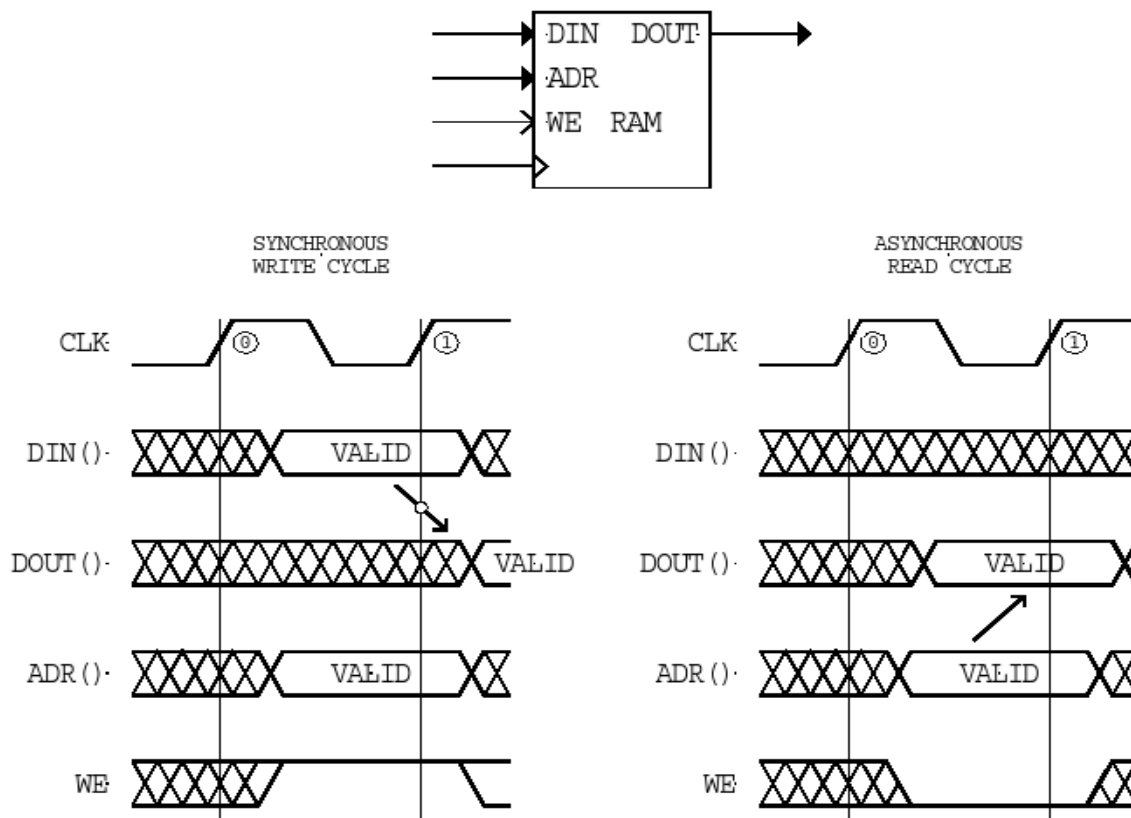


# PCI to WB transactions

We didn't examine in detail the architecture of PCI\_DEV\_1/2 modules since these devices won't be part of the synthesized core. On the contrary, SIMPLE\_WB\_SLAVE is one of the possible modules that we can add to the WB bus in order to create a single modular chip interfaced to the PCI bus through the bridge; therefore we're going to explain the basic principles of its interface and its implementation.

In our example, SIMPLE\_WB\_SLAVE is a simple RAM; if we choose to implement this memory module with the following interface (which is very similar to the WB interface) and timing diagrams, then it will work more efficiently on the WB bus, as stated by WB\_b3 specifications:



The above memory, called "FASM" by the WB\_b3 specifications, can be realized with the following simple verilog module (which you can find in the testbench). The only difference with the above timing diagrams is that our FASM, during read cycles, synchronously presents data after the rising edge of the clock input.

```
file: FASM.v
timescale 1 ps / 1 ps
define MEM_SIZE 4096*2

module fasm(CLK,WE,ADR,DI,DO);
    input CLK;
    input WE;
    input [7:0] ADR;
    input [31:0] DI;
    output reg [31:0] DO;
    reg [7:0] ram[0:MEM_SIZE - 1]; // memory cells
    integer i;
```

```
/* The following registers are only for debugging purposes on gtkWave. */
```

```
wire[7:0] debug_ram_byte_0 = ram[0];
```

```
wire[7:0] debug_ram_byte_1 = ram[1];
```

```
wire[7:0] debug_ram_byte_2 = ram[2];
```

```
wire[7:0] debug_ram_byte_3 = ram[3];
```

```
wire[7:0] debug_ram_byte_4 = ram[4];
```

```
wire[7:0] debug_ram_byte_5 = ram[5];
```

```
wire[7:0] debug_ram_byte_6 = ram[6];
```

```
wire[7:0] debug_ram_byte_7 = ram[7];
```

```
initial //initialize all RAM cells to 0 at startup
```

```
begin
```

```
    for (i=0; i < `MEM_SIZE; i = i + 1)
```

```
        ram[i] = 0;
```

```
end
```

```
//READ
```

```
always @(ADR)
```

```
begin
```

```
    DO[7:0] <= #1000 ram[ADR + 3];
```

```
    DO[15:8] <= #1000 ram[ADR + 2];
```

```
    DO[23:16] <= #1000 ram[ADR + 1];
```

```
    DO[31:24] <= #1000 ram[ADR];
```

```
end
```

```
//WRITE
```

```
always @(posedge CLK)
```

```
begin
```

```
    if (WE == 1'b1)
```

```
    begin
```

```
        ram[ADR + 3] <= #1000 DI[7:0];
```

```
        ram[ADR + 2] <= #1000 DI[15:8];
```

```
        ram[ADR + 1] <= #1000 DI[23:16];
```

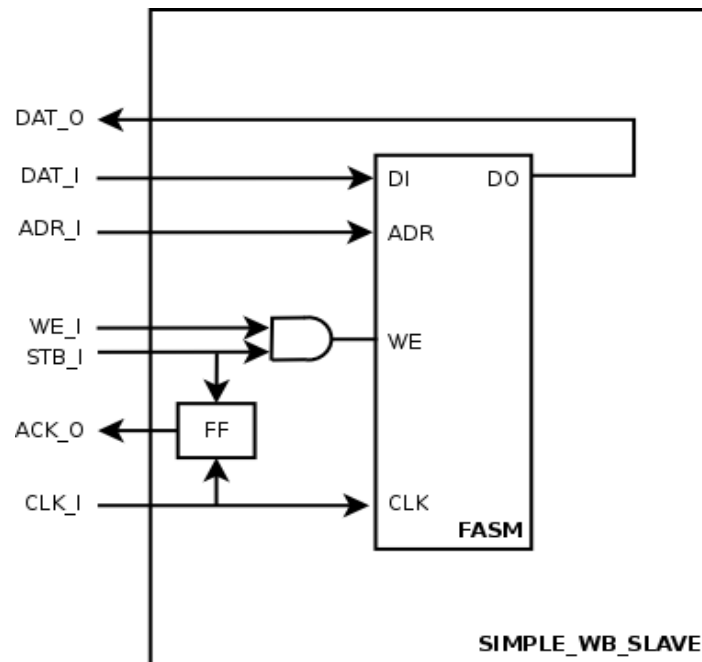
```
        ram[ADR] <= #1000 DI[31:24];
```

```
    end
```

```
end
```

```
endmodule
```

At this point, SIMPLE\_WB\_SLAVE can be implemented, using the previous FASM module, similarly to figure A-16 of the WB\_b3 specification doc, in the following way:



```

/***** file: SIMPLE_WB_SLAVE.v *****/

```

```

#include "pci_testbench_defines.v"
#include "pci_constants.v"
#include "FASM.v"

```

```

module WB_SIMPLE_WB_SLAVE

```

```

(
    CLK_I,
    RST_I,
    ACK_O,
    ADR_I,
    CYC_I,
    DAT_O,
    DAT_I,
    ERR_O,
    RTY_O,
    SEL_I,
    STB_I,
    WE_I,
    CTI_I,
    BTE_I

```

```

);

```

```

input          CLK_I;
input          RST_I;
output         ACK_O;
input `WB_ADDR_TYPE  ADR_I;
input          CYC_I;
output `WB_DATA_TYPE  DAT_O;
input `WB_DATA_TYPE  DAT_I;
output         ERR_O;
output         RTY_O;
input `WB_SEL_TYPE  SEL_I;
input          STB_I;

```

```

input          WE_I;
input [2:0]    CTI_I;
input [1:0]    BTE_I;

reg calc_ack;
reg one_wait_state;
assign ACK_O = calc_ack && STB_I && CYC_I;
assign RTY_O = 0;
assign ERR_O = 0;

fasm ram_instance
(
    .DO    (DAT_O),
    .DI    (DAT_I),
    .ADR    (ADR_I[7:0]),
    .CLK    (CLK_I),
    //.EN    (1'b1), //not implemented in our test fasm
    //.RST    (1'b0), //not implemented in our test fasm
    .WE    (STB_I && WE_I)
);

always@(posedge CLK_I && one_wait_state === 1'b1)
begin
    if(STB_I)
        calc_ack <= #1000 ~calc_ack;
end

task set_one_wait_state;
    input val;
begin
    calc_ack <= ~val;
    one_wait_state <= val;
end
endtask

initial
begin
    calc_ack <= 1;
    one_wait_state <= 0;
end

endmodule

```

Note that some signals, inherited from the “official” testbench, have been included - but not used - in the module.

An additional flip flop and gate has been added in the ACK\_O circuit, as indicated in paragraph A.7.3 of WB\_b3 specs, since our FASM performs synchronous read cycles; this flip flop introduces a necessary wait cycle for each read transaction (therefore, the resulting read transactions on the WB bus are slower than write transactions)

The wait cycle can be set/unset by calling the task set\_one\_wait\_state (look at the code: it's self-explanatory).

Now we have to connect WB\_SLAVE\_UNIT to PCI\_TARGET\_UNIT: this can be easily obtained using a point to point interconnection (remember that the associated WB bus is shared only by one master and one slave module):

```

/***** file: SYSTEM.v *****/
.

```

```

.
.
SIMPLE_WB_SLAVE wishbone_slave
(
    .CLK_I      (wb_clock),
    .RST_I(reset_wb),
    .ACK_O      (ACK_I),
    .ADR_I      (ADR_O),
    .CYC_I      (CYC_O),
    .DAT_O      (MDAT_I),
    .DAT_I      (MDAT_O),
    .ERR_O      (ERR_I),
    .RTY_O      (RTY_I),
    .SEL_I(SEL_O),
    .STB_I(STB_O),
    .WE_I (WE_O),
    .CTI_I (CTI_O),
    .BTE_I      (BTE_O)
);

```

As we have just seen, it's very simple to interface a memory model to `PCI_TARGET_UNIT`: and with little knowledge of the WB protocol and little verilog additional code, we have implemented a RAM core ready to be accessed R/W through the PCI bus.

Now let's perform read/write cycles from `PCI_DEV_1` to our FASM;

In order to do that, it's necessary to configure the `PCI_AM_1` register, through which we can specify how large is the region pointed by the bridge's base address 1. The FASM module has a size of 8K; if we set `PCI_AM_1` to the value `32'hFFFF0000`, then the region will have a size of 8K: as reported in `$PCI/doc/pci_specification.pdf` (see page 30), each bit in the `PCI_AM_1` register corresponds to one address line.

The above register's address is calculated by adding an offset of 118 (hex) to the bridge's base address 0 (see `pci_constants.v`); we will perform a memory write PCI transaction with the following task, which we won't examine in detail, since it will be replaced by a system call (`write()`) of the operating system (Linux) that has to communicate with the synthesized core through the PCI bus; anyway, note that `BE`, according to PCI specification, indicates which AD's byte lanes carry meaningful data (for example: `BE=4'b1100` indicates that meaningful data are on `AD[8:15]` and `AD[0:7]`, while `BE=4'b0110` requires meaningful data on `AD[24:31]` and `AD[0:7]`)

```

/***** file: SYSTEM.v *****/
.
.
.
$display(" ");
$display(" Setting P_AM_1");
//SIMPLE_PCI_MEM_WRITE has the following arguments: master_number, address, data, byte_en, size
SIMPLE_PCI_MEM_WRITE (1, `TAR0_BASE_ADDR_0+32'h118, 32'hFFFF0000, 4'b0011, 1);

```

In addition:

- a) we enable ERROR reporting through the `ERR_EN` bit of PCI Error Control and Status register: `P_ERR_CS` (offset: 160 (hex) )
- b) given that our FASM's base address is 0, in some way the bridge must “translate” to `32'h00000000` all the requests which are addressed to ``TAR0_BASE_ADDR_1`: we can achieve that by enabling Address translation (through `AT_EN`, which is bit 2 of `P_IMG_CTRL0`) and setting

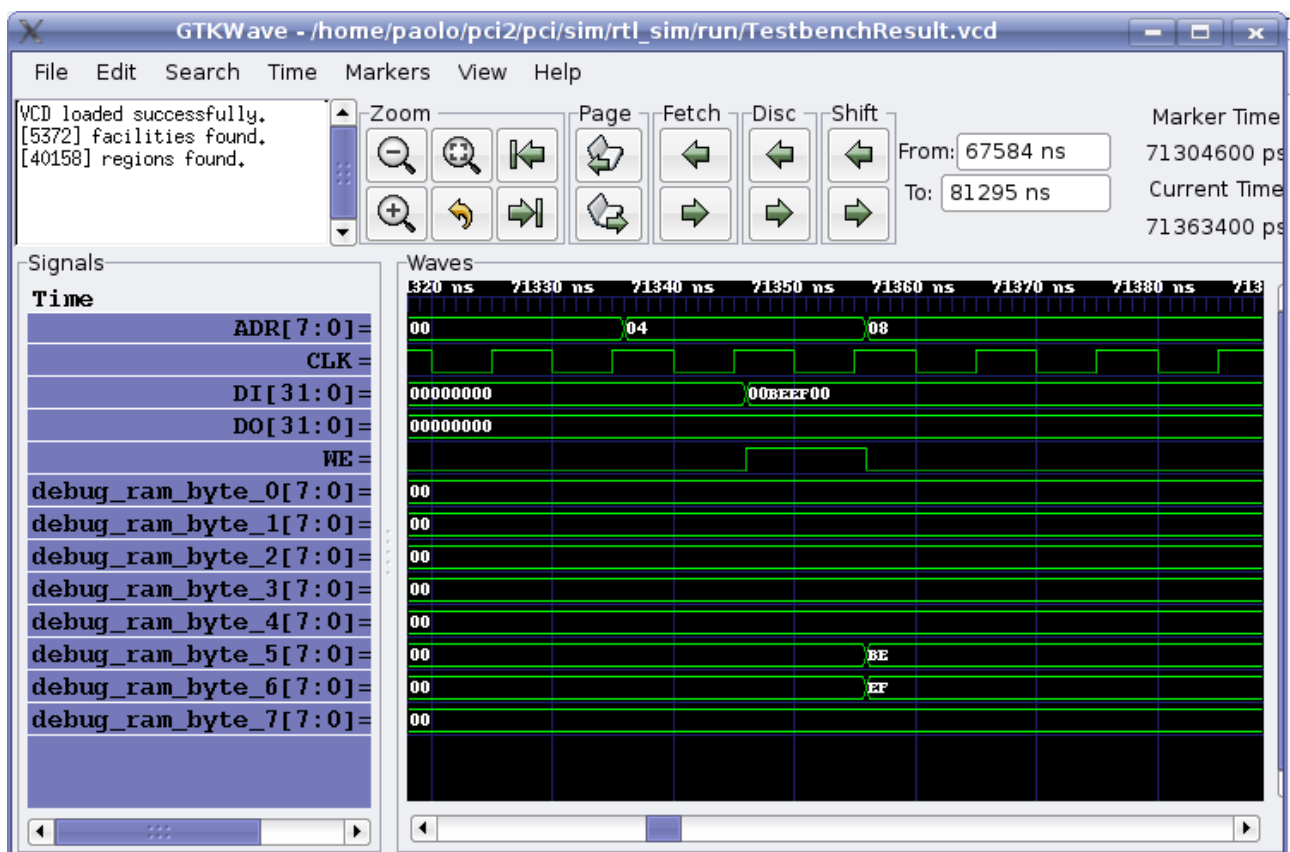
TAR0\_TRAN\_ADDR\_1 register (offset 11c (h) ) to 32'h00000000

```
file: SYSTEM.v
.
.
$display(" Setting ERR_EN bit of P_ERR_CS");
SIMPLE_PCI_MEM_WRITE (1,TAR0_BASE_ADDR_0 + 12'h160, 32'h00000001, 4'b1110,1);
$display(" Enabling Address translation (bit 2 of P_IMG_CTRL0)");
SIMPLE_PCI_MEM_WRITE (1,TAR0_BASE_ADDR_0 + 12'h110, 32'h00000004, 4'b1110,1);
$display(" Setting TAR0_TRAN_ADDR_1 to 32'h00000000");
SIMPLE_PCI_MEM_WRITE (1,TAR0_BASE_ADDR_0 + 12'h11c, 32'h00000000, 4'b0000,1);
```

At this point, PCI\_DEV\_1 can write a DWORD to the address 4'h4 of the memory, which is seen on the PCI bus as the offset 4 of `TAR0\_BASE\_ADDR\_1 :

```
file: SYSTEM.v
.
.
do_pause(50); //Wait 50 clk cycles
//PCI_DEV_1 writes 1 D-WORD to WB_SLAVE
$display(" Writing 32'h00BEEF00 to adr 4'h4 of our FASM");
SIMPLE_PCI_MEM_WRITE (1,TAR0_BASE_ADDR_1 + 3'h4, 32'h00BEEF00, 4'b1001,1);
```

We can verify, by examining TestbenchResult.vcd with GTKWave, that the previous value ( 32'h00BEEF00 ) is now stored in the right FASM's locations (see debug\_ram\_byte\_5/6 signals):



Now, we want to write 6 DWORDs starting from address 4'h8 of our FASM. If we make the region pointed by `TAR0\_BASE\_ADDR\_1 prefetchable (see on PCI specs - as well on Linux Device Drivers , third edition, page 316 - what “prefetchable” means) , by setting the PREF\_EN bit of P\_IMG\_CTRL0, then any BE can be used during a burst write:

```

/***** file: SYSTEM.v *****/
.
.
.
//PCI_DEV_1 writes 6 DWORDs to WB_SLAVE
do_pause(50); //Wait 50 clk cycles
$display(" Setting PREF_EN (bit 1 of P_IMG_CTRL0)");
//note that we don't clear the previously set AT_EN
SIMPLE_PCI_MEM_WRITE (1,`TAR0_BASE_ADDR_0 + 9'h110, 32'h00000006, 4'b1110,1);
$display(" Writing 6 DWORDS from adr 4'h8 of our FASM");
SIMPLE_PCI_MEM_WRITE (1,`TAR0_BASE_ADDR_1 + 4'h8, 32'hAAAAAAAA, ({ $random } % 4), 6);

```

*N.B) while performing a burst write of N DWORDs, the task SIMPLE\_PCI\_MEM\_WRITE calls another task inherited from the old testbench, which can be found in pci\_behaviorial\_master.v: this function starts from an initial value and obtains the remaining n-1 values by incrementing each byte of each DWORD by 8'h1; if, for example, the starting DWORD value is AAAAAAAAAA, then the following values will be ABABABAB, ACACACAC, ADADADAD, AEAEAEAE, AFAFAFAF... (see task Update\_Write\_Data of pci\_behaviorial\_master.v)*

Below is an explanation for the principal steps involved during a burst write of N DWORDs, to a chosen address, started from a PCI initiator:

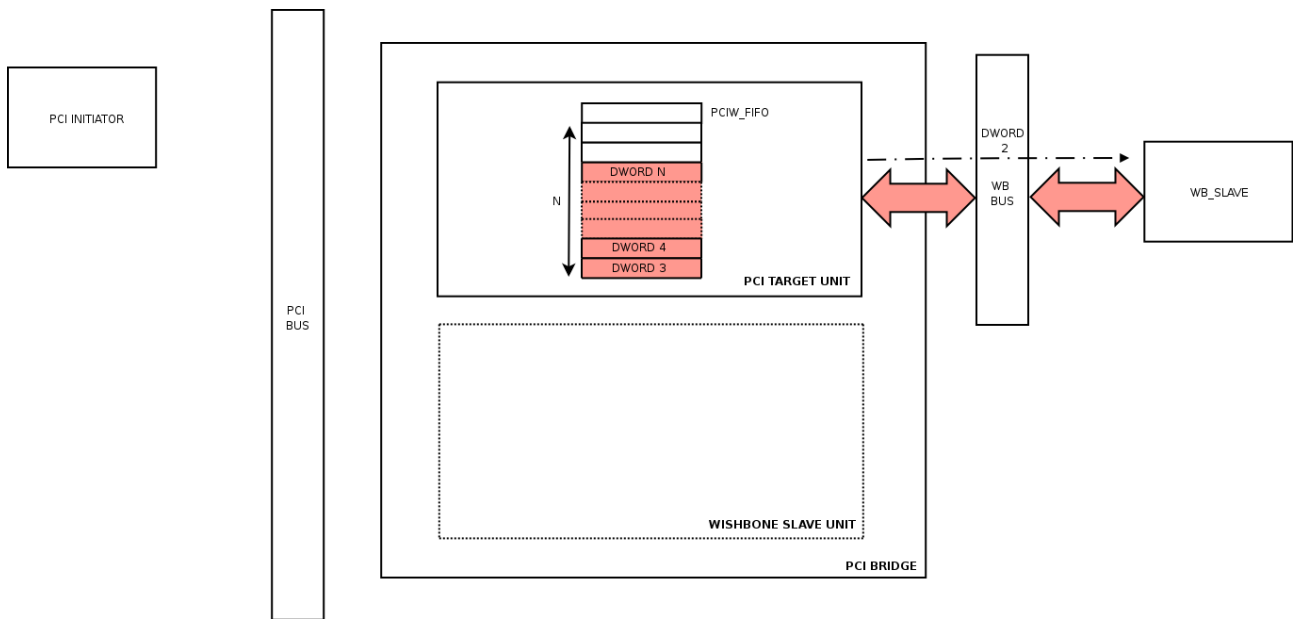
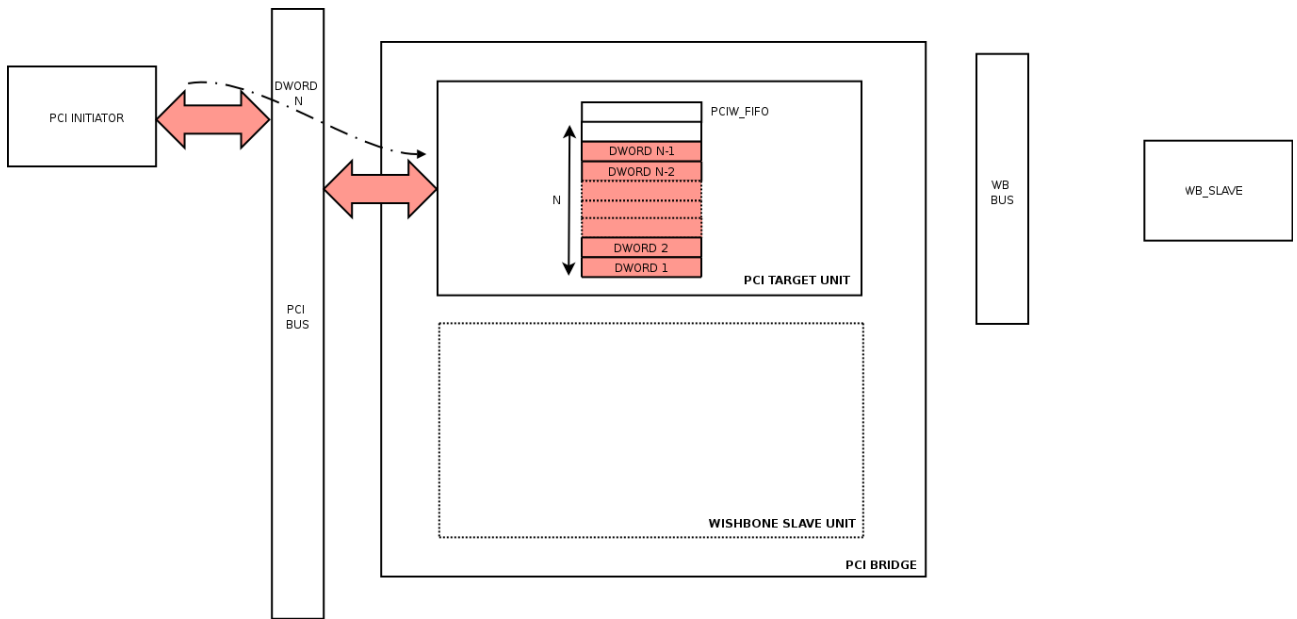
- 1) The PCI initiator starts a PCI write transaction, writing the DWORDs on the PCI bus
- 2) PCI TARGET UNIT buffers in PCIW\_FIFO - a FIFO with a depth defined by `PCIW\_DEPTH (see pci\_constants.v) – the previous DWORDs and, depending on the actual queue's state and the presence/absence of errors during the operation:
  - 2.a) ends the transaction
  - 2.b) stops the transaction with a Target Abort or Disconnect signal (see pci specs: the target signals a Disconnect by asserting #TRDY and #STOP together, and an Abort by deasserting #DEVSEL and asserting #STOP)
  - 2.c) retries the transaction with a retry signal (it asserts #STOP and doesn't assert #TRDY)
- 3) If the PCI transaction is terminated, the bridge executes N write cycles on the WB bus, by flushing N DWORDs from the fifo, starting from the chosen address and incrementing it by one DWORD at each ACK\_O signal got from WB slave.

*N.B: in order to have a linear incrementing mode for the chosen address, AD[1:0] must be equal to 2'b00 during the address phase. During this phase, AD[1:0] indicates the order in which the master is requesting the data to be transferred (see PCI specs, paragraph 3.2.2.2); another mode for the burst ordering is the “Cache-line Wrap mode” (AD[1:0] == 2'b10), which is not fully supported by the bridge.*

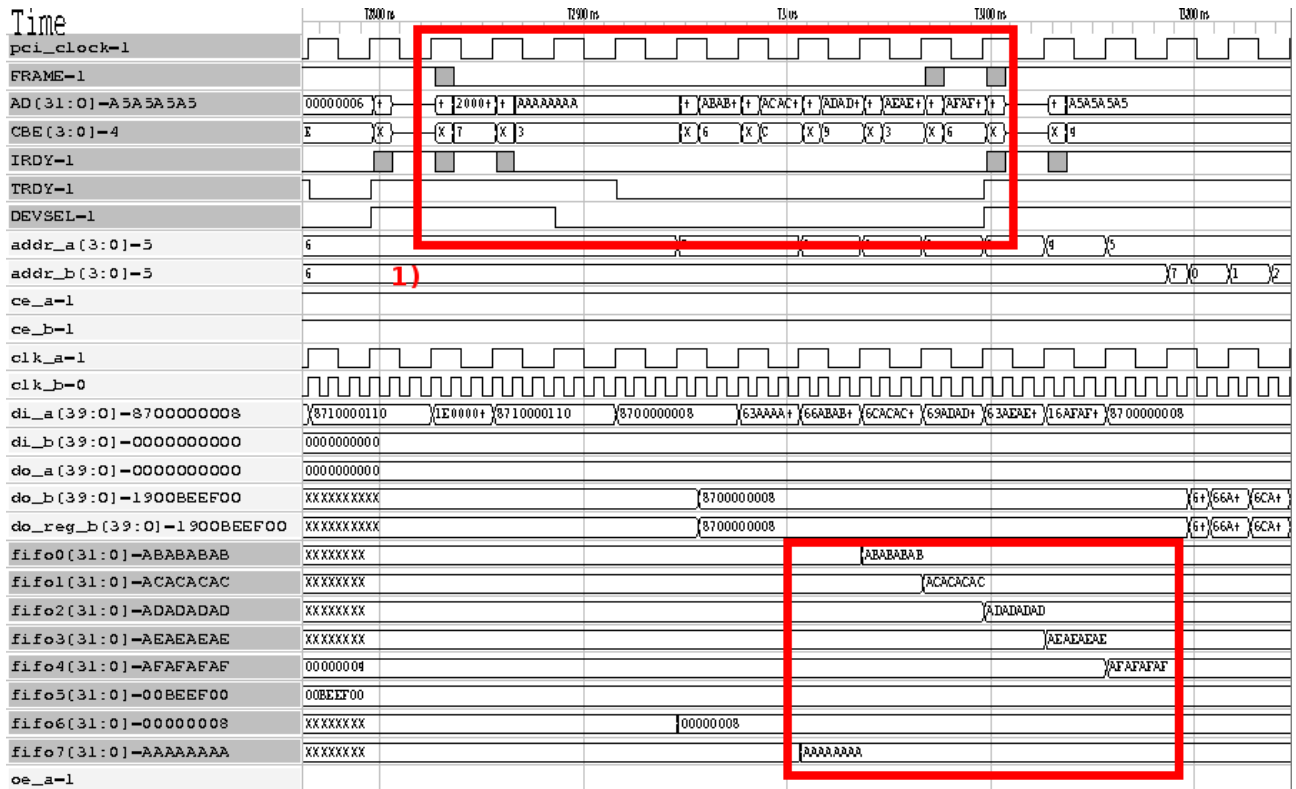
All the previous steps are explained in detail in \$PCI/doc/pci\_specification.pdf (paragraph 3.4.3); anyway, let's examine closer what happens during some cases of a PCI to WB write cycle:

A) 1, 2.a, 3

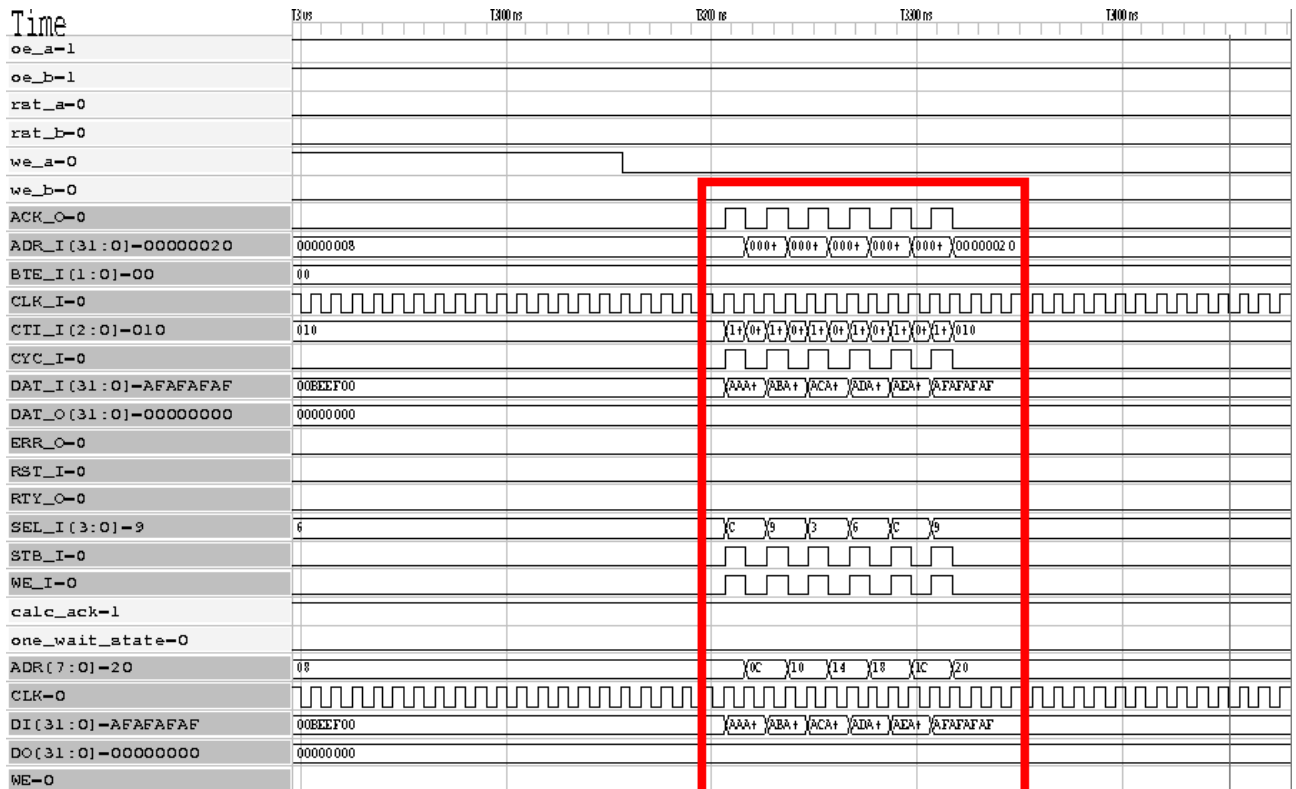
\*\*\*\* 1, 2a) \*\*\*\*







2a)



3)

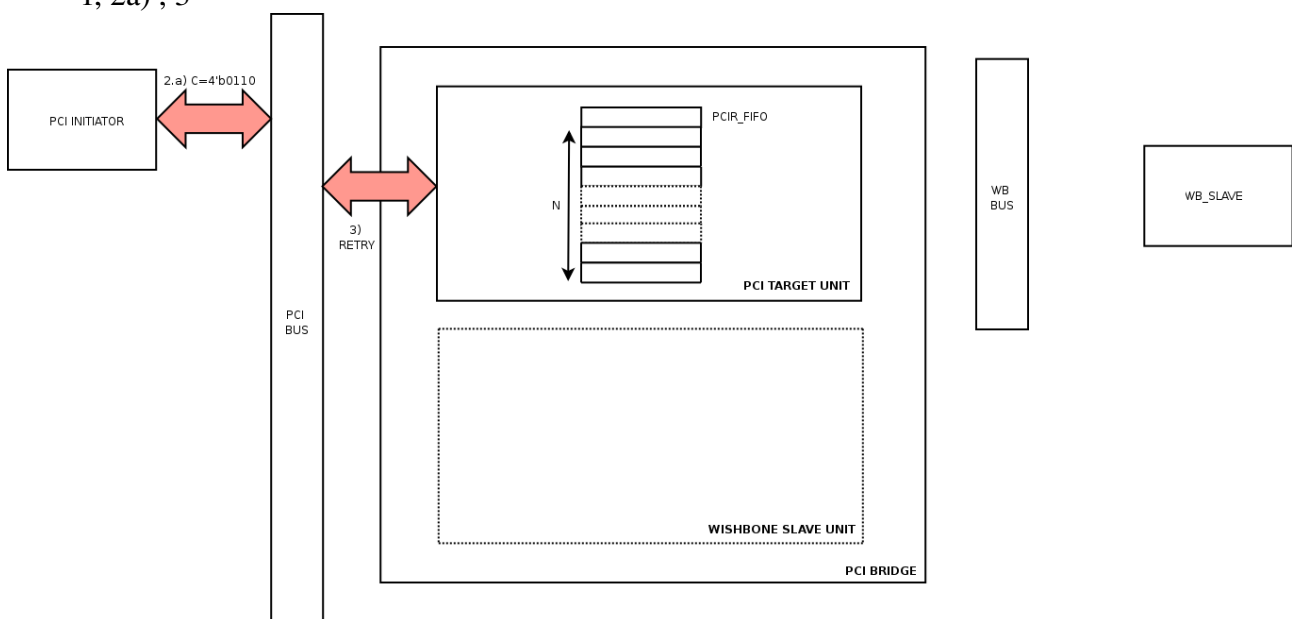
B) 1, 2.c : this case is very simple to reproduce and is left as exercise. Try to follow the previous write transaction with another burst write, without waiting that the first sequence of DWORDs has been flushed from the FIFO; then verify with gtkWave the retry signal on #STOP and #TRDY lanes

Now we want that PCI\_DEV\_1 reads back the written DWORDs; a burst read from a PCI initiator on a chosen address follows, sequentially, the below steps:

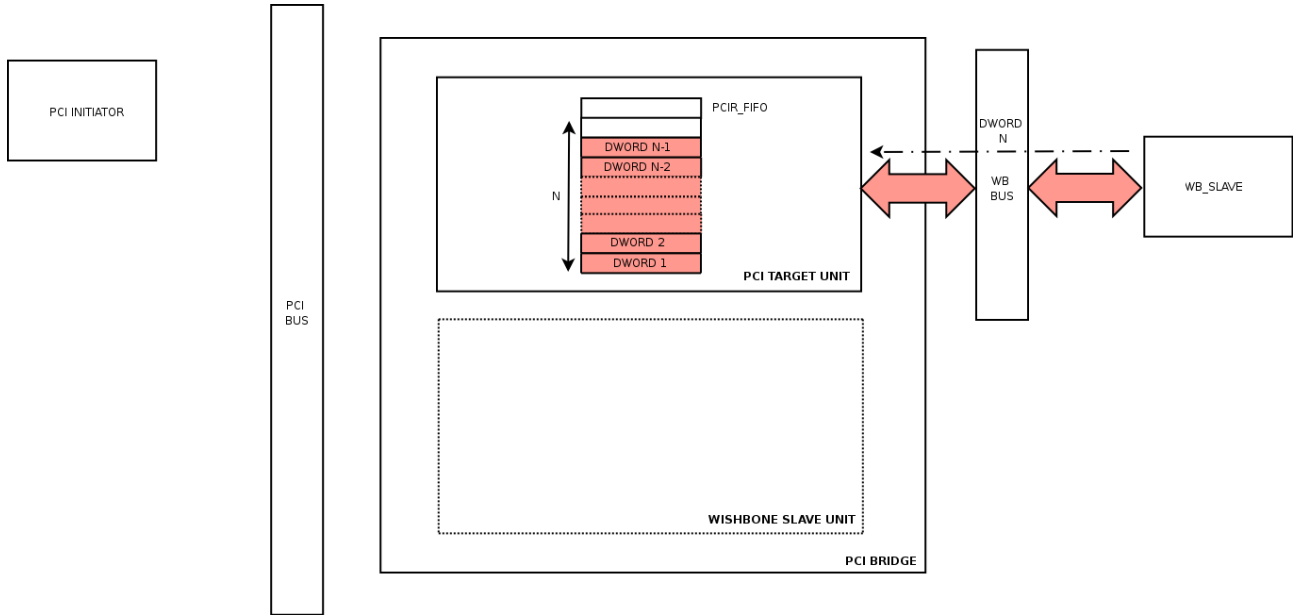
- 1) the value of the "Cache Line Size" register in the PCI configuration space must be set to the number of DWORDs to be read.
  - 2) the PCI initiator starts a read transaction of N DWORDs
    - 2.a) with Command Type = Memory Read (4'b0110) and setting to 1 PREF\_EN
    - 2.b) with Command Type = Memory Read Line (4'b1110) regardless of the value of PREF\_EN
  - 3) the bridge terminates the transaction with a retry signal, wich tells the initiator to repeat the transaction shortly after
  - 4) the bridge performs a sequence of N reads cycles on the WB bus, through WB\_MASTER\_MODULE, starting from the chosen address and incrementing it by one DWORD at each ACK\_O signal received from WB slave. In addition, WB\_MASTER\_MODULE retries or stops the sequence of reads in correspondence of RTY\_O ed ERR\_O signals received from WB slave (in our testbench we will not examine these last two cases; anyway they are very simple to reproduce, as exercise)
- The read values sequentially fill a FIFO: PCIR\_FIFO, wich has a depth defined by `PCIR\_DEPTH (see pci\_constants.v).

5) when N DWORDs are stored in the FIFO, or the FIFO is full, or the WB transaction has been retried or stopped, the PCI initiator can perform another read transaction (which doesn't terminate with a retry signal) during which the bridge flushes the FIFO putting N DWORDs on the PCI bus; if the queue has become empty before N DWORDs are put on the bus (when, for example, it contained a number of DWORDs smaller than N ) then the bridge terminates the transaction with a Target Disconnect signal

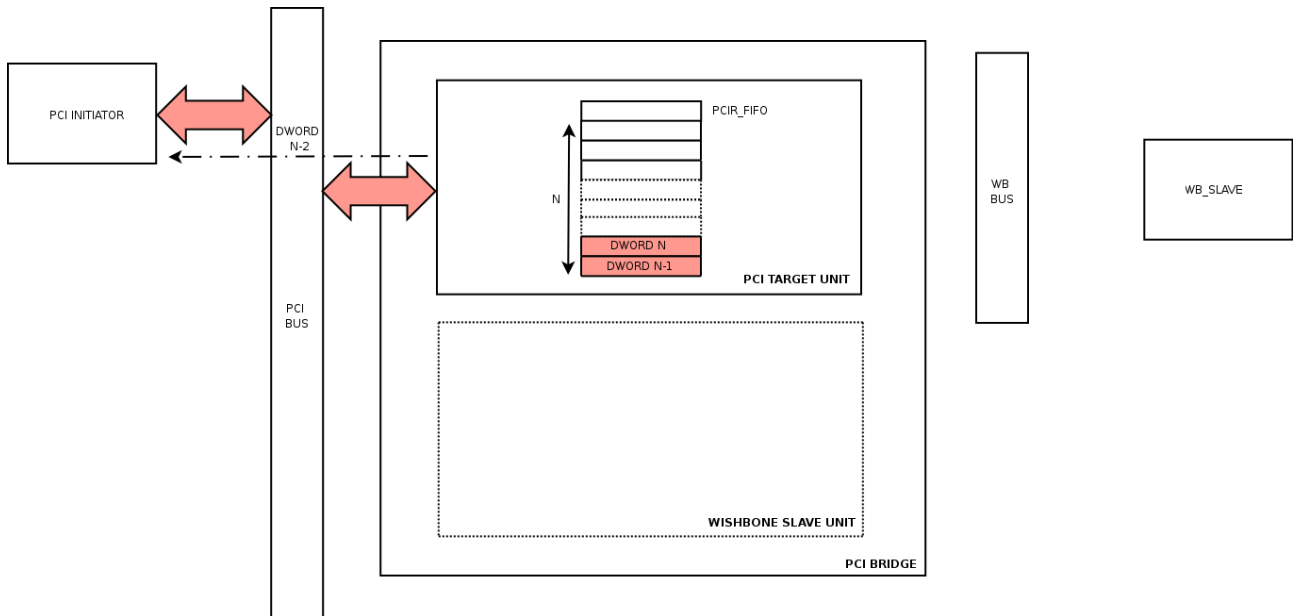
\*\*\*\* 1, 2a) , 3 \*\*\*\*



\*\*\*\* 4 \*\*\*\*



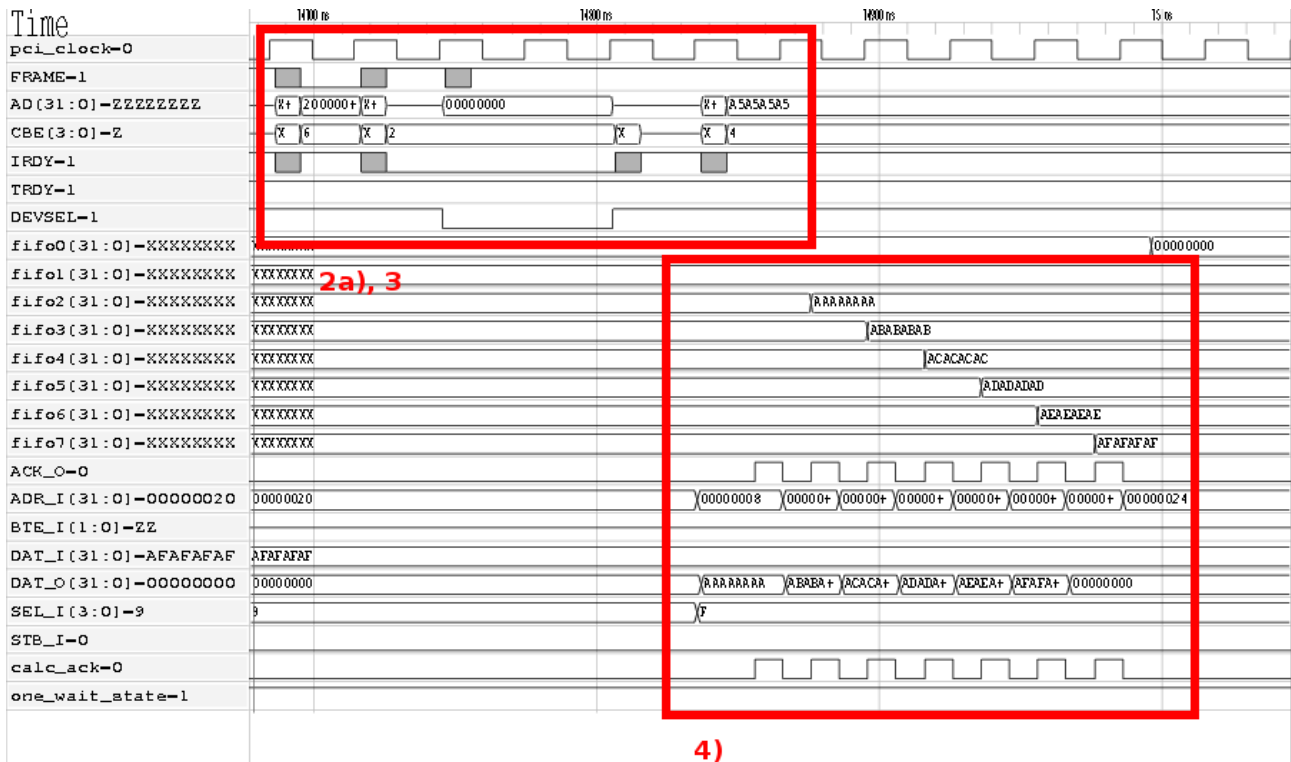
\*\*\*\* 5 \*\*\*\*

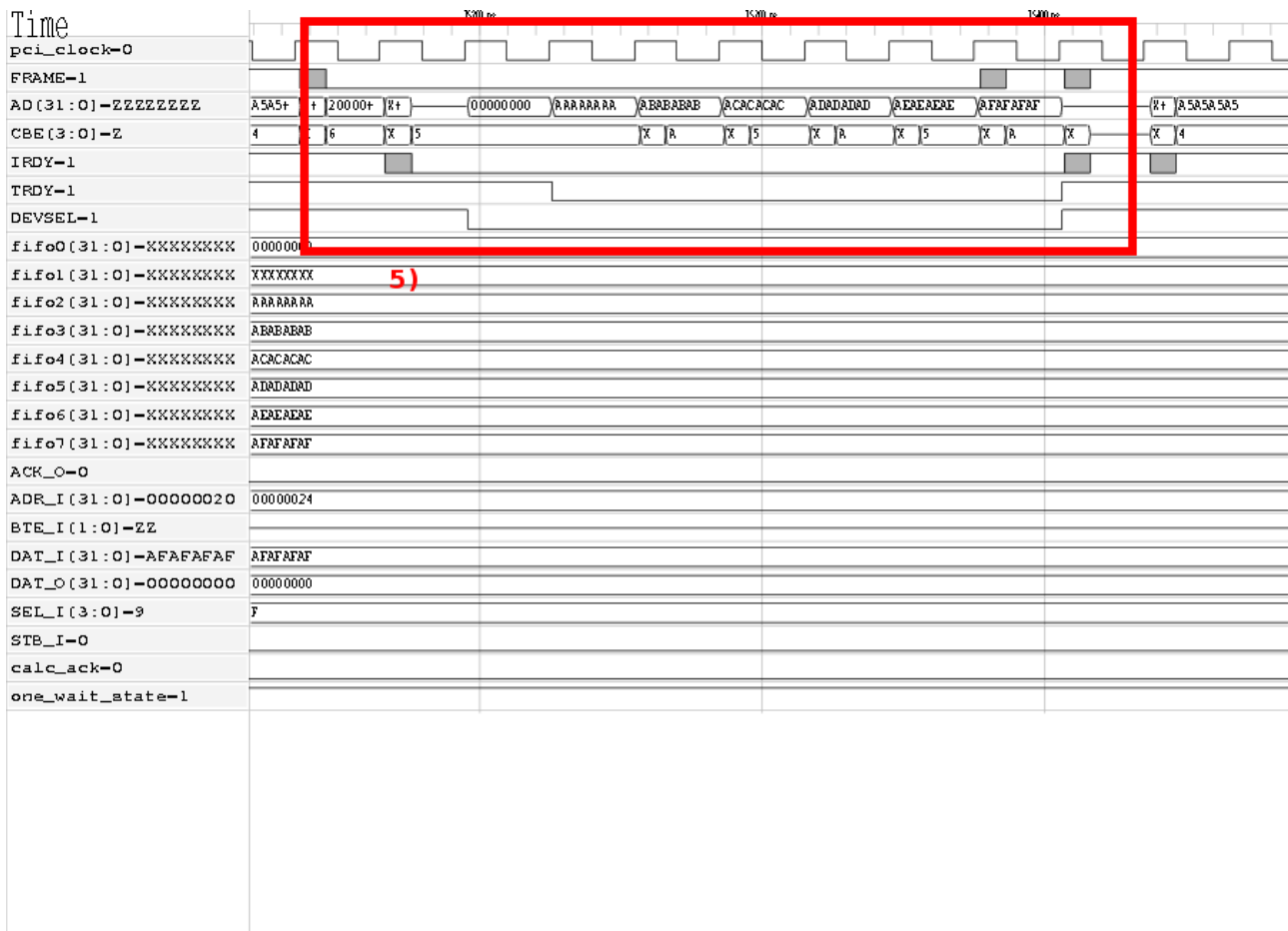


```

/***** file: SYSTEM.v *****/
.
.
.
//PCI_DEV_1 reads 6 DWORDs from WB_SLAVE
do_pause(50); //Wait 50 clk cycles
$display(" Setting the Cache Line Size register to %d word(s)", 8'h8);
configuration_cycle_write(0, 0, 0, 3, 0, 4'hF, {24'h0000_00, 8'h8});
$display(" Setting one wait state for each read cycle (our FASM is synchronous read type)");
wishbone_slave.set_one_wait_state(1);
//SIMPLE_PCI_MEM_READ task has the following arguments: master_number, address, C, byte_en, size, target termination
//C = 0110 --> memory read
//C = 1100 --> memory read multiple
//C = 1110 --> memory read line
$display(" Reading 6 DWORDs from adr 4'h8 of our FASM");
$display("          Step 1: do a PCI read transaction and get a retry signal");
SIMPLE_PCI_MEM_READ (1, `TAR0_BASE_ADDR_1 + 4'h8, 4'b0110, ($random % 14), 1'h1, `Test_Target_Start_Delayed_Read);
do_pause(20); //Wait 20 clk cycles
$display("          Step 2: retry the previous transaction");
SIMPLE_PCI_MEM_READ (1, `TAR0_BASE_ADDR_1 + 4'h8, 4'b0110, ($random % 14), 3'h6, `Test_Target_Normal_Completion);

```





N.B: if the Cache Line Size configuration register is not set, the PCI initiator can always read N DWORDs - where N == `PCIR\_DEPTH - with Command Type (C) == Memory Read Multiple (4'b1100)

## WB to PCI transactions

In a WB to PCI read/write transaction, a wishbone module has to perform bus mastering directly on the WB bus and indirectly (through the bridge) on the PCI bus; in this situation too we have two FIFOs on the bridge that buffer written/read DWORDs from/to the WB master, while translating a WB cycle into a PCI one; FIFOs' depths are defined in pci\_constants.v (WBW\_DEPTH, WBR\_DEPTH). The way of proceeding, in these transaction, is very similar to the PCI to WB ones; at this point, the reader already has all the basic knowledge to quickly understand these cycles and therefore we won't examine them in detail. The tesbench's verilog code is now totally self-explanatory; anyway, consider the following two notes:

- 1) The WB\_MASTER module has been taken from another open hardware project: WB dma (look at [www.opencores.org](http://www.opencores.org)), in order to replace the equivalent module in the official bridge's testbench (which was much more complex); inside the module's code you will find four basic tasks, which are self-explanatory: wb\_wr1, wb\_wr4, wb\_rd1, wb\_rd4
- 2) WB\_MASTER is linked to the WB bus with a point to point interconnection with the bridge's pci\_wb\_slave