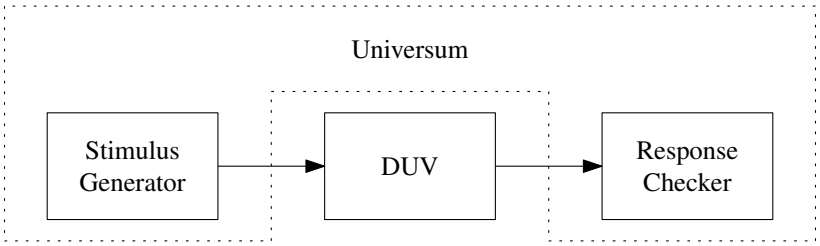
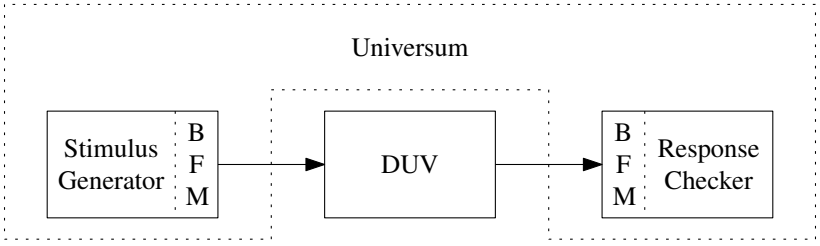
1 Bus Functional Models

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:

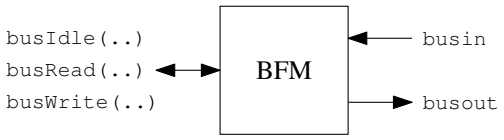


Da das DUV speziell bei SoC-Entwürfen meistens Busprotokolle wie Wishbone oder AMBA (AHB oder APB) verwendet, ist die Generierung eines korrekten Stimulus meist aufwändig. Daher wird zur Verifikation die Busschnittstelle oft ausgeklammert und nur das DUV selbst verifiziert – dies birgt allerdings natürlich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