AXI test master

G.J. van Loo, 28-9-2017

Introduction.

The danger with the AXI standard is that it seems so easy. But if you start implementing AXI components you will find that they can turn out very complex, especially if you want to have them operating with minimum dead cycles. The AXI test master is designed to help test AXI components by generating the most common AXI cycles.
This document assumes the user is somewhat familiar with the AXI bus protocol

Table of Contents

Basic operation.

The AXI master can be programmed to generate a great variety of AXI cycle. To do this it has a number of FIFOs (or queue what ever you want to call them) which are loaded with the type of AXI cycles the user wants to see. Then when the 'run' signal is asserted it will generate these cycles back-to-back, (as far as the ready signals allow).

It can generate the following type of cycles:

- Write address
- Write data
- Combined write address & data
- Read address

Besides those the user can also make read-response cycles.

For each cycle the user can specify how many wait states are inserted before the cycle is run. (From 0 to 255 cycles).

Besides all signals relevant to an AXI master the module has two more signals:

input run
If high the AXI master starts generating cycles.

output done
Goes high if all cycles have been generated (All FIFOs are empty)

Generate cycles.

For those of you who can't wait to read it all look at the example usage at the end. It has been taken from the first usage of the test bench.

Write address cycle

To generate a write address on the AXI bus the user must call the ***q_wadrs*** task. For each call a write address is queued up to be output at a later stage. This call does NOT do anything about the write data. It is the users responsibility to queue up the correct amount of write data. At least one write data value must be queued for each address. But if the user generates write-bursts more data must be queued.

The q_wadrs task takes the following input arguments:
input integer address;
The address to write to.

input integer length;
The burst length. This is NOT the AXI value but the actual length of the burst. Thus with four length bits you can use the values 1..16. The value will be translated into the AXI value (by subtracting 1).

input integer size;
The size of the data. The value can be the AXI size (See table below) but it also accepts the real values of 8,16,32..1024. Any other value will give an error message and the test bench will $stop. The Following table shows the possible size values:

Code	**Actual data size**
0	8
1	16
2	32
3	64
4	128
5	256
6	512
7	1024
8	8
16	16
32	32
64	64
128	128
256	256
512	512
1024	1024

input integer id;

The is the ID value output during the cycle.

input integer burst;
The is the burst type value output during the cycle.

input integer delay_random;
This is a random “true/false” flag for the delay. A value of 0 means the delay is used as specified. A value of non-zero (e.g. 1) means the delay value is used as 'modulo' argument for a random number. See also below.

input integer delay;
The delay specifies the number of clock cycles which the test bench waits before the data is put on the bus (The bus is 'idle' during those cycles). The delay depends on the delay_random parameter above. The behaviour is simplest explained using the actual code:

```
if (delay_random)
   actual_delay = $random % delay;
else
   actual_delay = delay;
```

Thus if delay_random is zero the delay is used as the number of clock cycles to wait.
If delay_random is non-zero the delay is used as to limit the maximum random number generated.

Write data cycle

To generate a write address on the AXI bus the user must call the ***q_wdata*** task. For each call a write data units is queued up to be output at a later stage. This call is normally used to generate data bursts.

The ***q_wdata*** task takes the following input arguments:
input integer data;
The data to write.

input integer strobes;
The write strobes.

input integer last;
This value must be 1 if this is the last beat of a data burst. It must be zero for all other data beats.

input integer id;
The is the ID value output during the cycle.

input integer delay_random;
This is a random “true/false” flag for the delay. A value of 0 means the delay is used as specified. A value of non-zero (e.g. 1) means the delay value is used as 'modulo' argument for a random number. See also below.

input integer delay;
The delay specifies the number of clock cycles which the test bench waits before the data is put on the bus (The bus is 'idle' during those cycles). The delay depends on the delay_random parameter above. The behaviour is simplest explained using the actual code:

```
if (delay_random)
   actual_delay = $random % delay;
else
```

```
   actual_delay = delay;
```

Thus if delay_random is zero the delay is used as the number of clock cycles to wait.
If delay_random is non-zero the delay is used as to limit the maximum random number generated.

Combined write address & data cycle

This call queues up a combined write address and write data. This is not equivalent with calling ***q_wadrs*** and ***q_wdata*** separately!
The major difference is that this call synchronises the delay of the two cycles. Thus it allows the user to specify a write address and write data to appear on the bus with exact interlocked timing.
If both are set to the same delay the write address and write data will be put on the bus in the same clock cycle. To offset them you can use different delays.

Address delay	Data delay	Result output
N	N	Both in the same clock cycle
N	N+m	Data m cycles after address
N+m	N	Data m cycles before address

The task is called ***q_wsync*** and has the following arguments:

Address arguments:

input integer address;
input integer length;
input integer size;
input integer id;
input integer burst;
input integer adelay_random;
input integer adelay;

Data arguments:

input integer data;
input integer strobes;
input integer last;
input integer wdelay_random;
input integer wdelay;

For an explanation of the arguments see the previous two chapters. The only difference is that there is now only one ID argument as the cycles use the same ID.

Read address cycle

The read address cycle is almost equivalent with the write address cycle. To generate a read address on the AXI bus the user must call the ***q_radrs*** task. For each call a read address is queued up to be output at a later stage. This call does NOT do anything about the read data. If no read data is queued up all read data is accepted and discarded.

The ***q_radrs*** task takes exactly the same arguments as the ***q_wadrs*** task. For completeness they are repeated here.
input integer address;
The address to read from.

input integer length;
The burst length. This is NOT the AXI value but the actual length of the burst. Thus with four length bits you can use the values 1..16. The value will be translated into the AXI value (by subtracting 1).

input integer size;
The size of the data. The value can be the AXI size (See table below) but it also accepts the real values of 8,16,32..1024. Any other value will give an error message and the test bench will $stop.
The Following table shows the possible size values:

Code	Actual data size
0	8
1	16
2	32
3	64
4	128
5	256
6	512
7	1024
8	8
16	16
32	32
64	64
128	128
256	256
512	512
1024	1024

input integer id;
The is the ID value output during the cycle.

input integer burst;
The is the burst type value output during the cycle.

input integer delay_random;
This is a random “true/false” flag for the delay. A value of 0 means the delay is used as specified. A value of non-zero (e.g. 1) means the delay value is used as 'modulo' argument for a random number. See also below.

input integer delay;
The delay specifies the number of clock cycles which the test bench waits before the data is put on the bus (The bus is 'idle' during those cycles). The delay depends on the delay_random parameter above. The behaviour is simplest explained using the actual code:

```
if (delay_random)
   actual_delay = $random % delay;
else
   actual_delay = delay;
```

Thus if delay_random is zero the delay is used as the number of clock cycles to wait.

If delay_random is non-zero the delay is used as to limit the maximum random number generated.

Read response cycle

It is possible to queue up read-response cycles. These do not generate AXI bus cycles but they can check read data arriving back at the master. Thus if you know for a read address what you should get back, you can load that into the test bench and it will perform the data check for you.

For the read data it is possible to use the "x' value alongside the 1/0 values. If a bit is specified as 'x' it means the bit will not be tested. For example if you read a 32-bit value but the top 16 bits are not important, you can specify a read value of 32'hxxxx1234.

input integer data;
input integer last;
input integer id;
input integer delay_random;
input integer delay;
input integer delay_mode;

The ***q_rdata*** task takes the following input arguments:
input integer data;
The data expected to come back. It is compared against the actual data being read. If some bits do not need to be checked specify them as 'x' (Don't acre).

input integer last;
This argument is not yet used. In future it will check the 'last' bit against the received bit.

input integer id;
This argument is not yet used. In future it will check the the ID value against the received ID value.

input integer delay_random;
This is a random “true/false” flag for the delay. A value of 0 means the delay is used as specified. A value of non-zero (e.g. 1) means the delay value is used as 'modulo' argument for a random number. See also below.

input integer delay;
The delay specifies the number of clock cycles which the test bench waits before it presents the ***rready*** signal. The delay depends on the delay_random parameter above. The behaviour is simplest explained using the actual code:

```
if (delay_random)
   actual_delay = $random % delay;
else
   actual_delay = delay;
```

Thus if delay_random is zero the delay is used as the number of clock cycles to wait.
If delay_random is non-zero the delay is used as to limit the maximum random number generated.

input integer delay_mode;
This argument is not yet used. It is intended to switch the read response delay between pre-cycle or post cycle.

Parameters.

The AXI test master has the following parameters:

ADRS_BITS
The number of address bits in both the read and write address.

DATA_BITS
The number of data bits in both the read and write data. Values above 32 are not yet supported.

LEN_BITS
The number of (burst) length bits

ID_BITS
The number of ID bits.

The following parameters have nothing to do with the AXI bus. They specify the number of entries which can be queue up.

AQ_DEPTH
Log-2 of the address queue depth for both the read addresses and write addresses. Thus a value of 10 means you can queue up 1024 addresses. The default is 10 (1K).

DQ_DEPTH
Log-2 of the data queue depth for both the read data and write data. Thus a value of 10 means you can queue up 1024 data units. The default is 16 (64K).

Limitations.

The test bench is till under development. In fact during the testing of AXI components I have come across more bugs in the test master then in my AXI component. I think by now most bugs have been eliminated.

Also to balance the time spend on test bench versus the actual code being tested not all possible features have been implemented. The following are know limitations:

Data width.
For now the test bench has been written for 32 bit data. To change this adapt the queue data task input from integer to a vector of the correct size.

Response channels.
All read-response and write response code values are ignored. Also the response channels are always ready.

Queue time
It takes one clock cycle to queue up a value. To speed up loading multiple master the fork-join construct can be used. (See example code)

RTL
One artificial construct is used to put data in the FIFOs. Apart from that I think the test bench can be

converted to real gates.

```
   @(negedge clk)
      awff_write <= 1'b1;
   @(posedge clk)
      #1 awff_write <= 1'b0;
```

user and/or other signals

There are no user or any other signals. These easily be added as the format for all signals, except ***ready*** and ***valid***, are the same.

Example usage.

The following is a code snippet from a test bench using three AXI masters.

```
      // Each master uses 1/3 of the memory 4096/3 = 0x555
      // As I uses bursts of up to 4 units
      // I have make sure I don't go over the boundary
      // into the next area so modulo a bit smaller

      // Go through all combinations of delay 0-3 and burst 1-3
      // Not yet tested is write data vs write address delay offsets

      for (b0=1; b0<4; b0=b0+1)
      begin // master 0 burst 1..3
         for (b1=1; b1<4; b1=b1+1)
         begin
            for (b2=1; b2<4; b2=b2+1)
            begin

               for (d0=0; d0<4; d0=d0+1)
               begin // master 0 delay 0..3
                  for (d1=0; d1<4; d1=d1+1)
                  begin
                     for (d2=0; d2<4; d2=d2+1)
                     begin

   fork
      // Write to queues in parallel
      begin
         v0 = 32'h0000+($random % 32'h0550);
         axi_test_master0.q_wsync(
                v0<<2, // address;
                   b0, // length;
                   32, // size;
                    0, // id;
                    1, // burst;
                    0, // delay_random;
                   d0, // delay;
                   v0, // data;
                 4'hF, // strobes;
            b0==1?1:0, // last;
                    0, // delay_random;
                   d0  // delay;
                   );
         v0 = v0 + 1;
         for (r0=1; r0<b0;r0=r0+1)
         begin
            axi_test_master0.q_wdata(
               v0, 4'hF,
               r0==b0-1?1:0,0,0,2
               );
            v0 = v0 + 1;
         end
      end
      begin
         v1 = 32'h0555+($random % 32'h0550);
         axi_test_master1.q_wsync(
                 v1<<2, // address;
                    b1, // length;
                    32, // size;
                     1, // id;
                     1, // burst;
                     0, // delay_random;
                    d1, // delay;
                    v1, // data;
                  4'hF, // strobes;
             b1==1?1:0, // last;
                     0, // delay_random;
                    d1  // delay;
                    );
         v1 = v1 + 1;
         for (r1=1; r1<b1;r1=r1+1)
         begin
            axi_test_master1.q_wdata(v1, 4'hF,
               r1==b1-1?1:0,1,0,2
               );
            v1 = v1 + 1;
         end
      end
      begin
         v2 = 32'h0AAA+($random % 32'h0550);
         axi_test_master2.q_wsync(
                v2<<2, // address;
                   b2, // length;
                   32, // size;
                    2, // id;
                    1, // burst;
                    0, // delay_random;
                   d2, // delay;
                   v2, // data;
                 4'hF, // strobes;
            b2==1?1:0, // last;
                    0, // delay_random;
                   d2  // delay;
                   );
         v2 = v2 + 1;
         for (r2=1; r2<b2;r2=r2+1)
         begin
            axi_test_master2.q_wdata(v2, 4'hF,
               r2==b2-1?1:0,2,0,2
               );
             v2 = v2 + 1;
          end
      end
   join
                    end
                 end
              end

            end
            // Can not queue all tests so:
            run_and_check_task;
         end
      end

      // ---------------------------
      //
      // read data test
      // Self checking
      //
      // --------------------------


      // Loop through all combinations of
      // burst =1..4 and delay = 0..3
      // Read address is random
      for (b0=1; b0<4; b0=b0+1)
   begin
      for (b1=1; b1<4; b1=b1+1)
      begin
          for (b2=1; b2<4; b2=b2+1)
         begin

            for (d0=0; d0<4; d0=d0+1)
            begin
               for (d1=0; d1<4; d1=d1+1)
               begin
                  for (d2=0; d2<4; d2=d2+1)
                  begin

fork
   // Write to queues in parallel
   begin  // join task 0
      v0 = 32'h0000 + ($random % 32'h0550);
      // Queue up a read (address & expected data)
      // (synced reads do not exist)
      axi_test_master0.q_ardata(
             v0<<2, // address;
                b0, // length;
                32, // size;
                 0, // id;
                 1, // bursttype;
                 0, // delay_random;
                d0, // delay;
                v0, // rdata;
         b0==1?1:0, // rlast;
                 0, // rdelay_random;
                d0, // rdelay;
                 0  // rdelay mode
                );
      // Next value
      v0 = v0 + 1;
      // If we have a burst (b0>1) queue more expected data
      for (r0=1; r0<b0;r0=r0+1)
      begin
         // args= data,last,id,delay_random,delay,delay_mode
         axi_test_master0.q_rdata(v0,
            r0==b0-1?1:0,0,0,d0,0
            );
         v0 = v0 + 1;
      end
   end
   begin  // join task 1
      v1 = 32'h0555 + ($random % 32'h0550);
      axi_test_master1.q_ardata(
             v1<<2, // address;
                b1, // length;
                32, // size;
                 1, // id;
                 1, // bursttype;
                 0, // delay_random;
                d1, // delay;
                v1, // rdata;
         b1==1?1:0, // rlast;
                 0, // rdelay_random;
                d1, // rdelay;
                 0  // rdelay_mode;
                );
      v1 = v1 + 1;
      for (r1=1; r1<b1;r1=r1+1)
      begin
         // args= data,last,id,delay_random,delay,delay_mode
         axi_test_master1.q_rdata(v1,
            r1==b1-1?1:0,1,0,d1,0
            );
         v1 = v1 + 1;
      end
   end
   begin  // join task 2
      v2 = 32'h0AAA + ($random % 32'h0550);
      axi_test_master2.q_ardata(
             v2<<2, // address;
                b2, // length;
                32, // size;
                 2, // id;
                 1, // bursttype;
                 0, // delay_random;
                d2, // delay;
                v2, // rdata;
         b2==1?1:0, // rlast;
                 0, // rdelay_random;
                d2, // rdelay;
                 0  // rdelay_mode;
               );
      v2 = v2 + 1;
      for (r2=1; r2<b2;r2=r2+1)
      begin
         // args= data,last,id,delay_random,delay,delay_mode
         axi_test_master2.q_rdata(v2,
             r2==b2-1?1:0,2,0,d2,0
             );
         v2 = v2 + 1;
      end
   end // join task 2
join

                  end // for d2
               end // for d1
            end // for d0

         end // for b2

         // Run now to prevent axi master queues overflow
         # (CLK_PERIOD*5);
         run = 1'b1;
         # (CLK_PERIOD*5);
         wait (done0 & done1 & done2);
         # (CLK_PERIOD*5);
         run = 1'b0;
         # (CLK_PERIOD*50);


      end // for b1
   end // for b0

   //
   // Check round robin is working
   //
   // Ful request, no delays all channels
   // Should read from 0,1,2
   // starting with whom ever was last

   v0 = 32'h0000;
   v1 = 32'h0555;
   v2 = 32'h0AAA;
   // Bursts of 3
   b0=3;
   b1=3;
   b2=3;

   for (q=0; q<100; q=q+1)
   begin

      fork
         // Write to queues in parallel
         begin  // join task 0
            v0 = v0 + 1;
            // Queue up a read (address & expected data)
            // (synced reads do not exist)
      axi_test_master0.q_ardata(
             v0<<2, // address;
                b0, // length;
                32, // size;
                 0, // id;
                 1, // bursttype;
                 0, // delay_random;
                 0, // delay;
                v0, // rdata;
         b0==1?1:0, // rlast;
                 0, // rdelay_random;
                 0, // rdelay;
                 0  // rdelay mode
                );
      // Next value
      v0 = v0 + 1;
      // If we have a burst (b0>1) queue more expected data

      for (r0=1; r0<b0;r0=r0+1)
      begin
         // args= data,last,id,delay_random,delay,delay_mode
         axi_test_master0.q_rdata(v0,
            r0==b0-1?1:0,0,0,0,0
            );
         v0 = v0 + 1;
      end
   end
   begin  // join task 1
      v1 = v1 + 1;
      axi_test_master1.q_ardata(
             v1<<2, // address;
                b1, // length;
                32, // size;
                 1, // id;
                 1, // bursttype;
                 0, // delay_random;
                 0, // delay;
                v1, // rdata;
         b1==1?1:0, // rlast;
                 0, // rdelay_random;
                 0, // rdelay;
                 0  // rdelay_mode;
                );
      v1 = v1 + 1;
      for (r1=1; r1<b1;r1=r1+1)
      begin
         // args= data,last,id,delay_random,delay,delay_mode
         axi_test_master1.q_rdata(v1,
            r1==b1-1?1:0,1,0,0,0
            );
         v1 = v1 + 1;
      end
   end
   begin  // join task 2
      v2 = v2 + 1;
      axi_test_master2.q_ardata(
             v2<<2, // address;
                b2, // length;

       32, // size;
                 2, // id;
                 1, // bursttype;
                 0, // delay_random;
                 0, // delay;
                v2, // rdata;
         b2==1?1:0, // rlast;
                 0, // rdelay_random;
                 0, // rdelay;
                 0  // rdelay_mode;
               );
      v2 = v2 + 1;
```

```
            for (r2=1; r2<b2;r2=r2+1)
            begin
               // args= data,last,id,delay_random,delay,delay_mode
               axi_test_master2.q_rdata(v2,
                   r2==b2-1?1:0,2,0,0,0
                   );
               v2 = v2 + 1;
            end
         end // join task 2
      join
   end // for q


   # (CLK_PERIOD*5);
   run = 1'b1;

   //
   // Determine input port processing order
   //

   // Pick up the first one
   wait( (s0_arvalid & s0_arready) |
         (s1_arvalid & s1_arready) |
         (s2_arvalid & s2_arready) ) ;
   if (s0_arvalid & s0_arready) port=0;
   if (s1_arvalid & s1_arready) port=1;
   if (s2_arvalid & s2_arready) port=2;
   @(posedge clk) ;

   // check round robin operation
   for (q=1; q<100; q=q+1)
   begin
      @(posedge clk)
      if (s0_arready | s1_arready | s2_arready)
      begin
         exp_port = (port + 1) % 3;
         if (s0_arvalid & s0_arready) port=0;
         if (s1_arvalid & s1_arready) port=1;
         if (s2_arvalid & s2_arready) port=2;
         if ( exp_port != port )
         begin
            $display("%m @%0t: swtich priority error, Have %d exepected %d",
                      $time,port,exp_port);
            #1 $stop;
         end

      end
   end

   # (CLK_PERIOD*5);
   wait (done0 & done1 & done2);
   # (CLK_PERIOD*5);
   run = 1'b0;
   # (CLK_PERIOD*50);

   v0 = 32'h0000;
   v1 = 32'h0555;
   v2 = 32'h0AAA;

   # (CLK_PERIOD*50)
   $display("************************");
   $display("**                    **");
   $display("**    Test Passed     **");
   $display("**                    **");
   $display("************************");
   $stop;
```

end