A Behavioral VHDL Bus Functional Model for the Pentium (TM) Processor

Stephen Hunt, Ashok Mehta, Dan Patterson, Sandeep Shah
Microprocessor Products Group
Intel Corporation
Santa Clara, CA 95052

Abstract
In the Personal Computer marketplace, hardware system designers are faced with the challenge of building differentiating system-level features around more and more complex standard product microprocessor components. Each new generation of the microprocessor spawns a new generation of the motherboards and systems. Success in the marketplace is often driven by time-to-market for these key leading edge products. This paper discusses the motivation for, and the development of, a behavioral VHDL bus functional model (BFM) for the Pentium processor [1], a new high performance, and Intel386/486(TM) compatible microprocessor from Intel corporation.

Introduction
For many years, system designers have faced the problem of obtaining accurate simulatable representations of the standard components used in the system in a timely enough manner to enable simulation-based validation of ASICs used in the board/system. As the complexity of the standard products increased, the complexity of developing the models followed suit. With the Pentium CPU, Intel undertook the development of a behavioral model of the bus interface in order to ensure timely and accurate model availability to system designers.

First, this paper discusses applications of high-level modeling that have been used to provide models to system designers in the past. Next, the motivations for the application of a VHDL behavioral modeling approach used in the development of the BFM for the Pentium CPU is discussed. We cover important aspects of the BFM environment such as the Bus Functional Language (BFL) and Bus Functional Compiler (iBFC). These allow the user to exercise the BFM with the desired bus cycle sequence. We outline the methodology for the behavioral core model for incorporation of a timing shell around the BFM to model AC timings. We cover the validation method applied to guarantee the BFM's clock-level accuracy to the RTL design. Finally, we present future directions and draw conclusions.
General Applications for High-Level Models

At the Intel Microprocessor Products Group, high-level modeling has played an important role in three fundamental areas related to enhancing both component and system design time-to-market. First, component designers have developed and applied instruction set simulators as a point of reference in order to validate a component-level RTL model implementation. This technique has proven useful for architecture validation utilizing random instruction and manual regression based testing techniques. Second, architects have developed performance simulators in order to make micro-architectural tradeoffs and to establish realistic performance targets for a design implementation. This paper focuses on a third aspect of high-level modeling; system-level validation using a BFM. During the course of the BFM development, we learned that this application is relevant not only to pre- and post-silicon system design but also to pre-silicon component validation in a systems environment; this application increases the quality of the initial silicon samples of the standard product.

All high-level models share one common thread; they are abstract. They don't necessarily reflect the actual low-level design implementation. As a result, they are typically easier to build since the developer of the model need not be concerned with structural design details. They execute faster during simulation, since low-level RTL details are not part of the simulation. They are also easier to maintain during the course of a pre-silicon design evolution since changes in the design implementation are not necessarily even modeled in a high-level behavioral model. Further, those design changes that are specification related and do require changes in the behavioral model are typically easier to implement, since a behavioral representation of a given architectural feature is more abstract and thus tends to be more localized than in the design implementation (i.e., RTL) representation (i.e., it hasn't been decomposed yet).

For system designers, early availability of efficient and accurate high-level models for the standard product components used in the system has become one of the most critical factors in achieving fast time-to-market. The system designer is faced with the task of building differentiating features around the CPU and designing in the cache, I/O, and memory systems that, together, form a complete, operational computer system. As the complexity of the CPU has grown, the requirement for accurate high-level models has become an absolute necessity.

For the standard components being modeled in the system, the model requirements are driven from the point of view of interoperability with commercial EDA environments used by system designers, simulation model performance (i.e., behavioral models are much faster than structural models), powerful control, and test sequencing since the CPU is the heart of the system (and is a very effective test generation tool), and of protection of intellectual property (an Intel requirement). Another very important aspect of applying behavioral modeling for System Design is that of the timing of model availability with respect to the next generation component platform/system design cycle. Pre-silicon availability of an accurate and well-supported component model has been raised by many of our customers as the way to shorten time-to-market. This enables the system
design process to begin as soon as the component-level specification is solid enough to generate a high-level behavioral model.

**Pentium CPU-Cache Platform Model Overview**

A typical Pentium CPU/Cache platform system simulation model environment is shown in Figure 1. It consists of a BFM of the Pentium processor, full functional models (FFMs) of the 82496 cache controller (CC), 82491 cache ram (CR) [2], and the iBFC front end. The memory bus controller (MBC) block is typically designed by the system designer using ASICs. Given the user's BFL source program, the iBFC generates BFL object files along with a listing file. The BFL object files are loaded into an array within the BFM at simulation startup and executed during the simulation run, thus creating the user specified bus cycle sequence. The memory and I/O sub systems required for a complete Pentium CPU based system are not shown.

![Diagram](image)

**FIGURE 1 Pentium CPU/Cache Platform Model Environment**

**Motivations for using VHDL**

VHDL [3] is an industry standard hardware description language. The Department of Defense and industry funded the development that was adopted as an IEEE standard (IEEE 1076-1987). Every major EDA vendor supports some portion of VHDL. The selection of VHDL as the language for the CPU-Cache platform was based on the fundamental goal of making the models available on all those EDA platforms that are used by system designers. In order to further enhance model portability, we chose to code within the Viewlogic VHDL subset since it is most easily ported to the various VHDL EDA platforms. In order to support our Verilog users, we translated our VHDL model to Verilog.
VHDL was designed to be very general in its applicability to system design. VHDL supports powerful constructs such as user-defined types, records, and concurrency that permit high levels of abstractions in the model. An example is the use of records to store the type and attributes of a bus cycle request. This enabled us to model the bus protocol without regard to the underlying structural implementation. The EDA industry and users are working to further enhance behavioral modeling capability in the language in 1992. Further, emerging logic system standards such as IEEE 1164 will help model interoperability as soon as the user library IEEE is built in by EDA vendors as STD.

Bus Functional Model Overview

The BFM for the Pentium processor is a software simulation model that emulates the Pentium CPU bus cycle protocols. It does not execute the instruction set, nor does it maintain its internal caches. The primary usage of this model is to exercise/debug a hardware design based on the Pentium processor; hence, it models only the bus-level interface (functionality and timing). Since the BFM does not execute instructions, an external BFL is needed to provide commands to invoke various bus cycles.

Using the BFL, the user can specify the required bus cycle sequence, which is then executed by the BFM during a simulation run. These sequences are typically designed to exercise the various standard components at the system level as well provide a means to exercise one's ASIC designs. BFL commands are executed sequentially unless an exception, such as an interrupt or other bus cycle altering condition, occurs. In this case, the exception triggers transfer of control to an exception service routine (ESR). Table 1 outlines the BFL commands

<table>
<thead>
<tr>
<th>BFL Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Commands</td>
<td>Typical Pentium CPU bus cycles</td>
</tr>
<tr>
<td>Read</td>
<td>Generate READ bus cycle</td>
</tr>
<tr>
<td>Write</td>
<td>Generate WRITE bus cycle</td>
</tr>
<tr>
<td>Idle</td>
<td>Put Pentium CPU into bus idle state</td>
</tr>
<tr>
<td>Special Commands</td>
<td>Special Pentium CPU bus cycles</td>
</tr>
<tr>
<td>Invd</td>
<td>Pentium CPU special flush cycle</td>
</tr>
<tr>
<td>Wbinvd</td>
<td>Pentium CPU special writeback cycle</td>
</tr>
<tr>
<td>Halt</td>
<td>Pentium CPU special halt cycle</td>
</tr>
<tr>
<td>Shutdown</td>
<td>Pentium CPU special shutdown cycle</td>
</tr>
<tr>
<td>Btm</td>
<td>Pentium CPU special branch trace cycle</td>
</tr>
</tbody>
</table>

Table 1: BFL Command Summary
<table>
<thead>
<tr>
<th>BFL Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign</td>
<td>Toggle execution tracing outputs</td>
</tr>
<tr>
<td>Inq_result</td>
<td>Special cache lookup result command</td>
</tr>
<tr>
<td>Instrumentation commands</td>
<td>Useful for synchronizing with external H/W</td>
</tr>
<tr>
<td>Sleep</td>
<td>Put Pentium CPU into bus idle state until awakened</td>
</tr>
<tr>
<td>Wakeup</td>
<td>Wake up another Pentium CPU</td>
</tr>
<tr>
<td>General purpose commands</td>
<td>Control functions</td>
</tr>
<tr>
<td>Control_flags</td>
<td>Set/reset various processor register bits</td>
</tr>
<tr>
<td>Restart</td>
<td>Start back at the beginning of BFL flow</td>
</tr>
<tr>
<td>Print</td>
<td>Controls BFM messages in log file output</td>
</tr>
<tr>
<td>Printf</td>
<td>Print a string and/or value of user variable</td>
</tr>
<tr>
<td>High-level constructs</td>
<td>Used for BFL flow control</td>
</tr>
<tr>
<td>Variable declarations</td>
<td>User may declare variables</td>
</tr>
<tr>
<td>Variable assignments</td>
<td>To store a value into a variable</td>
</tr>
<tr>
<td>If-then-else</td>
<td>Conditional execution of BFL statements</td>
</tr>
<tr>
<td>While</td>
<td>Conditional looping of BFL statements</td>
</tr>
<tr>
<td>BFL procedures</td>
<td>Parameterized procedures are available</td>
</tr>
</tbody>
</table>

Figure 2 depicts the basic operation of the BFM. The BFM is driven by the ROM, loaded with the user's compiled BFL sequence. A sequencer directs the flow of control in the model, and a bus state machine implements the actual bus protocol on the hardware interface. A timing/logic shell adds the delays on outputs and performs the setup/hold checks and interfaces to the actual H/W pin interface of the Pentium CPU. A software interface is used for special instrumentation capabilities to facilitate synchronization of the BFL sequence to external events in the system.

In order to handle exceptions, special provisions had to be made to allow users to specify bus cycles that should occur for each type of exception. This was done by allowing the user to specify Exception Service Routines (ESRs) in the BFL. A special case of ESRs involves the internal caches. Since the BFM does not model the internal caches of the Pentium CPU, we devised a method to determine what actions to take upon, for instance, an external snoop cycle. This was also handled by having ESRs for such cases. For example, in the ESR for external snoop cycles, the user specifies whether the snoop is a hit or miss and, furthermore, if it's a hit to a modified line. In the latter case the BFM schedules a writeback cycle to occur.
The user has control over when each command is scheduled to occur, allowing emulation of internal bus cycle requests. For example, the BFM may or may not respond with a new address and cycle specification in response to a next address request for address pipeline mode, depending on whether an internal request is pending or not. This capability also allows the user to specify idle clocks to emulate the idle state of the Pentium CPU bus.
The user can specify a series of parameters for the basic BFL commands in order to create the various types of cycles that the BFM executes. Figure 3 shows an example BFL command sequence that provides a view of the usage of parameters. The special BFL commands allow the user to specify special cycles in the BFL sequence. The general commands are non-bus cycle generating commands that are used to set flags in the BFM's processor model (e.g., enable/disable interrupts) or to simply restart the BFL sequence. High-level commands are useful for programming algorithmic bus cycle sequences and structuring a test in a modular fashion. Instrumentation commands are useful when performing multiprocessor based system simulation.

```
-- Basic Read/Writes
READ(ADDR => x#F00000C0#);
WRITE(ADDR => x#F00000C8#, DATA => x#CAFEDEAD0000001#);

-- Locked Operation
READ(ADDR => x#F00000D0#, LOCK);
WRITE(ADDR => x#F00000D0#);

-- I/O Cycle
READ(ADDR => x#100#, IO);
```

**Figure 3** Simple BFL Source Code Example

**Development of the BFM**

In the development of the BFM, our first step was to carefully study the specification and micro-architecture spec and, in particular, the bus cycle protocols and bus cycle state machine. Since the BFM models all bus pins, a detailed understanding of each pin's function and timing was required. We needed a method for the user to specify each type of bus cycle. We accomplished this by dividing up the various unique types of bus cycles into equivalent BFL commands. Most bus cycle types occur in various "flavors" (e.g., a READ cycle can occur to memory or I/O); thus, most BFL commands have a parameter list allowing the user to select the flavor desired. In order to ensure that the planned BFM developments matched the customers' requirements, we developed an External Model Specification (EMS), reviewed it with customers, and revised it based on feedback. Contained within the EMS are detailed descriptions of the BFM and BFL along with the theory of operation and usage of the model for system design.

The BFM does not require extensive time to prepare stimulus that cause the desired bus cycles to occur. This is good from a hardware verification standpoint where the user is typically interested only in the bus-level interface to the Pentium CPU. Such a model may be developed much faster than a full functional model and therefore may be available some time before silicon arrives. This allows customers to debug their hardware in a software environment prior to committing to real hardware.
On the other hand, such a model is limited in what it may do. For example, customers interested in software verification or performance evaluation will not be able to use such a model. This type of model must also be used carefully. Since it does not emulate instructions, the user must be careful to write meaningful bus cycle sequences.

**Coding Methodology**

An abbreviated portion of some of the BFM data structures used for the BFL opcodes, that hold the various types and attributes for each Pentium processor bus cycle type, are shown below:

```plaintext
type bflOpcode is (undefinedOpcode, readOpcode, writeOpcode, idleOpcode, printfOpcode, control_flagsOpcode);
variable opcodeType: bflOpcode;

type busCycleOpcode is
   record
      opcodeType: bflOpcode;
      mfoBar: bit;
      addressValue: bit_vector(31 downto 0);
      dataValue: bit_vector(63 downto 0);
      idleClocks: integer;
   end record;
variable nextCycle: busCycleOpcode;
```

The BFM calls a procedure that uses TEXTIO to load an ascii file of the BFL opcodes into its internal ROM, and the bus state machine calls the following procedure when a Pentium processor internal bus request is generated. The rate of request generation is a function of the bus protocol and the users' BFL.

```plaintext
procedure getNextBflCommand
(
   variable bflRomAddress: inout integer; -- BFL instruction pointer
   variable bflOpcodeRom: in bflRom(0 to bcrOmHiAddr); -- BFL opcode array
   variable controlFlags: inout controlFlag; -- Record of BFM flags
   variable opcode: inout busCycleOpcode -- BFL opcode record
) is
begin -- Fetch and decode the next BFL instruction
decodeBusCycleOpcode (bflRomAddress, bflOpcodeRom, opcode);
bflRomAddress := bflRomAddress + 1; -- Increment the BFL instruction pointer
loop -- Loop until a bus command is found
   case opcode.opcodeType is
      when printfOpcode =>
         executePrintf (opcode); -- Non-bus command: execute immediately
      when control_flagsOpcode =>
         executeControlFlags (opcode, controlFlags); -- Non-bus command: execute immediately
      when others =>
         exit; -- Bus command: return command in variable ‘opcode’, and return control to bus state machine
   end case;
   -- Just executed non-bus command, so fetch and decode the next BFL instruction
   decodeBusCycleOpcode (bflRomAddress, bflOpcodeRom, opcode);
bflRomAddress := bflRomAddress + 1; -- Increment the BFL instruction pointer
end loop;
end getNextBflCommand;
```
The following code fragment shows how the basic bus state machine of the Pentium processor was modelled in the BFM.

type busStates is (T1, T2, T3, T4);
...
Bus_State_Machine: process
  variable currentState: busStates;
begin
  wait until CLK = '1';
  -- Signal 'bflRequestSig' indicates that a bus cycle request is being made (when = '1')
  if (bflRequestSig = '1') and (opcode.opcodeType = undefinedOpcode) then
    getNextBflCommand (bflRomAddress, bflOpcodeRom, controlFlags, opcode);
  end if;
  case currentState is
    when T1 =>
      sampleT1Inputs (...); -- Sample inputs required to determine state transition
      ... perform "current state" actions which occur on the rising edge of the clock immediately prior to the next state transition, for current cycle type (e.g., sample any relevant inputs for current cycle type. For example, in a data transfer state, sample the databus if currently in a READ cycle).
      ... determine next state, and perform appropriate "next state" actions which occur immediately after the state transition, for current cycle type (e.g., if next state is a data transfer state, and current cycle is a WRITE, drive the databus)...
      ... equate the 'currentState' variable to the next state
        (e.g., 'currentState:= T2;')
    when T2 =>
      ...
    when T3 =>
      ...
    ... etc ...
  end case;
end process;

AC Timing Methodology

The requirements for providing timing in the BFM are derived from the EIA-567 standard for VHDL Commercial Component Models[4] and also from specific needs of customers. They include very basic requirements such as propagation delays selected based on minimum, typical, and maximum specifications and setup/hold checks, in order to validate system-level timing. Model options are provided in order to enable certain checks and options.

The implementation uses a timing shell concept, that surrounds the BFM core. With this approach, the BFM core does not provide delays directly to the outputs but rather uses concurrent procedure calls in the timing shell to provide these delays. Timing checks that are dependent on internal states are invoked by the BFM core using timing check procedures provided in a Timing Package. Continuously executing timing checks can be invoked directly from the timing shell. Information that needs to be instance specific is provided using VHDL generics. This includes selection flags, operating constraints, and output delays. Information that is not instance specific is provided in the Timing Package.
Validation Methodology

The validation methodology is one of successive levels of testing starting from a basic test that is generated by hand, followed by a detailed trace based validation against the internal RTL simulator. We utilized an internal Pentium processor chipset design project as an alpha site. Focusing on the BFM, several methods were employed. Manual tests will check out gross-level functionality. We augmented the manual test suite with a second test suite derived from the Bus-Unit test plan utilizing the power of the iBFC tool and BFL. A special tool was developed to extract BFL from the internal Intel simulation environment running an RTL description of the Pentium CPU executing assembly based diagnostics. The resultant BFL was executed against the BFM along with RTL external pin stimulus, and the responses were used and verified the BFM to be clock-for-clock accurate with the RTL model of the Pentium CPU. This process resulted in delivery of a high quality model to our beta sites.

Future Plans

We are working on extending the capabilities of our models to include more detailed modeling of the Pentium CPU’s internal caches. We are also investigating ways of linking our C-based instruction set and performance simulation technology to VHDL in order to provide a software model that runs fast enough to be of use in a system simulation environment and meets the needs of a broader range of system design activities for Intel’s ever-more-powerful and complex microprocessor products.

Conclusions

In conclusion, we have presented a method and example of the use of VHDL for development of a bus functional model for the Pentium Processor. The model has proven to be a very rewarding experience to Intel’s customers by providing them an accurate, timely, and simulatable behavioral representation of the Pentium processor bus protocol. Furthermore, the model was very effective at ensuring the customers’ ability to design and validate Pentium processor based platforms and systems. We also outlined the integration of the cache models with the BFM, touched on the validation methodology, and provided a glimpse of our future plans.

Acknowledgments

We owe a special thanks to our customers who helped us define the model and provided excellent beta testing feedback. Also, to all those special vendors out there that put up with our complaints about their simulation tools: thanks for fixing bugs and providing great support!

We also acknowledge the guidance and great support of Joe Montgomery, the godfather and champion of system-level models at Intel. We also would like to thank the DT-SL and CFG-TTM teams for all the hard work leading to and during the alpha testing.

Notice: Trademarks mentioned in this paper are the property of their respective owners.
References


