has the value LEN=255.
M_AXI_ARSIZE	3 bits	Master	Burst size signal, indicating the data width of each beat in the burst (in bytes). The three bits provide eight combinations, allowing the user to specify data widths of 1, 2, 4, 8, 16, 32, 64, and 128 bytes using values 0x000 to 0x111, respectively.
M_AXI_ARBURST	2 bits	Master	Burst type signal, indicating whether the burst is Fixed (0b00), Incrementing (0b01), or Wrapping (0b10). The last signal combination (0b11) is reserved.
M_AXI_ARLOCK	1 bit	Master	Lock signal used to indicate atomic accesses. The value LOCK = 0 indicates a normal access, and LOCK = 1 indicates an exclusive access. In most use cases this signal is driven to LOCK = 0.
M_AXI_ARCACHE	4 bits	Master	Caching mode signal, used by the master to indicate to the rest of the system how the transaction may (or may not!) be cached. Caching modes are a complex topic and beyond the scope of this document. For the sake of simplicity we shall use the value CACHE = 0b0000, which denotes "Device Non-bufferable" mode whereby caching of the transaction is not permitted, and data is required to be sent to / fetched from the end device (i.e. the slave) without being pre-fetched, cached, or otherwise modified in any way.
M_AXI_ARPROT	3 bits	Master	Protection signal, used to indicate whether the transaction is privileged, uses features from the ARM Security Extensions, or is an instruction access. We will keep things simple by using mode PROT = 0b000, which denotes an unprivileged, secure, data access. (e.g. This is what most people would call a "normal" transaction)
M_AXI_ARQOS	4 bits	Master	Quality of Service (QoS) signal which is only used when implementing transaction priority (which must be supported by masters, slaves, and interconnects across the entire system). We will use QOS = 0b0000, which denotes that the master is not participating in any QoS scheme.
M_AXI_ARREGION	4 bits	Master	Address region identifier, used to expand the address map and permit overlapping addresses by specifying multiple regions. In most designs the 4GB address space is more than ample, so we will not use this signal and leave it set to its default value. REGION = 0b0000.

As can be seen from the above table, the majority of the additional signals for AXI4 are not terribly relevant to a simple design and have been set to the most "normal" mode for the sake of simplicity. You may adjust this for your design as you see fit.

Read Data Channel

For the read data channel, the signals are almost identical to the signals that we discussed in the previous application note when designing the AXI4-lite slave. Two signal have been added, "RID" and "RLAST", which are used to support transaction identifiers, and to support burst transactions by indicating which beat is the last beat in the burst.

AXI4 Read Data Channel			
Signal Name	Size	Driven by	Description
S_AXI_RDATA	Variable, 8-1024 bits	Slave	Data bus from the slave peripheral / interconnect, providing read data to the Master.
S_AXI_RVALID	1 bit	Slave	Valid signal, asserted by the slave to show that the data bus and control signals are driven with valid values, and indicating that they may be sampled by the Master.
S_AXI_RREADY	1 bit	Master	Ready signal, indicating that the Master is ready to accept the value on the other signals.
S_AXI_RRESP	2 bits	Slave	A "Response" status signal showing whether the transaction completed successfully or whether there was an error. A successful transaction is denoted by RESP = 0b00.
M_AXI_RID	4 bits	Master	Transaction identifier, used in systems where multiple are issued by one master without waiting for the previous transaction to complete. The use of transaction IDs are beyond the scope of this document so we will use ID="0000".
M_AXI_RLAST	1 bit	Slave	"Last beat" signal, indicating that the data beat currently being transferred is the last in the burst. This signal is also asserted for single beat transactions, because it is both first and last beat in the burst.

Write Data Channel

Once again, the write data channel is almost identical to that which was described in the previous application note for the AXI4-lite custom slave. The additional signals are the same as we noted for the read data channel, the "WID" to carry the transaction identifier information, and "WLAST" signal to denote the last beat in a burst.

AXI4 Write Data Channel			
Signal Name	Size	Driven by	Description
M_AXI_WDATA	Variable, 8-1024 bits	Master	Data bus from the Master to the AXI interconnect / slave peripheral.
M_AXI_WVALID	1 bit	Master	Valid signal, asserting that the M_AXI_WDATA and other control signals can be sampled by the interconnect / slave.
M_AXI_WREADY	1 bit	Slave	Ready signal, indicating to the master that the interconnect / slave is ready to accept the values on the other signals.
M_AXI_WSTRB	(WDATA_width/8) bits	Master	A "strobe" status signal showing which bytes of the data bus are valid and should be read by the interconnect / slave. Each bit of STRB represents one byte of data.
M_AXI_WID	4 bits	Master	Transaction identifier, used in

			systems where multiple transactions are issued by one master without waiting for the previous transaction to complete. The use of transaction IDs are beyond the scope of this document so we will use ID="0000".
M_AXI_WLAST	1 bit	Slave	"Last beat" signal, indicating that the data beat currently being transferred is the last in the burst. This signal is also asserted for single beat transactions, because it is both first and last beat in the burst.

The write strobe signals were discussed in great detail in the previous application note, but in the case of the AXI4 "full" implementation there is the possibility to have a wide range of data bus widths. For this reason, the number of WSTRB bits is dependent on the width of the "WDATA" bus and is calculated using "Width_of_WSTRB = Width_of_WDATA / 8". In the example design we shall only be implementing 32 or 64 bit data transfers, so the strobe bits will always all be asserted.

Write Response Channel

The write response channel is almost identical to that which was described in the previous application note for the AXI4-lite custom slave. The only additional signal is "BID", to tag the responses with a transaction identifier.

AXI4 Write Response Channel			
Signal Name	Size	Driven by	Description
M_AXI_BREADY	1 bit	Master	Ready signal, indicating that the Master is ready to accept the "BRESP" response signal from the interconnect / slave.
M_AXI_BRESP	2 bits	Slave	A "Response" status signal showing whether the transaction completed successfully or whether there was an error. For a successful response, RESP = 0b00.
M_AXI_BVALID	1 bit	Slave	Valid signal, asserting that the M_AXI_BRESP may be sampled by the Master.
M_AXI_BID	4 bits	Slave	Transaction identifier, driven by the slave to denote for which transaction ID it is providing a response.

Using modular design techniques to develop the Custom AXI4 Master

The potential complexity of the AXI4 specification is such that a design can quickly become unmanageable. It is therefore vital to break the design down into simple modules in order to have any hope of successfully creating it. Fortunately the AXI4 specification has been written in such a way that we can easily achieve this, and the most obvious method is to implement each of the five AXI4 channels separately.

In the example design provided, each AXI4 channel is implemented using its own state machine. The intent of this is to allow the channels to be tested in isolation (testbenches for all five channels are also provided), and then the five channels can be combined together under the control of a top level state machine. It is unnecessary for us to discuss the design of all five channels, because the implementation and coding style for all five channels is similar. Please refer to the provided example design for further details.

All five channels have two distinct sets of ports; one set of AXI4 ports, and one set of user ports. The purpose of each channel module is to provide an intuitive user interface via the "user" ports, and then have the module implement the associated AXI specification without requiring the user to know anything about AXI4. As an example, here is the ports list for the AXI4 Address channel (used for both read address and write address channels)

```
entity AXI_ADDRESS_CONTROL_CHANNEL is
       PORT
              (
              -- User signals
              Clk                    : in  STD_LOGIC;
              resetn                 : in  STD_LOGIC;
              go                     : in  STD_LOGIC;
              done                   : out STD_LOGIC;
              error                  : out STD_LOGIC;
              address                : in  std_logic_vector(31 downto 0);
              burst_length           : in  integer range 1 to 256;
              burst_size             : in  integer range 1 to 128;
              increment_burst        : in  STD_LOGIC;

              -- AXI Master signals
              AxID                   : out STD_LOGIC_VECTOR (3 downto 0);
              AxADDR                 : out STD_LOGIC_VECTOR (31 downto 0);
              AxLEN                  : out STD_LOGIC_VECTOR (7 downto 0);
              AxSIZE                 : out STD_LOGIC_VECTOR (2 downto 0);
              AxBURST                : out STD_LOGIC_VECTOR (1 downto 0);
              AxLOCK                 : out STD_LOGIC;
              AxCACHE                : out STD_LOGIC_VECTOR (3 downto 0);
              AxPROT                 : out STD_LOGIC_VECTOR (2 downto 0);
              AxVALID                : out STD_LOGIC;
              AxREADY                : in  STD_LOGIC;
              AxQOS                  : out STD_LOGIC_VECTOR (3 downto 0);
              AxREGION               : out STD_LOGIC_VECTOR (3 downto 0)
              );
end AXI_ADDRESS_CONTROL_CHANNEL;
```

The AXI4 signals have already been described, and the user signals are described in the table below.

Signal Name	Port type	Description
clk	1 bit input	Main clock signal.
resetn	1 bit input	Main reset signal. Active low.
go	1 bit input	Input signal used to initiate an AXI4 transaction, using the values provided on the other user ports. Active high. The internal state machine will wait for this signal to be de-asserted and re-asserted by the user before another operation is initiated.
done	1 bit output	Status signal indicating that the operation of the state machine has completed. This signal should be read in conjunction with the "error" signal to determine whether the AXI4 operation completed successfully or not. The

		Done output will remain asserted until the user de-asserts the "Go" signal, or will be toggled high for one clock cycle otherwise.
error	1 bit output	Status signal indicating that an error occurred during operation. This signal is asserted at the same time as the "done" signal, and has identical handshaking as "done" with respect to the "go" input.
address	32 bit input	An input to allow the user to specify the target address for the AXI4 transaction. In the case of burst transactions, this signal is used to indicate the address of the first beat.
burst_length	Integer input (range 1 to 256)	An integer input denoting the number of beats in the burst. Single beat transactions have a burst length of 1.
burst_size	Integer input (range 1 to 128)	An integer input denoting the size (data width) of each beat to be transferred, in bytes. Only eight values are valid on this signal; 1, 2, 4, 8, 16, 32, 64, and 128. All other combinations will result in an error.
increment_burst	1 bit input	Input signal used to denote the type of burst to be generated. 1 = incrementing burst. 0 = non-incrementing burst (mailbox / FIFO style). Single beat transactions may use either style of burst to achieve the same result. Wrapping bursts are not supported in the provided example design.

Implementing each AXI channel using a finite state machine

Provided that the custom AXI Master is broken down into manageable pieces, it is a relatively simple task to implement each part of the design. A code extract from the provided example is shown below, showing some of the assignments to the AXI control signals, and also the operation of the state machine.

```
AxADDR <= address when address_enable = '1' else (others => '0');
AxID <= (others => '0');  -- Transaction ID = 0
AxLOCK <= '0';  -- Normal transaction
AxCACHE <= (others => '0');  -- Non-bufferable & Non-cacheable
AxPROT <= (others => '0');  -- Normal access, secure access, data access.
AxQOS <= (others => '0');  -- QoS scheme not used.
AxREGION <= (others => '0');  -- Memory region scheme not used; defaulting to region zero.

state_machine_decisions : process (current_state, go, AxREADY, burst_size, burst_length)
begin
    done <= '0';
    address_enable <= '0';
    AxVALID <= '0';
    error <= '0';

    case current_state is
        when reset =>
            next_state <= idle;

        when idle =>
            next_state <= idle;
            if go = '1' then
                case burst_size is
                    when 1|2|4|8|16|32|64|128 =>
                        case burst_length is
                            when 1 to 256 => next_state <= running;
                            when others => next_state <= error_detected;
                        end case;
                    when others => next_state <= error_detected;
                end case;
            end if;
        when running =>
            next_state <= running;
            address_enable <= '1';
            AxVALID <= '1';
            if AxREADY = '1' then
                next_state <= complete;
            end if;

        when complete =>
            next_state <= complete;
            done <= '1';
            if go = '0' then
                next_state <= idle;
            end if;

        when error_detected =>
                next_state <= error_detected;
                done <= '1';
                error <= '1';
                if go = '0' then
                    next_state <= idle;
                end if;

        when others =>
            next_state <= reset;
    end case;
end process;
```

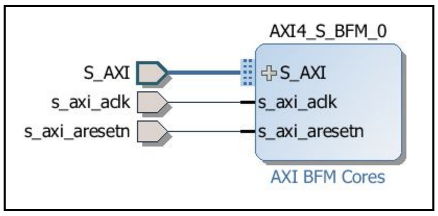
All of the other AXI channels can be implemented in a similar way, and this greatly reduces the complexity of the design. Arguably the most difficult section of the design to implement is the level of hierarchy above (and controlling) the five AXI channel modules. However this is only because there are a larger number of control signals from the five AXI channel modules which need to be managed, but this too is achieved by the use of a finite state machine. In the provided example, even the top level of hierarchy has only seven states implemented within the FSM.

Creating a test platform for developing the Custom AXI4 Master

When we designed the AXI4-lite custom slave in the previous application note, it was relatively straight forward to construct a test environment against which to validate the correct operation of our design. The testbench that was provided for this purpose included a VHDL model for an AXI4-lite master. This allowed designers to generate AXI4-lite transactions at their convenience, in order to test the operation of the custom slave. It was an adequate solution but was still time-consuming to implement, and consisted of around 2000 lines of VHDL. Although it is technically feasible to construct VHDL models to test the operation of the full AXI4 specification, it would represent a significant investment in time and would take significant time to debug the models. A better solution is to purchase and use the Bus Functional Models which are provided by Xilinx for this precise purpose. (http://www.xilinx.com/products/intellectual-property/do-axi-bfm.html) Bus Functional Models (BFMs) are designed to be incorporated into the HDL design flow, and provide an easier way to interactively simulate the behaviour of the AXI4 interconnect. In addition to modelling the interconnect protocols and signalling, the BFM package provides ready-made models for both masters and slaves. This allows the user to drop a black-box instance into their testbench and configure how they want it to behave in the simulation environment. The master model can be used to generate transactions of any type, including support for transaction IDs, bursts (including support for the various burst modes), caching types, and using any of the different data widths which are supported in the AXI specification. The slave model can be used to create a simulated response to AXI4 transactions, and may be instantiated multiple times to create a simulation environment which mimics a genuine processor SoC with multiple slave peripherals. Each slave model can be configured to either respond to read transactions with fixed data values, or can be configured to store the values written to it via AXI4 write transactions and then return that data later in the simulation, thus essentially behaving like a RAM model. Timing delays can also be simulated, ensuring that the Ready/Valid handshaking protocol of AXI4 can be fully tested, even if the latencies across multiple simulated AXI4 transactions are not consistent. Although there is a monetary cost associated with the purchase of the Bust Functional Models, it is a wise investment for anyone who is planning to develop their own custom AXI4 master devices, and will accelerate the design flow and increase productivity.

Adding the BFM to the design

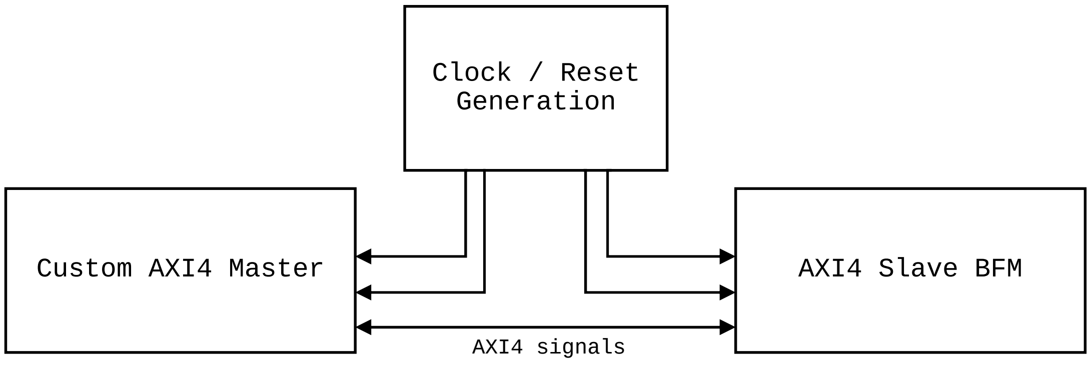
The Bus Functional Model is a complex piece of technology, but is relatively simple to add to a user design. The Xilinx Vivado tools offer a graphical block diagram environment which is used by many embedded processing system designers to connect the various AXI4-enabled IPs together. The BFMs also have graphical tiles which are available from the IP catalogue, making the integration of a BFM a quick and easy process. For the purposes of our simple example, we shall implement the BFM as a single IP in the block diagram, and then use the automatically generated VHDL wrapper to connect it to the AXI4 master that we will develop. In the block diagram example shown here, we have connected the three ports of the BFM (AXI4 interconnect, clock, and reset) and made them external ports in the block diagram. The resulting VHDL wrapper quickly provides us with the complete signal list for the AXI4 interconnect. In effect, the complexities of the BFM model are simplified to the point whereby all of the inner workings are hidden, and the BFM simply looks like any other AXI4 slave device. The designer will invariably be using HDL to design the custom AXI4 master, and therefore they can connect the



```
entity design_1_wrapper is
  port (
    S_AXI_araddr : in STD_LOGIC_VECTOR ( 31 downto 0 );
    S_AXI_arburst : in STD_LOGIC_VECTOR ( 1 downto 0 );
    S_AXI_arcache : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_arid : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_arlen : in STD_LOGIC_VECTOR ( 7 downto 0 );
    S_AXI_arlock : in STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_arprot : in STD_LOGIC_VECTOR ( 2 downto 0 );
    S_AXI_arqos : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_arready : out STD_LOGIC;
    S_AXI_arregion : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_arsize : in STD_LOGIC_VECTOR ( 2 downto 0 );
    S_AXI_aruser : in STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_arvalid : in STD_LOGIC;
    S_AXI_awaddr : in STD_LOGIC_VECTOR ( 31 downto 0 );
    S_AXI_awburst : in STD_LOGIC_VECTOR ( 1 downto 0 );
    S_AXI_awcache : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_awid : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_awlen : in STD_LOGIC_VECTOR ( 7 downto 0 );
    S_AXI_awlock : in STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_awprot : in STD_LOGIC_VECTOR ( 2 downto 0 );
    S_AXI_awqos : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_awready : out STD_LOGIC;
    S_AXI_awregion : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_awsize : in STD_LOGIC_VECTOR ( 2 downto 0 );
    S_AXI_awuser : in STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_awvalid : in STD_LOGIC;
    S_AXI_bid : out STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_bready : in STD_LOGIC;
    S_AXI_bresp : out STD_LOGIC_VECTOR ( 1 downto 0 );
    S_AXI_buser : out STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_bvalid : out STD_LOGIC;
    S_AXI_rdata : out STD_LOGIC_VECTOR ( 31 downto 0 );
    S_AXI_rid : out STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_rlast : out STD_LOGIC;
    S_AXI_rready : in STD_LOGIC;
    S_AXI_rresp : out STD_LOGIC_VECTOR ( 1 downto 0 );
    S_AXI_ruser : out STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_rvalid : out STD_LOGIC;
    S_AXI_wdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    S_AXI_wlast : in STD_LOGIC;
    S_AXI_wready : out STD_LOGIC;
    S_AXI_wstrb : in STD_LOGIC_VECTOR ( 3 downto 0 );
    S_AXI_wuser : in STD_LOGIC_VECTOR ( 0 to 0 );
    S_AXI_wvalid : in STD_LOGIC;
    s_axi_aclk : in STD_LOGIC;
    s_axi_aresetn : in STD_LOGIC
  );
end design_1_wrapper;
```

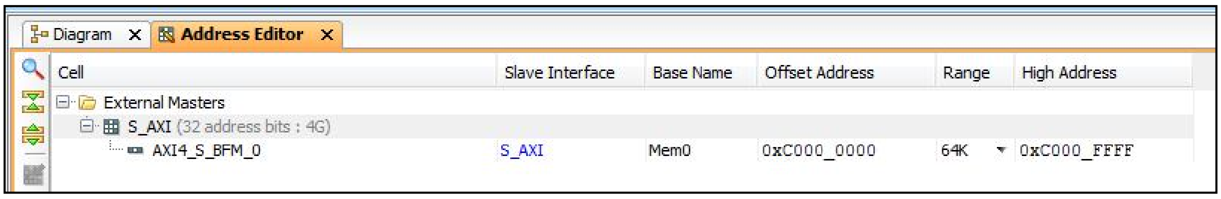
two blocks together using a simple “port map” statement in VHDL (or similar syntax in Verilog). In the case of the provided example, we have done precisely this. There is no need to implement the AXI4 interconnect as a separate entity because the specification allows a master and slave to communicate with no other logic in between. In an AXI4 design, the clock and reset are generated in the AXI4 interconnect and connected to both masters and slaves throughout the system. Therefore a clock and reset must be generated in the top level testbench for the purposes of running a simulation.

The resulting hierarchy diagram for this connection is therefore extremely simple, and is shown below.



Controlling the BFM’s behaviour

Adding the Bus Functional Model into the design is very straight forward, but sadly controlling its operation is not nearly so. In the case of a slave model the BFM needs to know how to respond to transactions and that in turn requires that it have an address range (Base address & High address), in the same way that genuine AXI4 slave peripherals have. This part of the configuration is easy to adjust, and is achieve using the Address Editor in the block diagram environment in the same way that would be done for a genuine slave peripheral. In the example below we can see that we have a single AXI4 slave BFM with the instance name “AXI4_S_BFM_0”, and we have assigned it an address range of 0xC0000000 to 0xC000FFFF.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
External Masters					
S_AXI (32 address bits : 4G)					
AXI4_S_BFM_0	S_AXI	Mem0	0xC000_0000	64K	0xC000_FFFF

The next piece of the puzzle is to configure the BFM slave with the desired functionality, to tell it how to respond to transactions when they are received. Unfortunately this is where the simplicity ends. When the Bus Functional Models were created by Cadence, the decision was made by Xilinx to only provide single language support and, due to the country of origin being the United States of America, the chosen language was Verilog. This is obviously a matter of some inconvenience for the vast majority of the world and requires that the BFM stimulus, and any levels of hierarchy above it in the BFM-enabled simulation, be written in Verilog rather than VHDL.

With this enormous and depressing limitation in mind, we can turn our attention to writing the Verilog file which controls the BFM slave. At the top of the file we start by declaring a "definition" which points to the location of the BFM slave instance in the hierarchy of the design. In our case we shall use the same name "AXI4_S_BFM_0" that we used in the block diagram (for convenience, not because the names are required to match). The definition in the Verilog code points to an instance called "`axi_master_tb.design_1_wrapper_inst.design_1_i.AXI4_S_BFM_0.cdn_axi4_slave_bfm_inst`", which can be derived in any design from the names of each level of hierarchy in the design, delimited by the "." character, and suffixed by "`cdn_axi4_slave_bfm_inst`", which is the hard-coded name of the instance which lies within the Cadence BFM.

```
`define AXI4_S_BFM_0 axi_master_tb.design_1_wrapper_inst.design_1_i.AXI4_S_BFM_0.cdn_axi4_slave_bfm_inst
```

The rest of the Verilog file is just a traditional Verilog "module", which has minimal inputs and outputs because the control of the model is achieved using function calls in the Verilog rather than by toggling signals in the traditional HDL fashion. We must then declare a long list of registers in the Verilog code to store various values as they are used by the simulation. These registers are far too numerous to discuss in detail in this document, but are shown in the example design.

```
module bfm_test
    (
    input clk,
    input rst_n,
    input sys_rst_n,
    input init_done,
    output interrupt
    );
```

The Bus Functional Models are extremely configurable, and one of the options is the level of textual reporting that the model will output to the console in the simulator. In the supplied design we have enabled the verbose output of the model, which will show not only information about each transaction as it is received from the master, but also about the operation of each of the five AXI4 channels in isolation. If your simulation becomes too verbose, you can re-configure these options to suit your specific requirements. Here we can begin to see how to call the various functions to control the BFM, using the "." notation applied to the instance name that we defined at the top of the file. If the simulation contained multiple BFMs, it would be possible to control each one individually using this method.

```
//Process setting constants and internal variables APIs for BFMs
initial
    begin
    //AXI BFM Channel Level Info
    `AXI4_S_BFM_0.set_channel_level_info(1);
    `AXI4_S_BFM_0.set_function_level_info(1);
    end
```

AXI4 Slave BFM – Channel Function Calls

The rest of the file in the supplied example design is configured in a way to make the simulation quick and easy to use, but not necessarily configured in such a way that would reflect real hardware in a genuine system. During the development of a custom AXI4 master, it is often desirable to have the BFM always respond in a known way so that the results can quickly be analysed in the simulation waveform viewer.

The Verilog code required to manage a write transaction from the AXI4 master is split into three phases, each represented by a BFM function call. These three phases represent the functionality for the three AXI4 write channels; "write address", "write data" and "write response".

The first of the function calls is called "RECEIVE_WRITE_ADDRESS" and, as the name suggests, instructs the BFM slave to listen for an incoming transaction on the write address channel. The code is shown here, and only requires one fixed parameter which is the ID field. In our design we made the decision to lock the ID field in the master to ID=0, and therefore we must instruct the AXI4 BFM to listen for a transaction which uses that ID. The rest of the parameters in this function call are outputs, and store the values of the various transaction fields which are received by the BFM from the AXI4 master. These fields can be analysed and acted upon in the testbench by the user if they desire, but in our case we shall not do anything too clever with them apart from view their values in the simulation waveform viewer.

```
//AXI4 Slave Channel Write RECEIVE_WRITE_ADDRESS API
`AXI4_S_BFM_0.RECEIVE_WRITE_ADDRESS
    (
    AXI4_S_BFM_0_ID,
    `IDVALID_FALSE,
    AXI4_S_BFM_0_W_ADDR,
    AXI4_S_BFM_0_W_BURST_LENGTH,
    AXI4_S_BFM_0_W_BURST_SIZE,
    AXI4_S_BFM_0_W_BURST_TYPE,
    AXI4_S_BFM_0_W_LOCK,
    AXI4_S_BFM_0_W_CACHE,
    AXI4_S_BFM_0_W_PROT,
    AXI4_S_BFM_0_W_REGION,
    AXI4_S_BFM_0_W_QOS,
    AXI4_S_BFM_0_AWUSER,
    AXI4_S_BFM_0_W_IDTAG
    );
```

Once the addressing information has been received, we can call the next AXI4 slave BFM function "RECEIVE_WRITE_BURST". The name of this function can be a little misleading in some cases, because the same function is used even if the incoming transaction is not actually a burst (i.e. a single beat transactions could be considered as a burst transaction with a burst length of one). Again, we could use the various incoming fields to allow the simulation to make intelligent decisions and respond in different ways to different transactions, but in our example we are going to keep things simple. The last function call is to instruct the BFM to send a "write response" via the third AXI4 channel. This is the simplest function call of all, and requires very few parameters to be passed into the function. The combination of these three functions are all that is required to make the BFM respond to any write transaction from our AXI4 custom master, and the three function calls are combined into a Verilog "forever" block, inside an "initial" block, with a few added lines to detect the rising edges of the clock and reset signals. *(You may notice some "fork" and "join" keywords in the Verilog code. Don't worry; this is simply to allow the various sections of the code to run concurrently rather than sequentially)*

```
//AXI4 Slave Channel Write WRITE_BURST API
`AXI4_S_BFM_0.RECEIVE_WRITE_BURST
    (
    AXI4_S_BFM_0_W_ADDR,
    AXI4_S_BFM_0_W_BURST_LENGTH,
    AXI4_S_BFM_0_W_BURST_SIZE,
    AXI4_S_BFM_0_W_BURST_TYPE,
    AXI4_S_BFM_0_W_DATA,
    AXI4_S_BFM_0_DATA_SIZE,
    AXI4_S_BFM_0_W_USER
    );
```

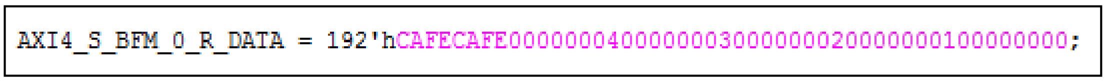
```
//AXI4 Slave Channel Write SEND_WRITE_RESPONSE API
`AXI4_S_BFM_0.SEND_WRITE_RESPONSE
    (
    AXI4_S_BFM_0_W_IDTAG,
    `RESPONSE_OKAY,
    S_BUSER
    );
```

The full listing of this section of the code is shown below, and the structure of the BFM stimuli should be clear to understand.

```
// Write Transaction Process for APIs
    initial
        begin
        //Check for the de-assertion(0) and assertion(1)
        //of the reset on rising edge of clock.
        wait(rst_n === 0) @(posedge clk);
        wait(rst_n === 1) @(posedge clk);

        fork
            begin
            AXI4_S_BFM_0_ID = 0;  // Set up the transaction ID to listen for.
            forever
                begin
                //AXI4 Slave Channel Write_Burst APIs
                //AXI4 Slave Channel Write RECEIVE_WRITE_ADDRESS API

                `AXI4_S_BFM_0.RECEIVE_WRITE_ADDRESS
                    (
                    AXI4_S_BFM_0_ID,
                    `IDVALID_FALSE,
                    AXI4_S_BFM_0_W_ADDR,
                    AXI4_S_BFM_0_W_BURST_LENGTH,
                    AXI4_S_BFM_0_W_BURST_SIZE,
                    AXI4_S_BFM_0_W_BURST_TYPE,
                    AXI4_S_BFM_0_W_LOCK,
                    AXI4_S_BFM_0_W_CACHE,
                    AXI4_S_BFM_0_W_PROT,
                    AXI4_S_BFM_0_W_REGION,
                    AXI4_S_BFM_0_W_QOS,
                    AXI4_S_BFM_0_AWUSER,
                    AXI4_S_BFM_0_W_IDTAG
                    );

                @(posedge clk);
                @(posedge clk);

                //AXI4 Slave Channel Write WRITE_BURST API
                `AXI4_S_BFM_0.RECEIVE_WRITE_BURST
                    (
                    AXI4_S_BFM_0_W_ADDR,
                    AXI4_S_BFM_0_W_BURST_LENGTH,
                    AXI4_S_BFM_0_W_BURST_SIZE,
                    AXI4_S_BFM_0_W_BURST_TYPE,
                    AXI4_S_BFM_0_W_DATA,
                    AXI4_S_BFM_0_DATA_SIZE,
                    AXI4_S_BFM_0_W_USER
                    );

                @(posedge clk);
                @(posedge clk);

                //AXI4 Slave Channel Write SEND_WRITE_RESPONSE API
                `AXI4_S_BFM_0.SEND_WRITE_RESPONSE
                    (
                    AXI4_S_BFM_0_W_IDTAG,
                    `RESPONSE_OKAY,
                    S_BUSER
                    );
                end
            end
        join
        end
```

Using our simple Verilog code, the testbench / simulation doesn't actually do anything with the write data when it is received by the BFM, but it is sufficient for our needs because we only wish to verify that the AXI4 master is correctly generating valid AXI transactions by observing them in a simulation waveform window. In addition to observing the signals manually in simulation, one of the most powerful features of the Bus Functional Model is that it will report violations in the AXI specification if it has been wrongly implemented in a user's design. For example; during the development of the attached reference design, I managed to accidentally violate the AXI reset requirements by toggling the AXI resetN signal in the top level testbench out of synchronisation with the clock. Asynchronous resets are not permitted according to the AXI4 specification, and the BFM immediately reported it. Useful!

When controlling the read transactions on the BFM, the Verilog code is almost identical to the example we studied for the write transaction. The changes are that two different function calls are used which control the two AXI4 read channels; "Read Address", and "Read Data". The function calls are named "RECEIVE_READ_ADDRESS" and "SEND_READ_BURST", and have a similar parameter list to that which was seen before. Key to the correct operation is one parameter which was not present in the function calls for the write transaction, called "AXI4_S_BFM_0_R_DATA". This parameter is pre-initialised in the Verilog code and represents the data which will be sent by the BFM when it receives a read transaction request. Here we can see an example of how the data would be sent for a burst of 6 data beats, each 32 bits in size. The parameter has been initialised as a 192 bit hexadecimal value (i.e. 6 beats x 32 bits = 192 bits), which sends the hexadecimal values "CAFECAFE", "00000004", "00000003", "00000002", "00000001", and "00000000" to represent the six beats of the burst. For larger or smaller bursts, the values pre-initialised here would either be repeated or unused.



```
AXI4_S_BFM_0_R_DATA = 192'hCAFECAFE000000040000000300000002000000010000000 0;
```

An example of the Verilog code required to implement a response to a read transaction is shown below, and the two API calls for "RECEIVE_READ_ADDRESS" and "SEND_READ_BURST" can be clearly seen. Once again, the BFM API calls have been implemented inside a Verilog "forever" loop, which allows the BFM to respond to as many AXI4 transactions are generated by our custom master, without any modifications to the BFM code.

```
// Read Transaction Process for APIs serial or parallel
    initial
        begin
        //Check for the deassertion(0) and assertion(1)
        //of the reset on rising edge of clock.
        wait(rst_n === 0) @(posedge clk);
        wait(rst_n === 1) @(posedge clk);

        fork
            begin
            forever
                begin
                AXI4_S_BFM_0_R_DATA = 192'hCAFECAFE0000000400000003000000020000000100000000;

                //AXI4 Slave Channel Read Burst APIs
                //AXI4 Slave Channel Read RECEIVE_READ_ADDRESS API
                `AXI4_S_BFM_0.RECEIVE_READ_ADDRESS
                    (
                    0,  // Listen for transactions with ID = 0000
                    `IDVALID_FALSE,
                    AXI4_S_BFM_0_R_ADDR,
                    AXI4_S_BFM_0_R_BURST_LENGTH,
                    AXI4_S_BFM_0_R_BURST_SIZE,
                    AXI4_S_BFM_0_R_BURST_TYPE,
                    AXI4_S_BFM_0_R_LOCK,
                    AXI4_S_BFM_0_R_CACHE,
                    AXI4_S_BFM_0_R_PROT,
                    AXI4_S_BFM_0_R_REGION,
                    AXI4_S_BFM_0_R_QOS,
                    AXI4_S_BFM_0_ARUSER,
                    AXI4_S_BFM_0_R_IDTAG
                    );

                @(posedge clk);
                @(posedge clk);

                //AXI4 Slave Channel Read SEND_READ_BURST API
                `AXI4_S_BFM_0.SEND_READ_BURST
                    (
                    AXI4_S_BFM_0_R_IDTAG,
                    AXI4_S_BFM_0_R_ADDR,
                    AXI4_S_BFM_0_R_BURST_LENGTH,
                    AXI4_S_BFM_0_R_BURST_SIZE,
                    AXI4_S_BFM_0_R_BURST_TYPE,
                    AXI4_S_BFM_0_R_LOCK,
                    AXI4_S_BFM_0_R_DATA,
                    AXI4_S_BFM_0_RUSER
                    );
                end
            end
        join
        end
```

AXI4 Slave BFM – The Overall Structure

The full version of the Verilog BFM stimuli is too lengthy to reproduce in this document on a single page, but in the interests of completeness we shall briefly examine an edited version. The areas shown with the tags “<code_removed>” below are purely to save space in this document and to demonstrate the overall structure of the Verilog module. The removed code has described in detail above; please see the provided example files for more details.

```
`timescale 1 ns/10 ps

`define AXI4_S_BFM_0 axi_master_tb.design_1_wrapper_inst.design_1_i.AXI4_S_BFM_0.cdn_axi4_slave_bfm_inst

module bfm_test
    (
    input clk,
    input rst_n,
    input sys_rst_n,
    input init_done,
    output interrupt
    );

<CODE REMOVED - REGISTER DEFINITIONS>

//Process setting constants and internal variables APIs for BFMs
    initial
        begin
        //AXI BFM Channel Level Info
        `AXI4_S_BFM_0.set_channel_level_info(1);
        `AXI4_S_BFM_0.set_function_level_info(1);
        end

    // Write Transaction Process for APIs serial or parallel
    initial
        begin
        fork
            begin
            AXI4_S_BFM_0_ID = 0;  // Set up the transaction ID to listen for.
            forever
                begin
                `AXI4_S_BFM_0.RECEIVE_WRITE_ADDRESS
                    (
                           <CODE REMOVED – FUNCTION PARAMETERS>
                    );

                //AXI4 Slave Channel Write WRITE_BURST API
                `AXI4_S_BFM_0.RECEIVE_WRITE_BURST
                    (
                           <CODE REMOVED – FUNCTION PARAMETERS>
                    );

                //AXI4 Slave Channel Write SEND_WRITE_RESPONSE API
                `AXI4_S_BFM_0.SEND_WRITE_RESPONSE
                    (
                           <CODE REMOVED – FUNCTION PARAMETERS>
                    );
                end
            end
        join
        end

    // Read Transaction Process for APIs serial or parallel
    initial
        fork
            begin
            forever
                begin
                AXI4_S_BFM_0_R_DATA = 192'hCAFECAFE000000040000000300000002000000010000000;

                //AXI4 Slave Channel Read Burst APIs
                //AXI4 Slave Channel Read RECEIVE_READ_ADDRESS API
                `AXI4_S_BFM_0.RECEIVE_READ_ADDRESS
                    (
                           <CODE REMOVED – FUNCTION PARAMETERS>
                    );

                //AXI4 Slave Channel Read SEND_READ_BURST API
                `AXI4_S_BFM_0.SEND_READ_BURST
                    (
                           <CODE REMOVED – FUNCTION PARAMETERS>
                    );
                end
            end
        join
        end
endmodule
```

AXI4 Signal Names

```
--  AXI4 Clock and Reset
m_axi_aclk       : in  std_logic;
m_axi_aresetn    : in  std_logic;
--  AXI4 Read Address Channel
m_axi_arready    : in  std_logic;
m_axi_arvalid    : out std_logic;
m_axi_araddr     : out std_logic_vector(31 downto 0);
m_axi_arid       : out std_logic_vector (3 downto 0);
m_axi_arlen      : out std_logic_vector (7 downto 0);
m_axi_arsize     : out std_logic_vector (2 downto 0);
m_axi_arburst    : out std_logic_vector (1 downto 0);
m_axi_arlock     : out std_logic;
m_axi_arcache    : out std_logic_vector (3 downto 0);
m_axi_arprot     : out std_logic_vector (2 downto 0);
m_axi_arqos      : out std_logic_vector (3 downto 0);
m_axi_arregion   : out std_logic_vector (3 downto 0);
--  AXI4 Read Data Channel
m_axi_rready     : out std_logic;
m_axi_rvalid     : in  std_logic;
m_axi_rdata      : in  std_logic_vector(data_width-1 downto 0);
m_axi_rresp      : in  std_logic_vector(1 downto 0);
m_axi_rid        : in  std_logic_vector (3 downto 0);
m_axi_rlast      : in  std_logic;
-- AXI4 Write Address Channel
m_axi_awready    : in  std_logic;
m_axi_awvalid    : out std_logic;
m_axi_awaddr     : out std_logic_vector(31 downto 0);
m_axi_awid       : out std_logic_vector (3 downto 0);
m_axi_awlen      : out std_logic_vector (7 downto 0);
m_axi_awsize     : out std_logic_vector (2 downto 0);
m_axi_awburst    : out std_logic_vector (1 downto 0);
m_axi_awlock     : out std_logic;
m_axi_awcache    : out std_logic_vector (3 downto 0);
m_axi_awprot     : out std_logic_vector (2 downto 0);
m_axi_awqos      : out std_logic_vector (3 downto 0);
m_axi_awregion   : out std_logic_vector (3 downto 0);
-- AXI4 Write Data Channel
m_axi_wready     : in  std_logic;
m_axi_wvalid     : out std_logic;
m_axi_wid        : out std_logic_vector(3 downto 0);
m_axi_wdata      : out std_logic_vector(data_width-1 downto 0);
m_axi_wstrb      : out std_logic_vector((data_width/8)-1 downto 0);
m_axi_wlast      : out std_logic;
-- AXI4 Write Response Channel
m_axi_bready     : out std_logic;
m_axi_bvalid     : in  std_logic;
m_axi_bresp      : in  std_logic_vector(1 downto 0);
m_axi_bid        : in  std_logic_vector (3 downto 0);
```

The master AXI4 signal names are completely flexible from the point of view of the VHDL design. During the creation of a Xilinx IP block, the Vivado tools can be used to map each AXI4 signal onto the signal name that the designer used when creating the IP. The signal names shown here are recommended when designing a AXI4 master in VHDL. Using these signal names will allow the Vivado design tools to automatically detect the signal names during the "create and package IP" step, which will save time and effort for the user. In this example you will see that the "data_width" parameter has been used when declaring some of the AXI4 data signals. This is a generic in the VHDL code, allowing the design to be flexibly converted in terms of data width by adjusting the value in the "generic map" when the entity is instantiated from the level of hierarchy above. *Note: the AXI "USER" signals are not shown in this list, but can easily be added if required by your design.*

Controlling the AXI4 Transactions

In the same way as we implemeted for the AXI4-lite slave, the provided example shows the five AXI4 channels controlled by a top level finite state machine. The design is intentionally basic to enable it to be understood quickly, and is specifically designed to be as flexible as possible to adapt. As such, there are no complex and advanced features such as burst alignment detection, and data width vs address alignment checking. All of these additional features can be implemented in the controlling top level, without any need to modify the five VHDL entities which implement the AXI channels. These more advanced features are much easier to add than to remove, and they are very application specific so are best not implemented unless they're needed for your design. It is also important to remember that additional functionality will increase the logic size of the custom AXI master, which may be undesirable for some users. In line with the famous saying; less is often more.

User Interface

The purpose of designing a flexible custom AXI master in the first place is to remove the onus of understanding the AXI4 signalling and protocol for all future designs. The goal is therefore to create a simple user interface that can be easily integrated into other IP blocks / designs, without requiring any ongoing knowledge of the AXI4 specification. In the example provided, the user interface is kept to a bare minimum whilst retaining the ability to manage full length burst transactions and various data sizes. A pair of FIFOs has been implemented to store the data that might be sent and received by one of the high performance bursts. The user interface reflects this and includes the traditional “full” and “empty” flags for both read and write data. A simple pair of “enable” signals allows write data to be pushed into the FIFO and read from it, a “Read Not Write” (RNW) signal has been included, and the transaction can be initiated with the “go” signal. Simple “error”, “busy” and “done” outputs are also provided. A signal for choosing fixed address bursts vs incrementing address bursts is provided. Wrapping burst support has not been included in this example in the interests of code simplicity, and to facilitate easy modification of the supplied example design. One final signal has been implemented to allow the FIFOs to be cleared in the case of unintentional writes / reads.

Here is the complete list of ports which represent the user interface:

```
-- User signals
go                   : in  std_logic;
RNW                  : in  std_logic;
error                : out std_logic;
busy                 : out std_logic;
done                 : out std_logic;
address              : in  std_logic_vector(31 downto 0);
write_data           : in  std_logic_vector(data_width-1 downto 0);
read_data            : out std_logic_vector(data_width-1 downto 0);
burst_length         : in  std_logic_vector(7 downto 0);
burst_size           : in  std_logic_vector(6 downto 0);
increment_burst      : in  std_logic;  -- 1 = incrementing, 0 = mailbox / FIFO style
clear_data_fifos     : in  std_logic;
write_fifo_en        : in  std_logic;
write_fifo_empty     : out std_logic;
write_fifo_full      : out std_logic;
read_fifo_en         : in  std_logic;
read_fifo_empty      : out std_logic;
read_fifo_full       : out std_logic;
```

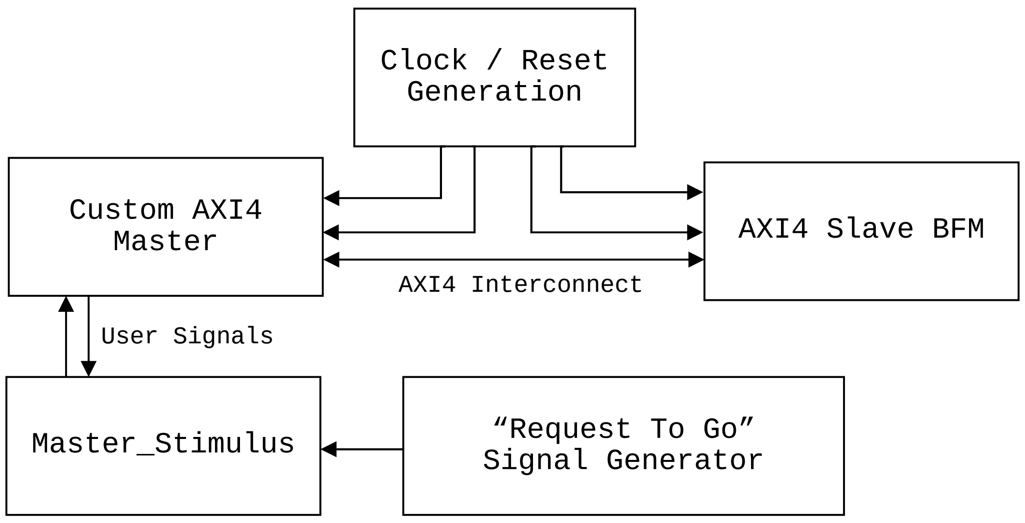
Test User Stimuli

In a genuine design, the custom AXI4 master would most likely be controlled by a state machine or other block of “intelligent” logic within the user’s HDL code. Given that the master must be tested in simulation to verify that it generates transactions correctly, we must create a form of stimuli in our test design which will mimic the controlling section of a real design. At this stage of development of the custom master, it is also important to thoroughly test the master’s ability to generate transactions with many different burst lengths. To fulfil these requirements, the provided files include a “master_stimulus.vhd” VHDL entity. This is a state machine which has the ability to generate single beat transactions and also burst transactions of every length between 2 and 256 beats, and can also perform a series of transactions which cover a range of burst lengths, sequentially. This VHDL entity has inputs and outputs which precisely match the "user" signals of the custom AXI master, and is designed to operate completely autonomously once instructed to start. The state machine starts with the smallest burst length specified in a range by the user. It writes the correct number of data values into the write FIFO, and then asserts the user control signals which instructs the custom master to generate a transaction of the matching burst size. Assuming that the write transaction is successful, the state machine then generates a read transaction at the same destination address and waits for it to complete. It then reads the data values from the read FIFO, by generating sufficient clock cycles with the "read_fifo_enable" signal asserted. The state machine then performs another iteration of the same sequence, but this time using a larger burst length. The process is repeated until the burst length reaches maximum value in the range specified by the user. The burst length range limits are communicated to the "master_stimulus" VHDL code using a pair of VHDL Generic values, "MINIMUM_BURST_LENGTH" and "MAXIMUM_BURST_LENGTH". These can be either hard coded, or passed down to the master_stimulus VHDL entity from the top level testbench in the hierarchy above it. There is also one final input signal, "request_to_go", which instructs the state machine to leave its idle state and start the sequence of operations described above.

The master_stimulus.vhd file has a dual purpose. During the initial tests it will be used in a simulation environment alongside the BFM models described previously. Later on this design process, the same master_stimulus VHDL will be used in a real test design running on a board. For this reason, the master_stimulus.vhd file has been written using synthesisable code and will allow us to generate a functional hardware block in the Zynq-7000 programmable logic.

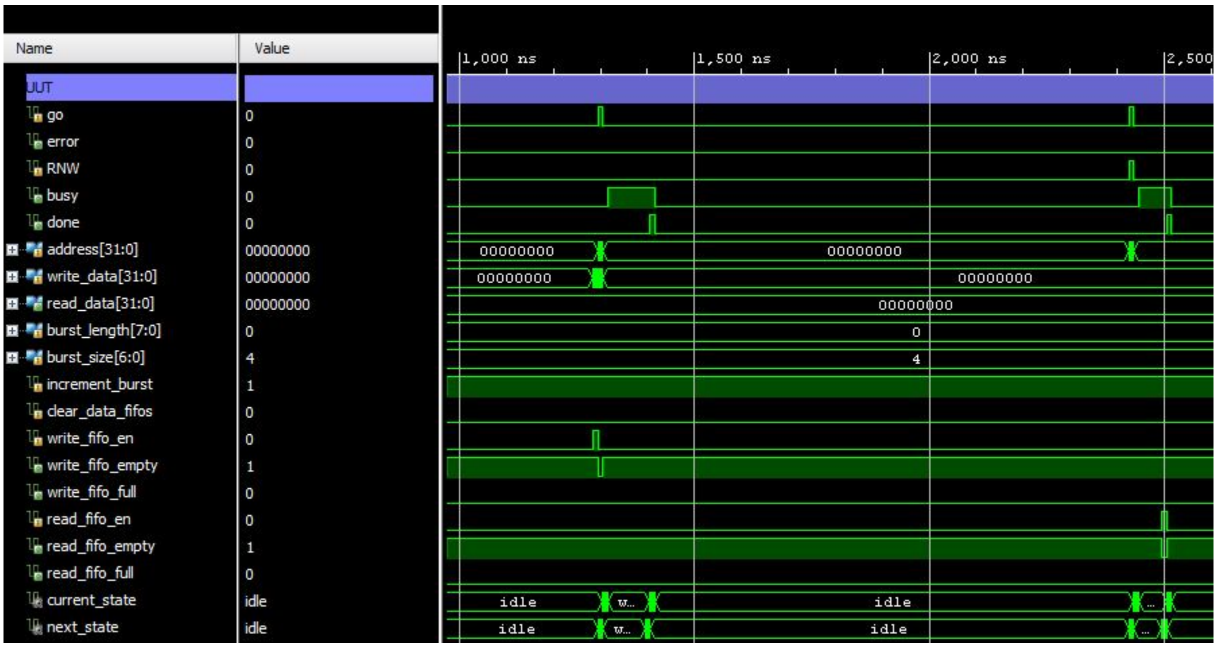
In the real hardware design / board testing, the "request_to_go" signal on the "master_stimulus" entity will be driven by another part of the design, but for simulation purposes we need a way to simply assert the signal. For this reason, the provided example design also contains an extremely simple VHDL entity containing a state machine. The state machine checks that the "busy" signal on the "master_stimulus" block to ensure that it is ready, and then asserts "request_to_go" signal.

The overall structure of the simulation environment is therefore as follows:



AXI4 Transaction Simulation

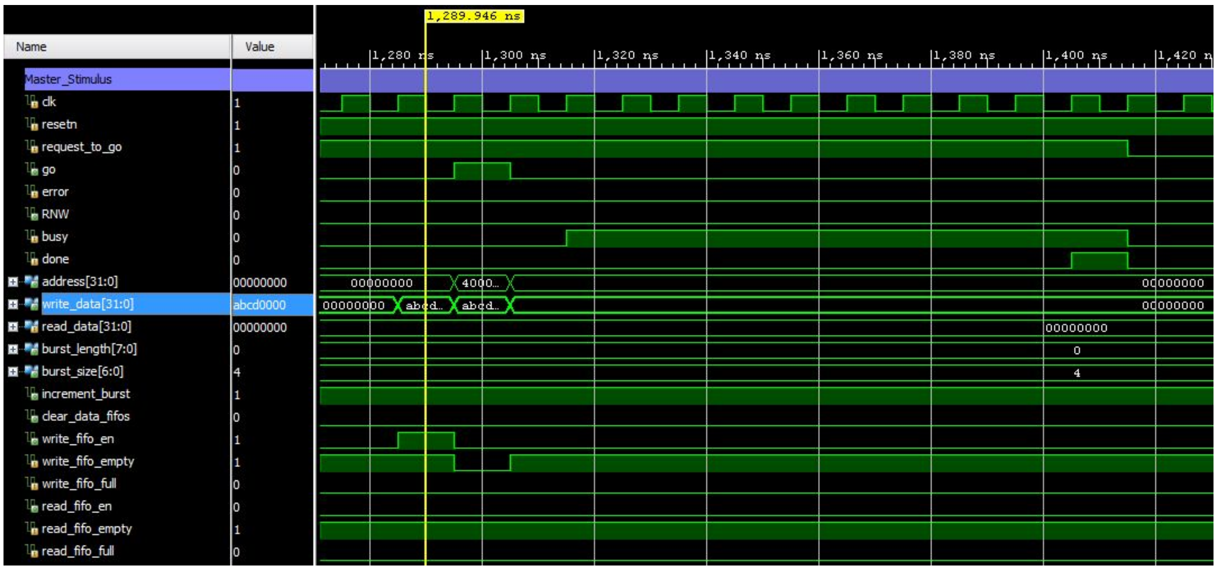
With the test environment in place, we can now attempt our first simulation run of the custom master. For the initial test we shall configure the master_stimulus block's parameters with the settings of MINIMUM_BURST_LENGTH=1 and MAXIMUM_BURST_LENGTH=1. This should generate just a single pair of transactions, one write and one read, each with a single data beat.



Name	Value
UUT	
go	0
error	0
RNW	0
busy	0
done	0
address[31:0]	00000000
write_data[31:0]	00000000
read_data[31:0]	00000000
burst_length[7:0]	0
burst_size[6:0]	4
increment_burst	1
clear_data_fifos	0
write_fifo_en	0
write_fifo_empty	1
write_fifo_full	0
read_fifo_en	0
read_fifo_empty	1
read_fifo_full	0
current_state	idle
next_state	idle

Here we can see that the simulation has generated exactly what we expected. The two transactions are clearly visible; a write followed by a read. Let us now look more closely at each transaction, and study the AXI4 protocol at work.

User Signals – Single Beat Write Transaction

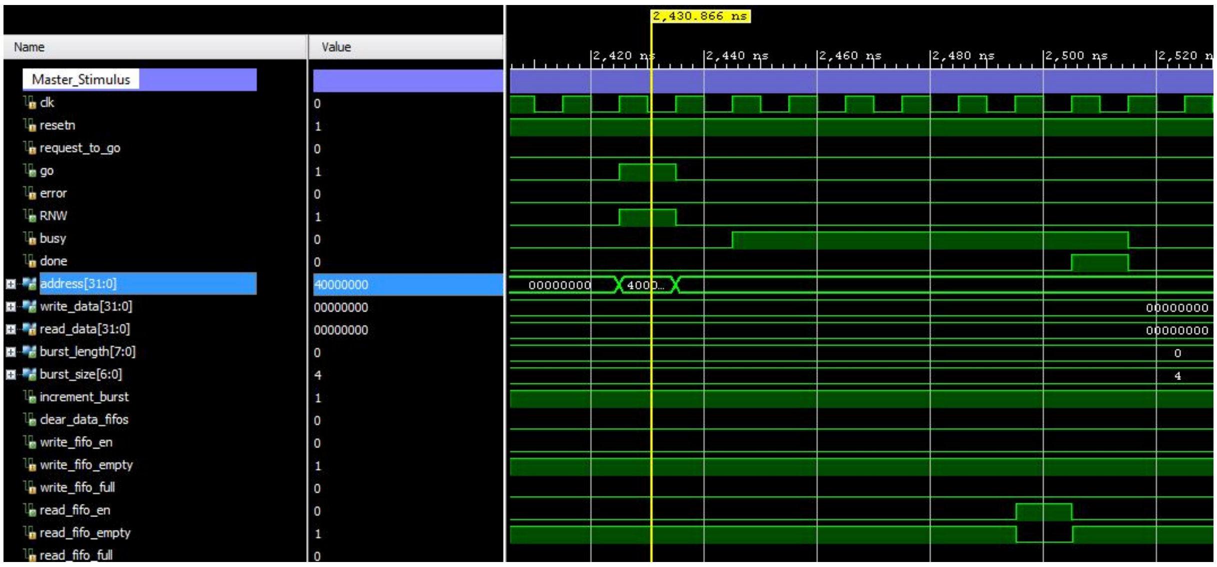


Name	Value
Master_Stimulus	
clk	1
resetn	1
request_to_go	1
go	0
error	0
RNW	0
busy	0
done	0
address[31:0]	00000000
write_data[31:0]	abcd0000
read_data[31:0]	00000000
burst_length[7:0]	0
burst_size[6:0]	4
increment_burst	1
clear_data_fifos	0
write_fifo_en	1
write_fifo_empty	1
write_fifo_full	0
read_fifo_en	0
read_fifo_empty	1
read_fifo_full	0

Here we can see the user signals for a write transaction. A single word of data is pushed into the write FIFO, and the "Go" signal is asserted on the next clock cycle.

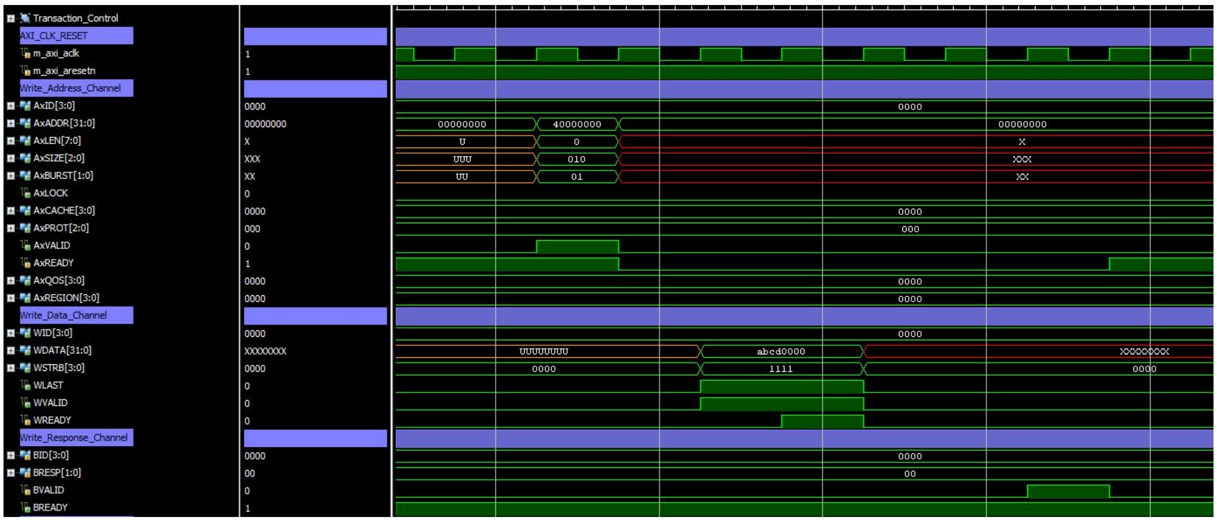
The "done" signal then indicates that the transaction completed successfully a number of cycles later, and the "busy" signal is de-asserted by the custom AXI master. The period of time in the middle is when the AXI transaction took place, and we shall examine that in more detail below.

User Signals – Single Beat Read Transaction



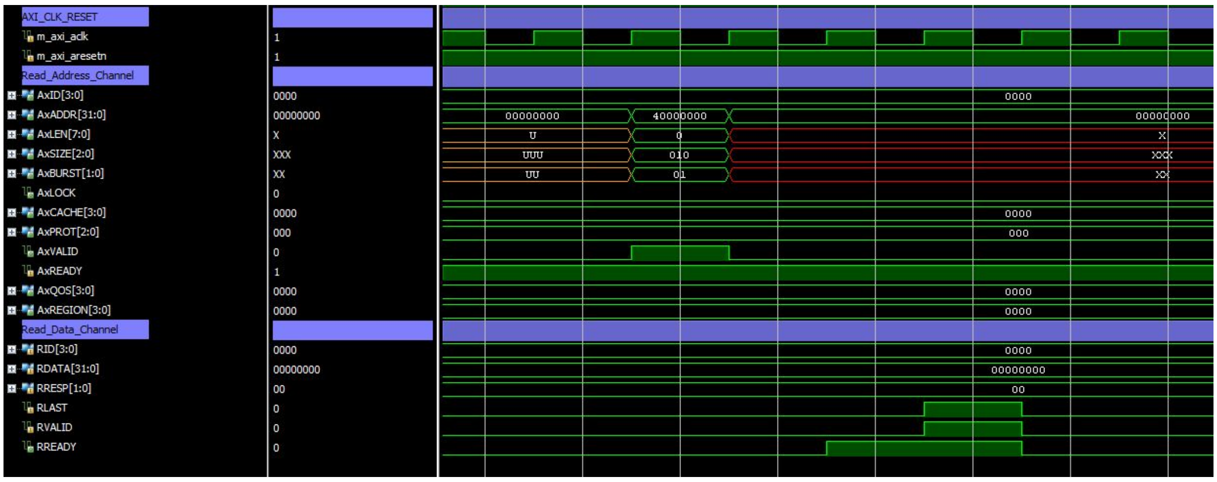
Here we can see the user signals for the corresponding read transaction. The address is sent along with the "Go" signal, and the "busy" output is asserted during the time that the AXI transaction was in progress (again, to be examined in more detail below). After several clock cycles, the "read_fifo_empty" signal is de-asserted by the AXI custom master, indicating that data has been received. The custom master then waits until the master_stimulus block has emptied the FIFO of all data (by asserting the read FIFO enable signal for the correct number of cycles), and then asserts the "done" signal to indicate the end of the operation.

AXI4 Signals – Single Beat Write Transaction



Here we can see the three channels pertaining to an AXI4 write transaction. The write address channel carries the destination address, and the length of the burst (AxLEN = 0, indicating a transfer of just one data beat). The write data channel shows the transfer of the single data beat (signified by the assertion of the WLAST signal on the first data beat), and also demonstrates how the AXI Ready/Valid handshaking works when one of the two parties is not ready to send / receive; The value of "ABCD0000" is only successfully transferred when both WVALID and WREADY are asserted. The write response channel indicates the end of the transaction, by sending the "success" response code "00" on BRESP. Once again, this response code is only valid when both BVALID and BREADY are both asserted in the same clock cycle.

AXI4 Signals – Single Beat Read Transaction

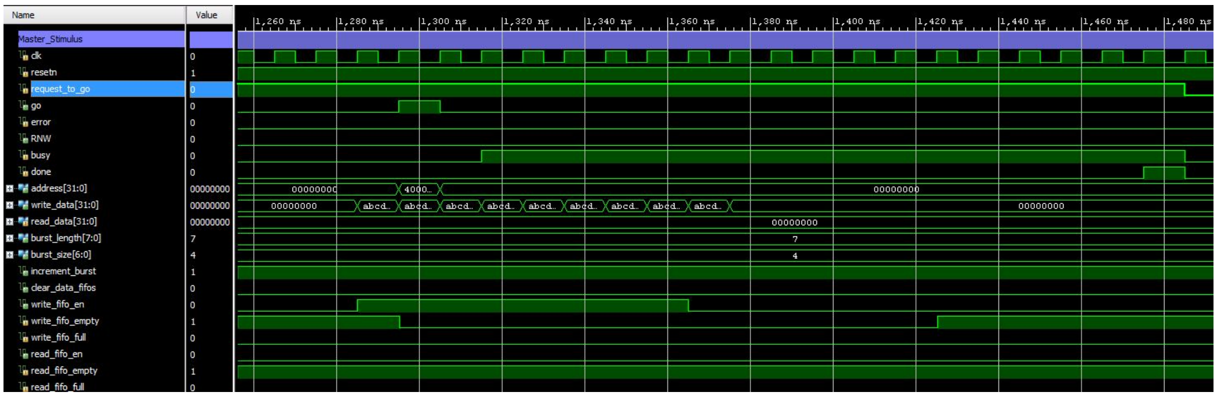


Here we can see the two channels which relate to the associated AXI4 read transaction. The read address channel carries identical information to that which we observed in the write transaction. The read data channel shows a single word of data being transferred, and once again the AXI Ready/Valid handshaking can be seen in operation. The data value transferred on this single beat transaction happens to be "00000000", and therefore no change in value is seen on the RDATA waveforms on either side of the transaction.

User Signals – Burst Write Transaction

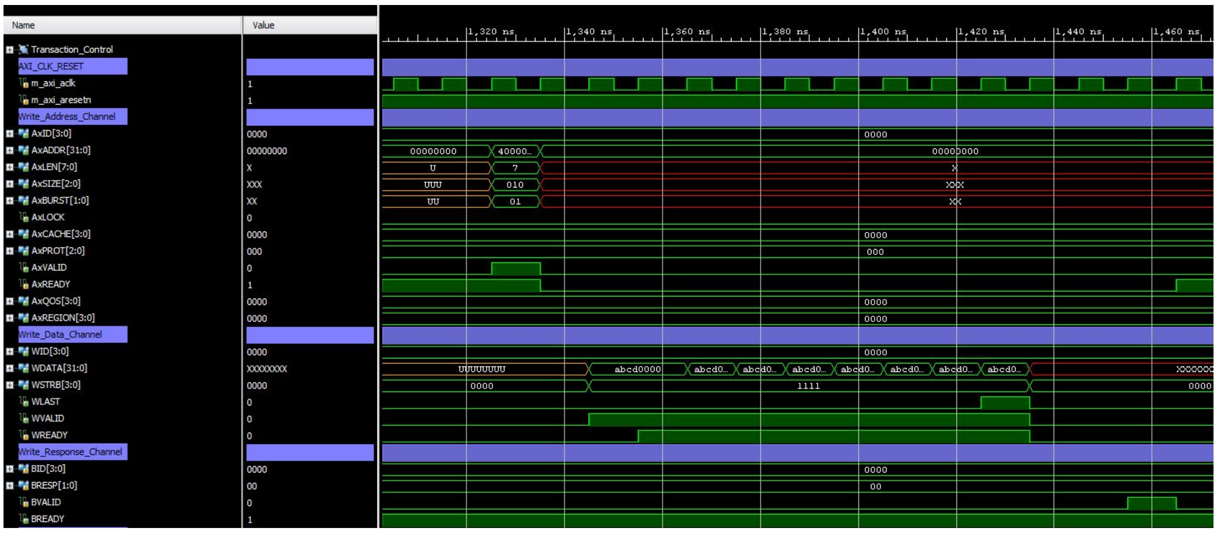
The single beat tests are operating correctly, so we must now check the operation of the custom master when burst transactions are requested. To check this we can use the flexible nature of the master stimulus block that we've created, and simply request a pair of write and read transactions with a burst length larger than one. For this experiment we will simply adjust the settings to MINIMUM_BURST_LENGTH=8 and MAXIMUM_BURST_LENGTH=8, and then re-run the simulation with no other modifications.

The first simulation screenshot is of the user signals relating to the write burst transaction. The master_stimulus block first pushes the eight data values into the write FIFO but, to increase efficiency, asserts the "go" signal after the first write in order to minimise the latency of the whole operation. The custom AXI master has been designed in such a way that it doesn't require all of the data values to be present in the write FIFO before a transaction can commence, and will control the downstream AXI4 handshaking signals to add wait cycles if the write FIFO is not populated quickly enough. Note: The waveforms show that there are nine different data values presented on the "write_data" input, but only eight of these are actually validated using the "write_fifo_en" signal. This is simply due to the design of the state machine within the master_stimulus block, which is designed to be efficient with clock cycles during the operation to populate the write FIFO.



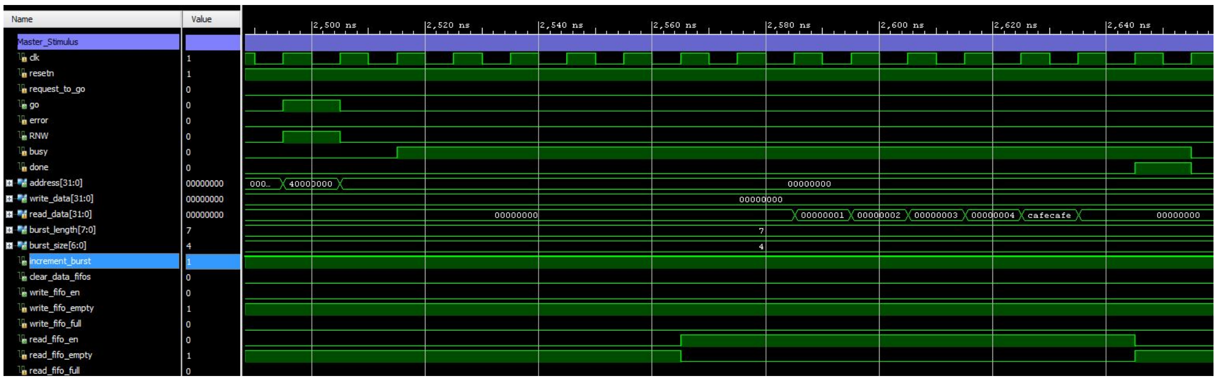
AXI4 Signals – Burst Write Transaction

The AXI4 signals waveform screenshot shows the corresponding transaction which is generated by the custom AXI master. The write address channel carries a different value on the "AxLEN" signal (AWLEN=7, denoting a burst of eight data beats), but this is the only difference from the single beat transaction example. The write data channel shows the eight data beats being sent across the WDATA signals, with a handshaking delay on the first beat due to the non-ready state of the WREADY signal. Note also the assertion of the WLAST signal only on the last data beat. The write response channel is identical to the example we reviewed for the single beat transaction.



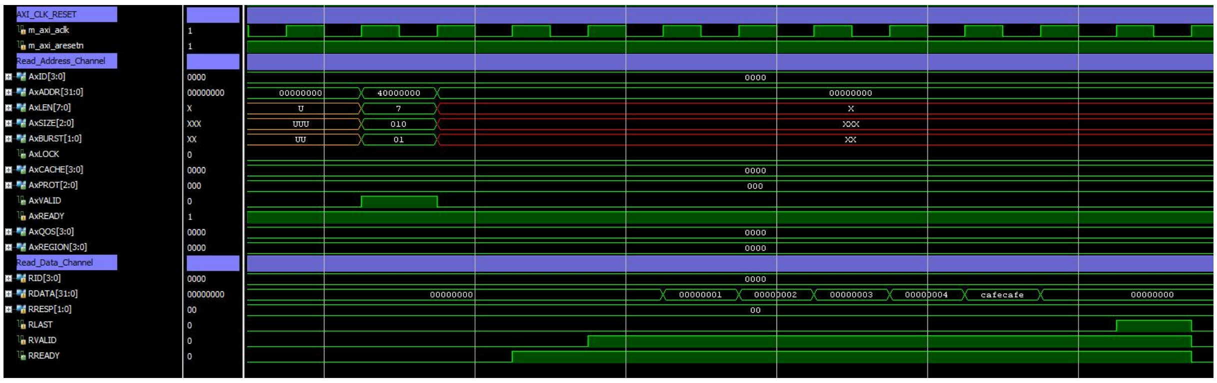
User Signals – Burst Read Transaction

For completeness, the final example shows the user signals for a burst read transaction. There is no major difference between this example and the single beat transaction example, except for the number of cycles that the read_fifo_empty signal remains de-asserted. When the master_stimulus block has read the eighth data word from the read FIFO, the “empty” signal is asserted and the “done” signal is pulsed high for one clock cycle. Note: To assist with user signal handshaking, the custom AXI master has been designed to hold the “done” signal in the asserted state until one cycle after “go” is de-asserted. This caters for situations where the upstream user logic is busy and may not be able to monitor the “done” signal for such a short, one clock cycle, period. An example of this scenario would be if the “go” and “done” signals were controlled by a processor GPIO peripheral. This handshaking functionality is not shown in these simulation screenshots.



AXI4 Signals – Burst Read Transaction

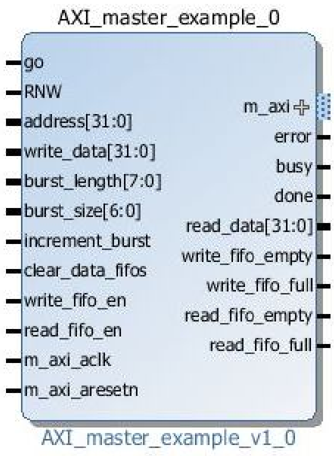
The final simulation waveforms shows the transfer of data across the AXI4 read address channel, and AXI4 read data channel. Lengthy explanation is not required, because the same rules and signal handshaking is used, as in the previous examples. The only peculiarity of note is the apparent lack of eight distinct data words on the RDATA lines. This is due to the BFM stimulus Verilog file only containing sufficient information to instruct the BFM to reply with 192 bits of data (6 words), and therefore the remaining beats were sent as zero values. If all eight data beats were required in the simulation, then the “bfm_test_stimulus.v” file would need to be modified to supply 256 bits of data rather than 192.



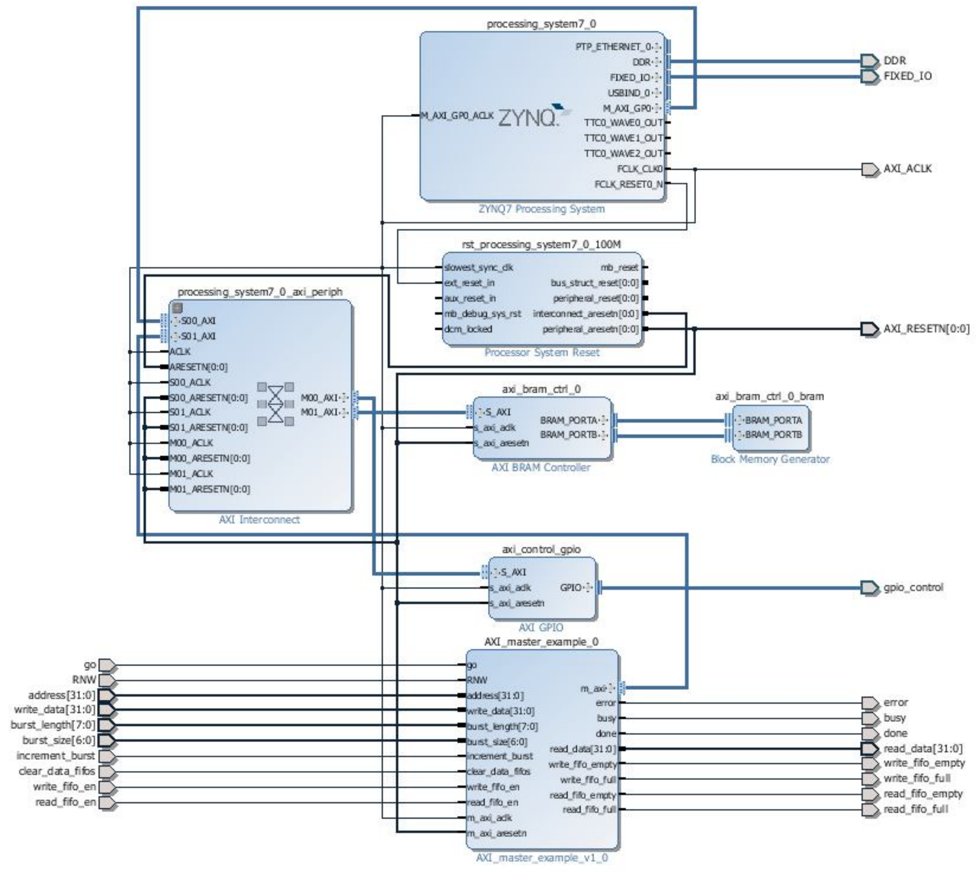
Further Simulation Possibilities

At this stage we could continue to simulate the design to test every possible combination of transactions. One example of this would be to adjust the MINIMUM_BURST_LENGTH and MAXIMUM_BURST_LENGTH parameters on the master_stimulus block, so as to create a series of burst transactions with increasing burst lengths. However in the interests of brevity we shall move to the next phase of this design and begin to implement the custom AXI master IP in real hardware on a development board.

Packaging the IP & Building a Test Project

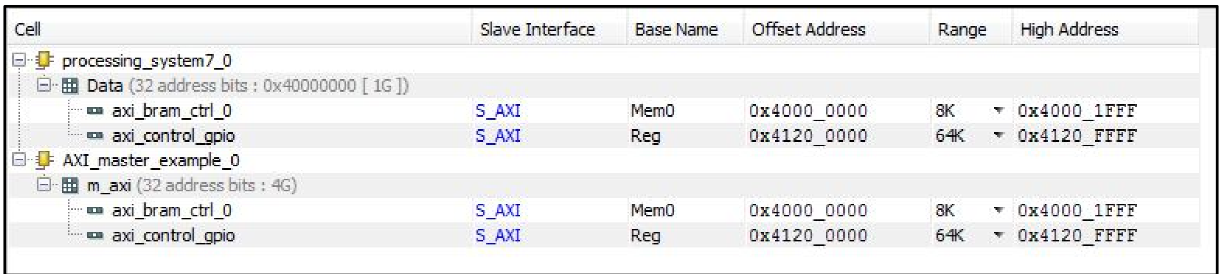


The next step of the design process is to package the IP to put it into a format that can be understood by the Xilinx Vivado block diagram GUI. This process is identical to the steps that were discussed in the previous application note for the custom AXI4-lite slave, and therefore will not be repeated here in any great detail. When the IP has been packaged, a symbol will be generated for the AXI4 master looking similar to the one shown here. The “M_AXI” port can be seen in the top right, with all of the individual AXI signals grouped into a single block diagram connection port. This allows the user to connect the master to other IP blocks in the Xilinx Vivado design in the same way that is possible for other IP from the catalogue. The user side ports remain fully custom, and can be connected either within the block diagram or routed to the HDL wrapper in the level of hierarchy above. It is the latter option which has been chosen for the example block diagram shown below.



The design shown is nothing more than our packaged custom master IP, which has been placed into a design with some standard Xilinx IP blocks from the catalogue. The blocks that were added to the design include the “Processing System” to represent the hardened PS section of the Zynq device, an 8KB block RAM, a GPIO, some reset sequencing logic, and the AXI crossbar interconnect structure. The BRAM will be used as the destination slave device, allowing our custom AXI master to read from and write to a simple memory. The Zynq processing system also has access to the BRAM, allowing us to set up test conditions to verify the correct operation of the custom master IP. The GPIO has been added to control a single pin, which has been routed to the HDL hierarchy level above so that it can control the “request_to_go” input on the master_stimulus block.

In this example we have assigned the base address of the BRAM to 0x40000000, and the base address of the GPIO to 0x41200000.

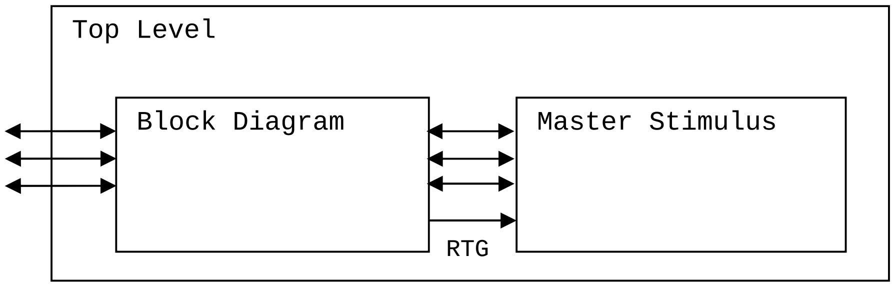


Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF
axi_control_gpio	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
AXI_master_example_0					
m_axi (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF
axi_control_gpio	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF

Note that the Address Editor now has two memory regions; one for each AXI master in the system. The first is used to assign addresses to slaves under the control of the Processing System (Cortex A9 cores), and the second is used to assign addresses to slaves under the control of our custom AXI master. In our design these two domains are served by the same AXI interconnect, and therefore the addresses are identical. These settings complete the block diagram for our test design.

Creating the Top Level HDL design

The Xilinx Vivado tools will automatically create a VHDL wrapper for the block diagram, and this can be used to quickly instantiate it into a top level design. It is therefore relatively simple to create a VHDL top level for our design, which simply instantiates the two main function blocks in the design; the block diagram, and the master_stimulus block. An example is provided in the supplied files but, for completeness, the overall test design looks like this.



The one important point to note here is that the “request_to_go” (RTG) signal has been connected to bit 0 of the GPIO output, and this will be important to us later when we begin to develop some test software. An example of this connection is shown in the VHDL code from the test design (highlighted in red text).

```
master_stimulus_instance : master_stimulus
    generic map
        (
        DATA_WIDTH => DATA_WIDTH,
        MINIMUM_BURST_LENGTH => MINIMUM_BURST_LENGTH,
        MAXIMUM_BURST_LENGTH => MAXIMUM_BURST_LENGTH
        )
    port map
        (
        clk => AXI_ACLK,
        resetn => AXI_RESETN,
        request_to_go => gpio_control_tri_o(0),
        go => go,
        done => done,
        busy => busy,
        error => error,
        RNW => RNW,
        address(31 downto 0) => address(31 downto 0),
        burst_length(7 downto 0) => burst_length(7 downto 0),
        burst_size(6 downto 0) => burst_size(6 downto 0),
        clear_data_fifos => clear_data_fifos,
        increment_burst => increment_burst,
        read_data(31 downto 0) => read_data(31 downto 0),
        read_fifo_empty => read_fifo_empty,
        read_fifo_en => read_fifo_en,
        read_fifo_full => read_fifo_full,
        write_data(31 downto 0) => write_data(31 downto 0),
        write_fifo_empty => write_fifo_empty,
        write_fifo_en => write_fifo_en,
        write_fifo_full => write_fifo_full
        );
```

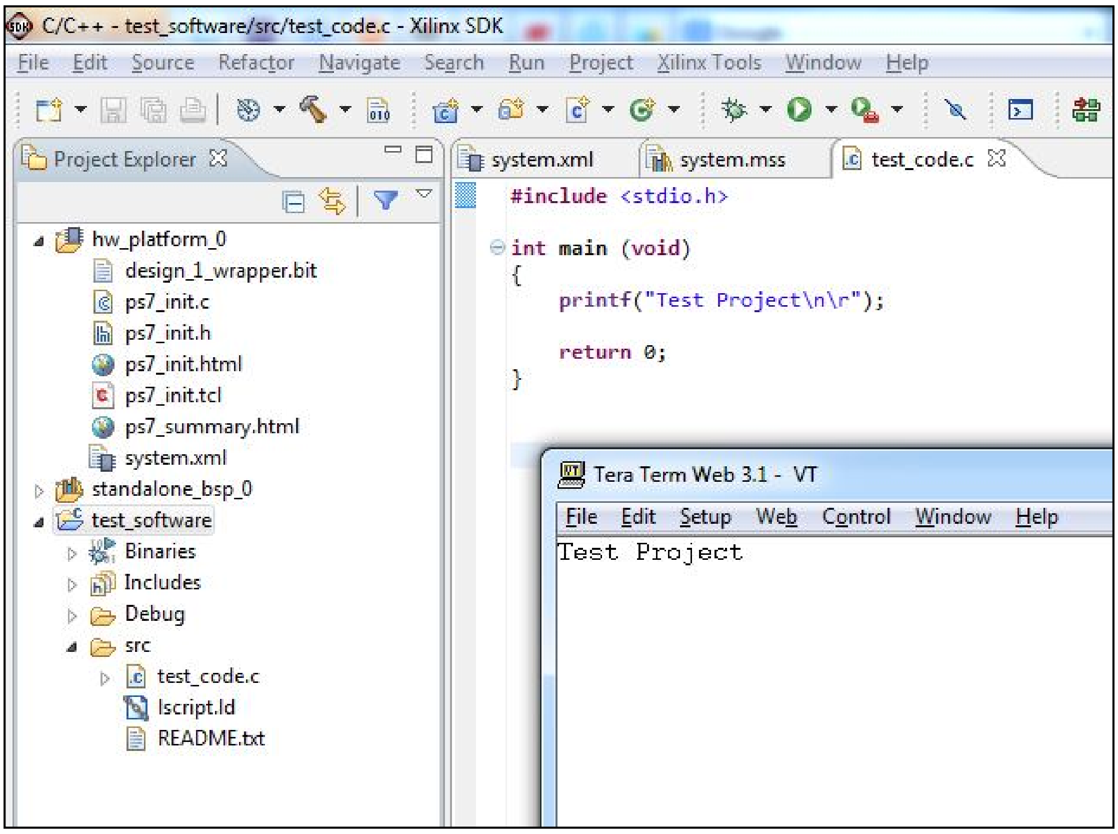
The usual flow in the Vivado tools to synthesise, implement, and create a bitstream for the design can now be followed. Once the bitstream has been generated, we can export the design as an HDF file, and then open the Software Development Kit (SDK) in preparation to write some test software code for the custom IP.

Writing Your First Lines of Test Software Code

Before writing any test code, it is necessary to create a standalone BSP for your software development. Create this in the usual way using the "File → New → Board Support Package" menu option. As always, we shall follow the golden rule of embedded design and verify that things are working properly on the board. We will therefore start with some simple software code and execute it on the board and so, in accordance with tradition, we shall start with a "Hello World" type application. To do this, create a software application project using the "File → New → Application Project" menu item, and choose the "Hello World" template.

```
#include <stdio.h>

int main (void)
{
        printf("Test Project\n\r");

        return 0;
}
```

Create a suitable linker script to link this code to the area of memory that suits your needs, and then run it on the board. Verify that your code is running on the board, remembering to download the bitstream to the board as appropriate.



```
#include <stdio.h>

int main (void)
{
    printf("Test Project\n\r");

    return 0;
}
```

```
Test Project
```

Initialise the GPIO

In order to control the “request_to_go” signal on the master_stimulus block, we must write some code to control the GPIO slave peripheral. The full details of the use of Xilinx GPIO drivers are covered in the “Zynq Workshop for Beginners” document, and therefore we shall treat the contents of the C code as assumed knowledge. We must also disable the caches on the cortex A9 processor so that we can write and read test values to/from the BRAM without the caches interfering with our results.

```
XStatus Status;
XGpio my_gpio_instance;
XGpio_Config *my_gpio_instance_config;

init_platform();

printf("Disable the caches to prevent skewing the results with BRAM reads / writes\n\r");
Xil_DCacheDisable();
Xil_ICacheDisable();

my_gpio_instance_config = XGpio_LookupConfig(XPAR_AXI_GPIO_0_DEVICE_ID);
Status = XGpio_CfgInitialize(&my_gpio_instance, my_gpio_instance_config,
my_gpio_instance_config->BaseAddress);
if (Status != XST_SUCCESS)
{
        printf("Failed to CfgInitialise.  Halting!\n\r");
        return (1);
}

XGpio_SetDataDirection(&my_gpio_instance, GPIO_CHANNEL_1, 0x00000000);

printf("Disable the AXI4 Custom Master\n\r");
XGpio_DiscreteWrite(&my_gpio_instance, GPIO_CHANNEL_1, 0x00000000);
```

The latter lines of code set the direction of the GPIO signals to outputs, and then ensure that all of the GPIO outputs are being driven to logic 0 before we begin.

Write some code to fill the BRAMs with known values

Before initiating the custom AXI master we must initialise the contents of the BRAM with known values; this will allow us to observe the changes that occur in the BRAM as a result of the custom AXI master in operation. At the same time, we should write some code that allows us to verify that we can correctly read and write from/to the BRAM using the processor. The following code is therefore designed to first clear the BRAMs with zeros and display the contents on screen, and then write a known pattern of data to the BRAM and once again read it back.

```
// Write zeros to fill the BRAM
printf("Writing zeros to fill the BRAM\n\r");
for (BRAM_offset = 0; BRAM_offset < (XPAR_AXI_BRAM_CTRL_0_S_AXI_HIGHADDR -
XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR)/4; BRAM_offset++)
{
        Xil_Out32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + (BRAM_offset*4), 0x00000000);
}

// Read back and display some of the test values
for (BRAM_offset = 0; BRAM_offset < 10; BRAM_offset++)
{
        temp_read_value = Xil_In32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + (BRAM_offset*4));
        printf("BRAM Address 0x%08X = 0x%08X\n\r", XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR +
(BRAM_offset*4), temp_read_value);
}

// Write sequential test values to fill the BRAM
printf("Writing sequential test values to fill the BRAM\n\r");
for (BRAM_offset = 0; BRAM_offset < (XPAR_AXI_BRAM_CTRL_0_S_AXI_HIGHADDR -
XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR)/4; BRAM_offset++)
{
        Xil_Out32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + (BRAM_offset*4), (0x12340000 + BRAM_offset));
}

// Read back and display some of the test values
for (BRAM_offset = 0; BRAM_offset < 10; BRAM_offset++)
{
        temp_read_value = Xil_In32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + (BRAM_offset*4));
        printf("BRAM Address 0x%08X = 0x%08X\n\r", XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR +
(BRAM_offset*4), temp_read_value);
}
```

Write some code to start the custom AXI master

The last remaining piece of code, and perhaps the most important one, is two lines which toggles bit zero of the GPIO, which is in turn connected to the “request_to_go” signal on the master_stimulus block. In theory, the resulting AXI burst transaction from the custom AXI master should happen so quickly that it will have completed long before the software can read back the BRAM to verify the result. It is therefore quite acceptable for us to immediately follow the GPIO control software with a loop to read back the contents of the BRAM.

```
printf("Enable the AXI4 Custom Master\n\r");
XGpio_DiscreteWrite(&my_gpio_instance, GPIO_CHANNEL_1, 0x00000001);
printf("Disable the AXI4 Custom Master\n\r");
XGpio_DiscreteWrite(&my_gpio_instance, GPIO_CHANNEL_1, 0x00000000);

printf("Read back the contents of the BRAM\n\r");
for (BRAM_offset = 0; BRAM_offset < 10; BRAM_offset++)
{
        temp_read_value = Xil_In32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + (BRAM_offset*4));
        printf("BRAM Address 0x%08X = 0x%08X\n\r", XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR +
(BRAM_offset*4), temp_read_value);
}
```

Important Note: It is vital to remember that the AXI burst transaction to the BRAM is happening in total isolation from the Cortex A9 processor. Therefore the processor has no knowledge that the transaction even took place. This is important because, had we left the processor's caches enabled, the re-read of the BRAM using software would reveal the previous values stored in the BRAM because that would be the data in the processor's caches. This is an incredibly simple and common mistake to make when designing a custom AXI master, and especially when integrating it and testing it within a larger processor system. You have been warned!

In a real design where caches would be used, there are other techniques available to the user which would allow the processor's caches to be selectively disabled for a given region of the address map. Other alternatives include the use of the Accelerated Coherency Port (ACP) in the Zynq architecture, which would allow the memory to remain cached but also allow cache coherency to be maintained. These techniques are beyond the scope of this document, but are explored elsewhere.

Testing the design in hardware

Everything is now in place for us to finally test the operation of the custom AXI master. In the SDK we can download the Programmable Logic Bitstream and the run our test software application on the processor in the usual way. The results clearly show that the custom AXI master is working, and the UART output of the board is shown and annotated below.

```
--AXI4 Custom Master Test--

Disable the AXI4 Custom Master
Writing zeros to fill the BRAM
BRAM Address 0x40000000 = 0x00000000
BRAM Address 0x40000004 = 0x00000000
BRAM Address 0x40000008 = 0x00000000
BRAM Address 0x4000000C = 0x00000000
BRAM Address 0x40000010 = 0x00000000
BRAM Address 0x40000014 = 0x00000000
BRAM Address 0x40000018 = 0x00000000
BRAM Address 0x4000001C = 0x00000000
BRAM Address 0x40000020 = 0x00000000
BRAM Address 0x40000024 = 0x00000000
Writing sequential test values to fill the BRAM
BRAM Address 0x40000000 = 0x12340000
BRAM Address 0x40000004 = 0x12340001
BRAM Address 0x40000008 = 0x12340002
BRAM Address 0x4000000C = 0x12340003
BRAM Address 0x40000010 = 0x12340004
BRAM Address 0x40000014 = 0x12340005
BRAM Address 0x40000018 = 0x12340006
BRAM Address 0x4000001C = 0x12340007
BRAM Address 0x40000020 = 0x12340008
BRAM Address 0x40000024 = 0x12340009
Enable the AXI4 Custom Master
Disable the AXI4 Custom Master
Read back the contents of the BRAM
BRAM Address 0x40000000 = 0xABCD0000
BRAM Address 0x40000004 = 0xABCD0001
BRAM Address 0x40000008 = 0xABCD0002
BRAM Address 0x4000000C = 0xABCD0003
BRAM Address 0x40000010 = 0xABCD0004
BRAM Address 0x40000014 = 0xABCD0005
BRAM Address 0x40000018 = 0xABCD0006
BRAM Address 0x4000001C = 0xABCD0007
BRAM Address 0x40000020 = 0x12340008
BRAM Address 0x40000024 = 0x12340009
--AXI4 Custom Master Test Complete--
```

Clear the Block RAMs, read back the values, then write test values to the BRAM and re-read them.

This verifies that the processor can correctly read and write to the BRAM, and also sets up a pattern in the BRAM so that we can see if it is modified later.

Toggle the "request_to_go" signal on the master stimulus block. This starts the operation of the custom AXI master.

Read back the BRAM to see what happened.
The first 8 addresses of the BRAM have been updated.

Conclusion & Invitation for user contributions

Success! We have implemented a custom AXI master and used it to generate AXI transactions controlled by user logic in the Zynq SoC. We have also demonstrated some basic methods to use software to initiate those AXI transactions.

The full source code for the custom AXI master is available for download, including all the testbenches, BFM stimuli, and software source code. To obtain this code, please visit the Architech / Silica GitHub repositories at https://github.com/Architech-Silica . Updates to this document will also be published as part of the GitHub repository.

As part of the community ethos of the Architech / Silica team, we welcome contributions in the form of hardware and software code. If you wish to participate and add your own features to this material, please fork the design from the GitHub repository, create your own branch containing the modifications, and submit your updates via a GitHub “Pull Request”.

Recommended Reading / Resources

- AMBA AXI and ACE Specification (Issue E) – ARM Infocenter. (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html)
- “Designing a Custom AXI-lite Slave Peripheral” – Rich Griffin, Architech / Silica EMEA. (http://www.architechboards.org/course/designing-custom-axi-lite-slave-peripheral)
- “Zynq Workshop For Beginners” – Rich Griffin, Architech / Silica EMEA. (http://www.architechboards.org/course/zynq-workshop-beginners)
- Architech / Silica GitHub Repositories (https://github.com/Architech-Silica)