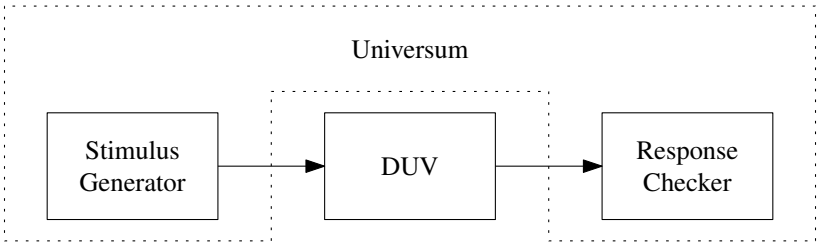
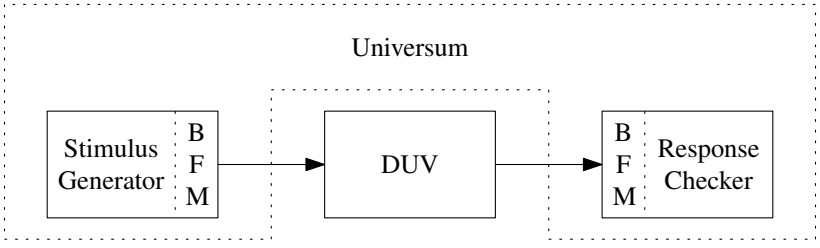
1 Bus Functional Models

Die *Testbench* eines *Design under Verification* (DUV) modelliert das das DUV umgebende Universum: ein Stimulus-Generator stellt alle in diesen Universum generierten und für das DUV relevanten Signale zur Verfügung, und ein *Response Checker* prüft die vom DUV generierten Signale:

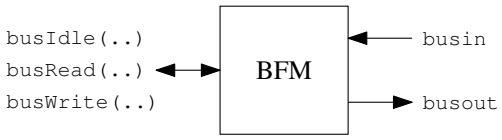


Da das DUV speziell bei SoC-Entwürfen meistens Busprotokolle wie Wishbone oder AMBA (AHB oder APB) verwendet, ist die Generierung eines korrekten Stimulus meist aufwändig. Daher wird zur Verifikation die Busschnittstelle oft ausgeklammert und nur das DUV selbst verifiziert – dies birgt allerdings natürlich die Gefahr, dass dabei Fehler in der Busschnittstelle und ihrer Anbindung an das DUV unentdeckt bleiben.

Idealerweise wird das DUV also gemeinsam mit seinen Schnittstellen verifiziert; dazu muss jedoch die Testbench auch das Busprotokoll unterstützen. Eine einfache, aber effektive Methode, die nicht nur den Abstraktionsgrad der Testbench stark erhöht, sondern auch eine hohe Wiederverwertbarkeit bietet, sind *Bus Functional Models* (BFM):



Ein BFM besteht aus einer Sammlung (sprich: *Package* !) von Datentypen, Konstanten und (*behavioral* !) Funktionen, die die verschiedenen möglichen Buszugriffe (zum Beispiel: *Idle*, *Write*, *Read*, *Burst Write* und *Burst Read*) kapseln:



In einer Testbench könnte der Test eines RAMs daher zum Beispiel so aussehen:

```
for i in 0 to cEndAddress loop
    busWrite(std_ulogic_vector(to_unsigned(i, gWidth)), -- address
             std_ulogic_vector(to_unsigned(i, gWidth)), -- data
             busin, busout);
    busRead(std_ulogic_vector(to_unsigned(i, gWidth)),  -- address
             busin, busout, rd);

    assert rd = std_ulogic_vector(to_unsigned(i, gWidth))
             report "wrong data" severity error;
end loop;
```

Die Manipulationen der einzelnen *Signals* wird dabei in Funktionen gekapselt:

```
type aBusIn is record
    dat_i : std_ulogic_vector(cWidth-1 downto 0);
    -- ...
end record;

type aBusOut is record
    we_o  : std_ulogic;
    adr_o : std_ulogic_vector(cWidth-1 downto 0);
    dat_o : std_ulogic_vector(cWidth-1 downto 0);
    -- ...
end record;
-- ...
procedure busWrite (
    constant addr   : in  std_ulogic_vector(cWidth-1 downto 0);
    constant data   : in  std_ulogic_vector(cWidth-1 downto 0);
    signal   busin  : in  aBusIn;
    signal   busout : out aBusOut) is
begin
    -- clock edge 0
    wait on busin.clk_i until busin.clk_i = '1';

    busout.adr_o <= addr;
    busout.dat_o <= data;
    busout.we_o  <= '1';

    -- ...
end busWrite;
```

1.1 Signal-Parameter

Die Prozeduren `busWrite` und `busRead` haben jeweils die beiden *Signal* -Parameter `busin` und `busout`. *Signal* -Parameter erlauben einer Prozedur, Werte von *Signals* zu lesen und

auch auf diese zu schreiben [Ash02]. Dabei gelten jedoch die gleichen Einschränkungen wie bei der Verwendung von *Signals* in einer *Entity* , insbesondere muss also die Richtung der Signals definiert werden. Daher ist es notwendig, wie im voranstehenden Listing die *Signals* entsprechend ihrer Richtung auf zwei *Records* aufzuteilen[1].

2 Wishbone

Wishbone [Ope02] (genauer: *WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores*) ist ein offenes (*public domain*) Bussystem für SoC-Entwürfe, so verwenden beispielsweise die meisten Cores von *OpenCores* (`www.opencores.org`) und auch Lattices Mico32 [Lat07] diesen Bus.

Implementieren Sie ein BFM für den Wishbone-Bus:

- Die Breite des Daten- bzw. Adressbusses wird auf 32 bzw. 8 Bit festgelegt. Beachten Sie dabei jedoch, dass natürlich keine *Magic Numbers* verwendet werden dürfen!
- Die wichtigsten Transferarten des Wishbone-Busses sind *Single Read* ([Ope02], S. 48), *Single Write* (S. 50), *Block Read* (S. 54) und *Block Write* (S. 57). Diese, und natürlich der *Idle* -Zyklus, müssen unterstützt werden.
- Der RMW-Zyklus sowie *Registered Feedback Bus Cycles* werden für diese Übung vernachlässigt.
- Ein byteweiser Zugriff ist nicht notwendig, das Signal `sel_o` kann also während eines Zugriffs immer an allen Stellen `'1'` sein.

Erstellen Sie aufbauend auf Ihr BFM eine Testbench, die das zur Verfügung gestellte Wishbone-RAM-Modell verifiziert[2]. Dabei können Sie, genau wie in den obigen Abbildungen, von einem Single-Master-System ausgehen, das Sie direkt mit dem DUV verbinden: der Stimulus-Generator ist also Teil einer Testbench, die die Buszugriffe mit Hilfe des BFMs durchführt.

Hinweise dazu:

- *Muss die maximale Anzahl der Waitstates, die der Slave erzeugen darf, begrenzt werden?*

  Dies wäre als Fehlerbehandlung interessant, falls der Slave überhaupt nicht antwortet. Es kann also zB über einen Parameter oder eine Konstante geprüft werden, ob eine maximale Anzahl von Waitstates vergangen ist und dann ein Fehler ausgegeben werden. Dies ist für die Übung aber nicht notwendig.

- *Müssen vom Master eingefügte Waitstates simuliert werden?*

  Auch dies ist für eine umfangreichere Verifikation natürlich sinnvoll, jedoch im Sinne

[1] Alternativ könnte ein ein einem *Package* deklariertes *Signal* verwendet werden.
[2] Beachten Sie, dass eine Testbench selbstüberprüfend sein muss! Wie können Sie das bei einem RAM einfach erreichen?

der Aufwandsbegrenzung der Übung auch nicht notwendig.

□ Beantworten Sie die folgenden, für die Funktionen des BFMs durchaus interessanten, Fragen:

- Welchen Effekt hat die VHDL-Anweisung `wait until clk_i = '1';`? Was passiert, wenn `clk_i` zum Zeitpunkt des Aufrufs bereits `'1'` ist?
- Welchen Effekt hat die VHDL-Anweisung `wait on clk_i until ack_i = '1';`?
- Warum müssen für die Funktionen des BFM Signal-Parameter verwendet werden (im Gegensatz zu *Constants* oder *Variables*)?

Weitere Informationen

- Für die weiteren Übungen werden Sie zum Teil Ihren im fünften Semester entwickelten PROL16 benötigen. Bitte stellen Sie sicher, dass Sie den Quellcode zur Verfügung haben!

Literatur

[Ash02] Peter J. Ashenden. *The Designer's Guide To VHDL*. Morgen Kaufmann, second edition, 2002.

[Lat07] Lattice Semiconductor Corporation. *LatticeMico32 Processor Reference Manual*, August 2007.

[Ope02] OpenCores Organization. *WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores, Rev B.3*, September 2002. http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf.

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*
>
> Brian W. Kernighan

1 Beantwortung der Fragen

2 Source Code

../../src/WishboneBFM–p.vhd

```
1  --
       ------------------------------------------------------------------------

2  -- Title      : Wishbone BFM
3  -- Project    : AMV2
4  -- Author     : Bernhard Selymes
5  --
       ------------------------------------------------------------------------

6  -- Description:
7  --
8  --
       ------------------------------------------------------------------------

9
10 library ieee;
11 use ieee.std_logic_1164.all;
12
13 package WishboneBFM is
14
15   constant cDataWidth : natural := 32;
16   constant cAddrWidth : natural := 8;
17   constant cByte      : natural := 8;
18
19   type aBusIn is record
20     clk_i : std_ulogic;
21    rst_i : std_ulogic;
22     dat_i : std_ulogic_vector(cDataWidth-1 downto 0);
23     ack_i : std_ulogic;
24   end record;
25
26   type aBusOut is record
27     we_o : std_ulogic;
28     adr_o : std_ulogic_vector(cAddrWidth-1 downto 0);
29     dat_o : std_ulogic_vector(cDataWidth-1 downto 0);
30     sel_o : std_ulogic_vector((cDataWidth/cByte)-1 downto 0);
31    stb_o : std_ulogic;
32    cyc_o : std_ulogic;
33   end record;
34
35   type aAddrBlock is array (natural range <>) of std_ulogic_vector(
       cAddrWidth-1 downto 0);
36   type aDataBlock is array (natural range <>) of std_ulogic_vector(
       cDataWidth-1 downto 0);
37
38   constant busInInit : aBusIn := (clk_i   => '0',
39                                   rst_i   => '0',
40                                   dat_i   => (others => '0'),
41                                   ack_i   => '0');
42
43   constant busOutInit   : aBusOut   := (we_o    => '0',
44                                 adr_o => (others => '0'),
45                                 dat_o => (others => '0'),
46                                 sel_o => (others => '0'),
```

```
47                                stb_o => '0',
48                                cyc_o => '0');
49
50    procedure SingleRead (
51      constant addr : in std_ulogic_vector(cAddrWidth-1 downto 0);
52      variable data : out std_ulogic_vector(cDataWidth-1 downto 0); --
          variable ?
53      signal busin  : in aBusIn;
54      signal busout : out aBusOut);
55
56    procedure SingleWrite (
57      constant addr : in std_ulogic_vector(cAddrWidth-1 downto 0);
58      constant data : in std_ulogic_vector(cDataWidth-1 downto 0);
59      signal busin  : in aBusIn;
60      signal busout : out aBusOut);
61
62    procedure BlockRead (
63      constant addr : in aAddrBlock;
64      variable data : out aDataBlock;
65     constant len  : in natural;
66      signal busin  : in aBusIn;
67      signal busout : out aBusOut);
68
69    procedure BlockWrite (
70      constant addr : in aAddrBlock;
71      constant data : in aDataBlock;
72     constant len  : in natural;
73      signal busin  : in aBusIn;
74      signal busout : out aBusOut);
75
76    procedure Idle (
77      signal busout : out aBusOut);
78
79  end WishboneBFM;
80
81  package body WishboneBFM is
82
83    -- Single Read
84    procedure SingleRead (
85      constant addr : in std_ulogic_vector(cAddrWidth-1 downto 0);
86      variable data : out std_ulogic_vector(cDataWidth-1 downto 0);
87      signal busin  : in aBusIn;
88      signal busout : out aBusOut) is
89    begin
90      -- clock edge 0
91      wait on busin.clk_i until busin.clk_i = '1';
92     busout.adr_o <= addr;
93     busout.we_o <= '0';
94     busout.sel_o <= (others => '1');
95     busout.cyc_o <= '1';
96     busout.stb_o <= '1';
97
98     -- setup, edge 1
99     wait on busin.clk_i until busin.ack_i = '1';
100
101    -- clock edge 1
102    data := busin.dat_i;
103    busout.stb_o <= '0';
104    busout.cyc_o <= '0';
105   end SingleRead;
```

```
106
107   -- Single Write
108   procedure SingleWrite (
109     constant addr : in std_ulogic_vector(cAddrWidth-1 downto 0);
110     constant data : in std_ulogic_vector(cDataWidth-1 downto 0);
111     signal busin  : in aBusIn;
112     signal busout : out aBusOut) is
113   begin
114     -- clock edge 0
115     wait on busin.clk_i until busin.clk_i = '1';
116    busout.adr_o <= addr;
117    busout.dat_o <= data;
118    busout.we_o <= '1';
119    busout.sel_o <= (others => '1');
120    busout.cyc_o <= '1';
121    busout.stb_o <= '1';
122
123    -- setup, edge 1
124    wait on busin.clk_i until busin.ack_i = '1';
125
126    -- clock edge 1
127    busout.stb_o <= '0';
128    busout.cyc_o <= '0';
129   end SingleWrite;
130
131   -- Block Read
132   procedure BlockRead (
133     constant addr : in aAddrBlock;
134     variable data : out aDataBlock;
135    constant len  : in natural;
136     signal busin  : in aBusIn;
137     signal busout : out aBusOut) is
138   begin
139
140    Reading: for i in 0 to len-1 loop
141      -- clock edge 0
142       wait on busin.clk_i until busin.clk_i = '1';
143      busout.adr_o <= addr(i);
144      busout.we_o <= '0';
145      busout.sel_o <= (others => '1');
146      busout.cyc_o <= '1';
147      busout.stb_o <= '1';
148
149      -- setup, edge 1
150      wait on busin.clk_i until busin.ack_i = '1';
151
152      -- clock edge 1
153      data(i) := busin.dat_i;
154    end loop;
155
156    busout.stb_o <= '0';
157    busout.cyc_o <= '0';
158
159   end BlockRead;
160
161   -- Block Write
162   procedure BlockWrite (
163     constant addr : in aAddrBlock;
164     constant data : in aDataBlock;
165    constant len  : in natural;
```

```
166      signal busin  : in aBusIn;
167      signal busout : out aBusOut) is
168    begin
169
170     Writing: for i in 0 to len-1 loop
171       -- clock edge 0
172        wait on busin.clk_i until busin.clk_i = '1';
173       busout.adr_o <= addr(i);
174       busout.dat_o <= data(i);
175       busout.we_o <= '1';
176       busout.sel_o <= (others => '1');
177       busout.cyc_o <= '1';
178       busout.stb_o <= '1';
179
180       -- setup, edge 1
181       wait on busin.clk_i until busin.ack_i = '1';
182     end loop;
183
184     busout.stb_o <= '0';
185     busout.cyc_o <= '0';
186
187    end BlockWrite;
188
189    -- Idle
190    procedure Idle (
191      signal busout : out aBusOut) is
192    begin
193     busout.adr_o <= (others => 'Z');
194     busout.dat_o <= (others => 'Z');
195     busout.we_o <= 'Z';
196     busout.sel_o <= (others => 'Z');
197     busout.cyc_o <= 'Z';
198     busout.stb_o <= 'Z';
199    end Idle;
200
201  end WishboneBFM;
```

../../src/WishboneBFM–tb.vhd

```
1  --
       -------------------------------------------------------------------------------
2  -- Title      : Wishbone BFM tb
3  -- Project    : AMV2
4  -- Author     : Reinhard Penn
5  --
       -------------------------------------------------------------------------------
6  -- Description:
7  --
8  --
       -------------------------------------------------------------------------------
9
10 library ieee;
11 use ieee.std_logic_1164.all;
12 use ieee.numeric_std.all;
13 use work.WishboneBFM.all;
14
15 entity tbWishboneBFM is
16 end entity;
```

```
17
18
19 architecture bhv of tbWishboneBFM is
20
21 COMPONENT RAM PORT (
22 clk_i : in   STD_ULOGIC;
23 rst_i : in   STD_ULOGIC;
24 adr_i : in   STD_ULOGIC_VECTOR(cAddrWidth-1 downto 0);
25 dat_i : in   STD_ULOGIC_VECTOR(cDataWidth-1 downto 0);
26 sel_i : in   STD_ULOGIC_VECTOR((cDataWidth/8)-1 downto 0);
27 cyc_i : in   STD_ULOGIC;
28 stb_i : in   STD_ULOGIC;
29 we_i  : in   STD_ULOGIC;
30 dat_o : out  STD_ULOGIC_VECTOR(cDataWidth-1 downto 0);
31 ack_o : out  STD_ULOGIC
32  );
33 END COMPONENT;
34
35 signal busIn   : aBusIn := busInInit;
36 signal busOut  : aBusOut   := busOutInit;
37
38 constant cEndAddress : natural := (2**cAddrWidth)-1;
39 constant testInput1 : std_ulogic_vector(cDataWidth-1 downto 0) := x"
      AAAAAAAA";
40 constant testInput2 : std_ulogic_vector(cDataWidth-1 downto 0) := x"
      55555555";
41
42 begin
43
44 DUV : RAM PORT MAP(
45 clk_i => busIn.clk_i,
46 rst_i => busIn.rst_i,
47 adr_i => busOut.adr_o,
48 dat_i => busOut.dat_o,
49 sel_i => busOut.sel_o,
50 cyc_i => busOut.cyc_o,
51 stb_i => busOut.stb_o,
52 we_i  => busOut.we_o,
53 dat_o => busIn.dat_i,
54 ack_o => busIn.ack_i
55 );
56
57 CLOCK:
58 busIn.clk_i <=  '1' after 5 ns when busIn.clk_i = '0' else
59         '0' after 5 ns when busIn.clk_i = '1';
60
61 Stimuli : process is
62 variable rd : std_ulogic_vector(cDataWidth-1 downto 0) := (others => '0');
63 variable addressBlock   : aAddrBlock(0 to cEndAddress) := (others => (
      others => '0'));
64 variable dataBlock      : aDataBlock(0 to cEndAddress) := (others => (
      others => '0'));
65 variable readDataBlock  : aDataBlock(0 to cEndAddress) := (others => (
      others => '0'));
66 begin
67
68   --test Single Read Write with data 10101...
69    for i in 0 to cEndAddress loop
70       SingleWrite(std_ulogic_vector(to_unsigned(i, cAddrWidth)), -- address
71       testInput1, -- data
```

```
72          busIn, busOut);
73       SingleRead(std_ulogic_vector(to_unsigned(i, cAddrWidth)), -- address
74          rd, busIn, busOut);
75
76       assert rd = testInput1
77          report "SindleTest: TestInput1 is wrong"
78          severity error;
79    end loop;
80
81    --test Single Read Write with data 010101...
82    for i in 0 to cEndAddress loop
83       SingleWrite(std_ulogic_vector(to_unsigned(i, cAddrWidth)), -- address
84       testInput2, -- data
85          busIn, busOut);
86       SingleRead(std_ulogic_vector(to_unsigned(i, cAddrWidth)), -- address
87          rd, busIn, busOut);
88
89       assert rd = testInput2
90          report "SingleTest: TestInput2 is wrong"
91          severity error;
92    end loop;
93
94    --test Block Read Write with data 1010101...
95    for i in 0 to cEndAddress loop
96       addressBlock(i) := std_ulogic_vector(to_unsigned(i, cAddrWidth));
97       dataBlock(i)    := testInput1;
98    end loop;
99
100   BlockWrite(addressBlock, -- address
101      dataBlock, -- data
102      cEndAddress+1, busIn, busOut);
103
104   BlockRead(addressBlock, -- address
105      readDataBlock, -- data
106      cEndAddress+1, busIn, busOut);
107
108   for i in 0 to cEndAddress loop
109      assert readDataBlock(i) = testInput1
110         report "BlockTest: TestInput1 is wrong"
111         severity error;
112   end loop;
113
114
115      --test Block Read Write with data 01010101...
116   for i in 0 to cEndAddress loop
117      addressBlock(i) := std_ulogic_vector(to_unsigned(i, cAddrWidth));
118      dataBlock(i)    := testInput2;
119   end loop;
120
121   BlockWrite(addressBlock, -- address
122      dataBlock, -- data
123      cEndAddress+1, busIn, busOut);
124
125   BlockRead(addressBlock, -- address
126      readDataBlock, -- data
127      cEndAddress+1, busIn, busOut);
128
129   for i in 0 to cEndAddress loop
130      assert readDataBlock(i) = testInput2
131         report "BlockTest: TestInput2 is wrong"
```

```
132           severity error;
133     end loop;
134 
135 assert false
136       report "This is not a failure: Simulation finished !!!"
137       severity failure;
138 
139       wait;
140 end process;
141 
142 end architecture;
```

../../sim/Compile.do

```
1 vlog ../src/ram.sv
2 vcom ../src/WishboneBFM-p.vhd
3 vcom ../src/WishboneBFM-tb.vhd
```

../../sim/Sim.do

```
1 vsim -novopt tbWishboneBFM
2 add wave -position end  sim:/tbwishbonebfm/busIn
3 add wave -position end  sim:/tbwishbonebfm/busOut
4 add wave -position end  sim:/tbwishbonebfm/testInput1
5 add wave -position end  sim:/tbwishbonebfm/testInput2
6 log -r *
7 run -all
```