



Designing Procedural-Based Behavioral Bus Functional Models for High Performance Verification

Gregg D. Lahti

gregg.d.lahti@intel.com

Tim L. Wilson

tim.l.wilson@intel.com

Intel Corporation



Purpose

- **Enable faster testing, more simulation cycles**
- **Find bugs faster, higher silicon tapeout confidence**
- **Self-checking, automated for better productivity**



Outline

- **Test bench basics**
- **Modeling devices as Bus Functional Models (BFMs)**
- **BFM high-level testing methodology**
- **High-level testing gotchas**
- **Synopsys PCI Source Model mods**
- **Conclusion**



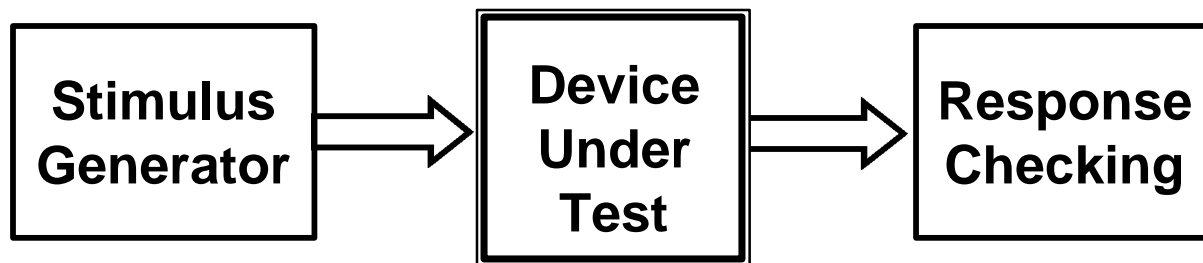
The Problem

- **Verification is 50% of Design Effort**
- **Need fast-as-possible cycles/second simulation speed**
- **Need high level of test complexity, control**
- **Simulate designs without IP core or gate-level models**



Test Bench Basics

Test bench block diagram



- **Device Under Test is your design**



Test Bench Basics

- **Stimulus application methods**
 - File-based vectors
 - Basic assertion statements (VHDL)
 - Use of Bus Functional Language in files to be applied through a model
 - **Synopsys PCI Source models**



Test Bench Basics

- **Response checking**
 - Human-based, look at simulation
 - Automated (preferred)
 - **Trace file comparison**
 - **Run once, verify, run again and compare against “golden” trace files**
 - **Self-checking**
 - **Test bench checks response during simulation**



Test Bench Basics

- **Trace file comparison problems**
 - Must first guarantee golden trace file manually
 - Requires large vector files hanging around, possible revision control problems
 - Slow, doesn't compensate for cycle slips
 - Won't work with behavioral abstractions



Test Bench Basics

- **Self-checking**

- Requires more model creation time
- No manual comparisons of golden trace to current simulation required
- Can test timing and protocol
- Can allow for cycle slips



Modeling Devices as BFM's

- **BFM acts as stimulus generation device**
 - Cycle and protocol accurate
 - Does timing and protocol checks
 - Accepts stimulus commands in some high-level format
 - **Bus Functional Language in files**
 - **Procedural-based driven**



Modeling Devices as BFM's

- **Driving BFM's**

- Bus Functional Language Code most prevalent
- Synopsys Source Models use it

```
print_msg("PM: memory read to PT to verify  
initialization ");
```

```
read_cycle(mem_read, "00000000",  
1, "0", "86551643", 0, false);
```



Modeling Devices as BFM's

- **Why BFL from files is undesirable**
 - File I/O slow compared to compiled and memory resident VHDL
 - Files must be parsed/processed
 - **Intelligent parser for loops, conditions, etc**
 - **No ability to read-modify-writes**
 - **May require external compiler**
 - **Synchronization of models difficult**
 - **Sampling/driving test bench signals difficult**



Modeling Devices as BFM's

- **BFL code can't do this example:**

```
-- write 32-bit words to every 8-byte address of memory chunk
tv := to_stdlogicvector(X"a5a5e3e3");
for i in 16#00000100# to 16#00000140# loop
  if (i mod 8 = 0) then          -- every other dword
    -- write value to DUT
    wr(conv_std_logic_vector(i,32),tv,cmd);
    tv := tv xor tv0;          -- just for kicks, xor next data
  end if;
end loop;
```



BFM High-Level Testing Methods

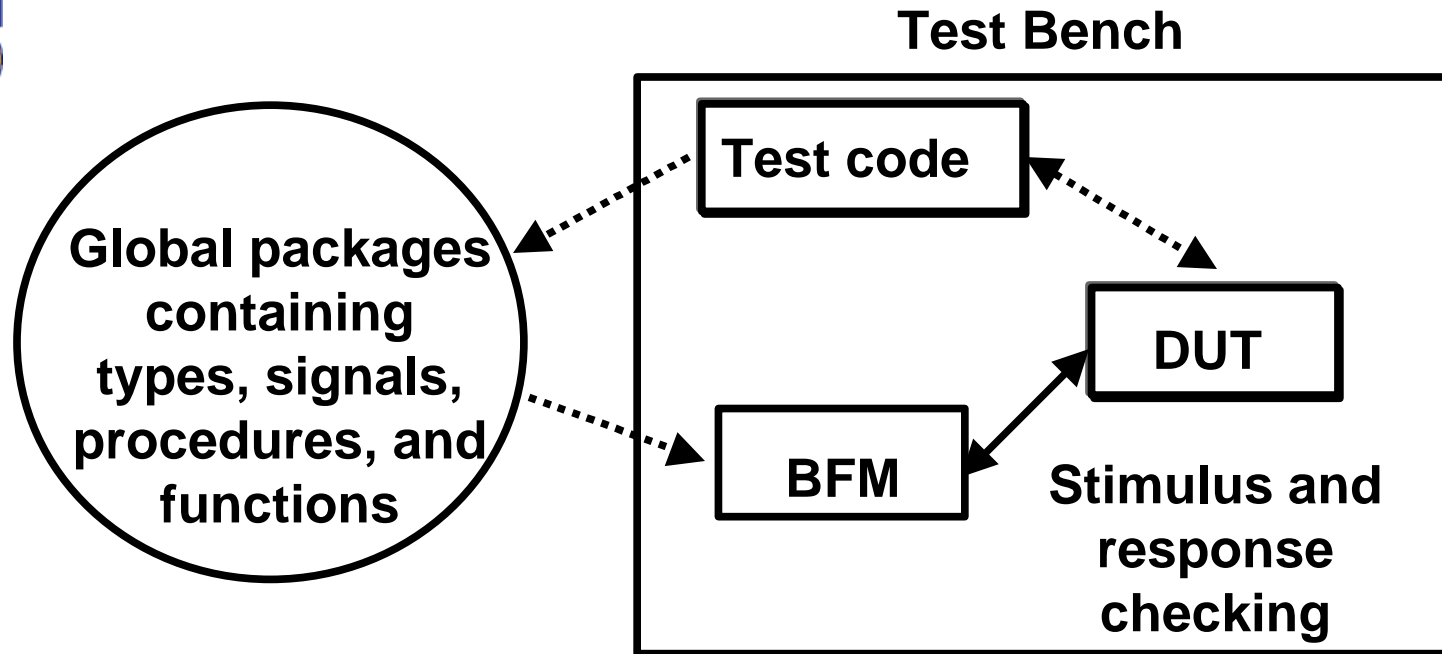
- **Use VHDL procedure calls to drive BFM**
 - Power of VHDL language for test code creation
 - No external parsers required
 - Compiled in native language, faster runtime
 - Better synchronization methods



BFM High-Level Testing Methods

- **Main components of VHDL-driven BFM**
 - BFM's instanced within the test bench
 - Package to contain globally accessible signals
 - Separate test code entity & architectures for stimulus
 - Configurations to bind abstractions

BFM High-Level Testing Methods



- **Global packages**
 - Allow communication of test bench signals and procedures between test code and BFM



BFM High-Level Testing Methods

- **Global package example**

```
type cycle_t is (idle, write, read);
type instruction is record
    bus_cycle : cycle_t; -- cycle type
    addr      : std_logic_vector(31 downto 0); -- address
    data      : std_logic_vector(31 downto 0); -- write/read value
    rdata     : std_logic_vector(31 downto 0); -- actual read data
    request   : std_logic; -- request handshake
    busy      : std_logic; -- busy handshake
end record;
signal cmd : instruction := (idle, (others => '0'),
                             (others => '0'), (others => 'Z'), '0', 'Z');
```



BFM High-Level Testing Methods

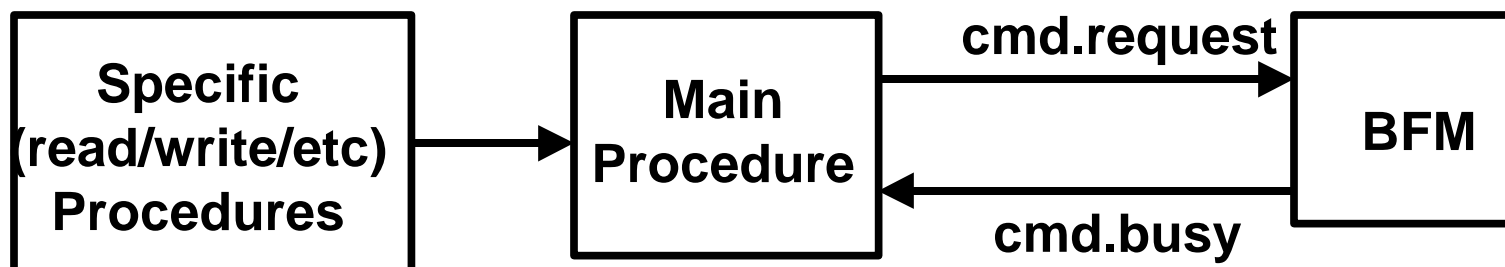
- **BFM procedure call example**

```
swr("00000000", "deadbeef", cmd); -- write cycle  
idle(cmd);                          -- idle  
srd("00000000", "deadbeef", cmd); -- read cycle
```

In VHDL, the global signal “cmd” must be passed with the procedure call to enable read/write access by the procedure

BFM High-Level Testing Methods

- **BFM executes commands via main procedure call and handshaking signals in “cmd” record**





BFM High-Level Testing Methods

- **Operation flow**

- Test code issues specific (read, write, etc) procedure.
- Specific procedure sets up data, calls main procedure.
- Main procedure requests cycle start BFM using `cmd.request` signal if `cmd.busy` is not asserted. If BFM busy, procedure waits until BFM is finished.



BFM High-Level Testing Methods

- **Operation flow continued**
 - BFM does its operation, signals done through cmd.busy signal.
 - Operation flow returns back to test code (specific procedure call origination) for next command.



BFM High-Level Testing Methods

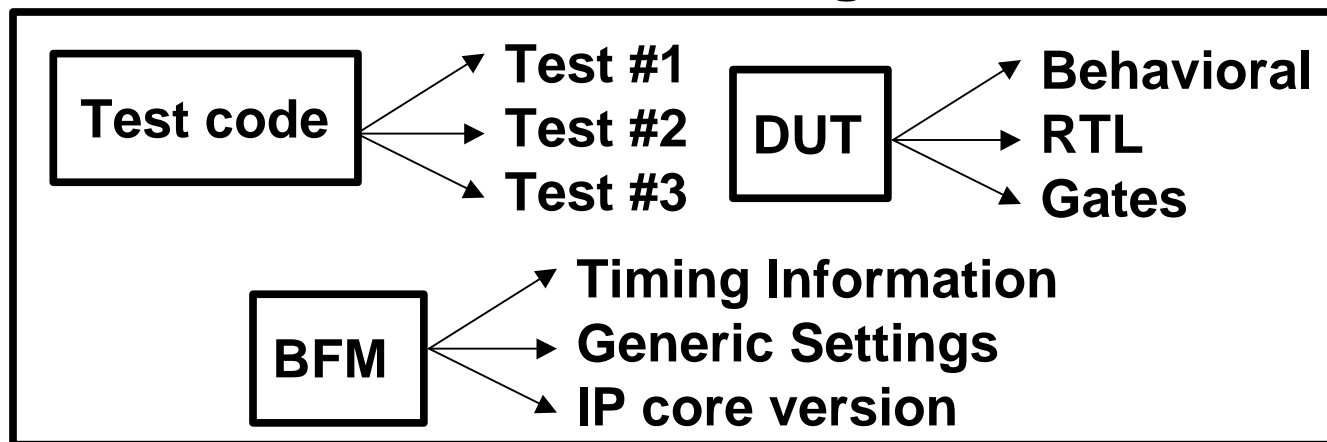
- **Read-modify-write example**

```
-- first read the memory
av := to_stdlogicvector(X"00000020");
srd(av, "babeface", cmd);
idle(cmd);
-- now get the actual data read in the tv variable
get_rdata(tv, cmd);
tv(15 downto 0) := tv(15 downto 0) - 16#0ac1#;
tv(31 downto 16) := tv(31 downto 16) + 16#1fbe#;
-- write it back
wr(av, tv, cmd);
-- read it for verification
rd(av, tv, cmd);
```

BFM High-Level Testing Methods

- **Configurations allow**
 - Binding of different architectures to entities
 - Binding of generic settings

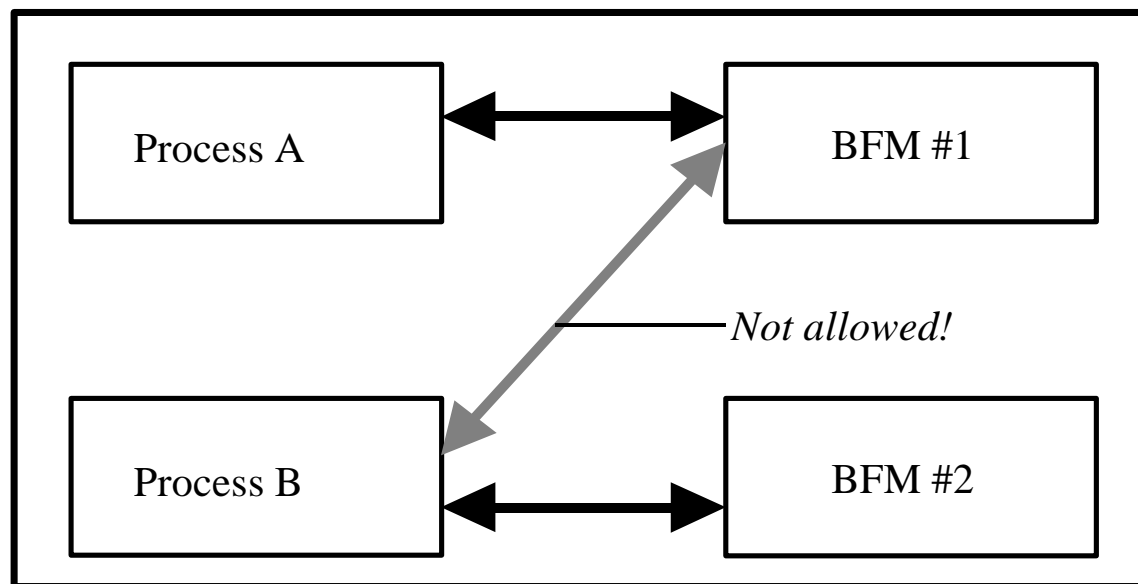
Test bench configuration



High Level Testing Gotchas

- **Multiple BFMs**

- Multiple test code processes drive multiple BFMs





High Level Testing Gotchas

- **Control multiple BFMs of a single type with the same procedure syntax**
 - Use array of records
 - Each BFM has a generic “unique number” assigned through configuration
 - Procedure calls include “unique number” to drive specific BFM
 - Each BFM “pipelines” one instruction, allowing simultaneously execution



BFM High-Level Testing Methods

- **In one project test bench we had**
 - 2 PCI Master BFMs
 - 2 PCI Slave BFMs
 - 1 PCI Monitor/Arbiter
 - 1 486 BFM
 - 2 SRAM/FLASH memory models



High Level Testing Gotchas

- **Coding BFM's for performance**
 - Use as few processes as possible
 - Limit component instantiations
 - Avoid using after delays
 - Use variables in processes
 - Use IEEE library
 - **IEEE packages usually built into simulators**
 - **Conversions & types quicker and proven**



Synopsys PCI Source Model Mods

- **You too can have procedural-driven PCI Source Models!**

- Enables procedure calls to drive PCI master BFM instead of file-resident BFL code
- Patches done to your own source
- Patch files on website

<http://www.primenet.com/~greggl/snug99.html>



Conclusions

- **Procedure-driven BFMs enables higher-level, more complex testing**
- **Enable thorough protocol and timing checks**
- **Finds bugs faster, get better tested silicon sooner**
- **See website for full example code**