

# Teaching Computer Architecture using VHDL

Peter J. Ashenden  
Department of Computer Science  
The University of Adelaide, SA 5005  
Australia  
petera@cs.adelaide.edu.au

## Abstract

This paper describes a project undertaken by students in a computer architecture course at the University of Adelaide. The project involves using VHDL to model a computer system at the behavioral and register transfer levels, and making measurements during simulation to get quantitative data relevant to a computer architect. The project brings to light a number of positive aspects of VHDL when used for modelling at high levels of abstraction in this kind of environment.

## Introduction

Computer architecture is often defined as the design of the interface between the software and the hardware of a computer system. A successful computer architect must understand the requirements of a computer system, both hardware and software, and also have a good appreciation of alternative hardware organizations and their cost and benefits. Hence the topic of computer architecture is a common ground between the disciplines of Computer Science and Electrical Engineering.

Much of computer architecture involves evaluating alternatives, and one of the most important tools for this is simulation. Simulation allows a designer to measure behavior and performance of a computer system without having to prototype it. The resultant cost savings and faster turn-around mean that more alternatives can be explored, leading to a more optimal product and reduced time to market.

Evaluating performance of a computer system at a high level of abstraction requires a quantitative model of performance. The model suggested by Hennessy and Patterson in [1] uses execution time of the computer running a program as the measure of performance, as follows:

$$\text{Execution time} = \text{Instruction count} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

Thus we can evaluate the performance of a computer system by simulating a high level model running some benchmark programs, and measuring the instruction counts and

represent the program counter (PC), instruction register (IR) and other special registers of the CPU, and an array of bit vectors to represent the general purpose register (GPR) file. The body of the process firstly resets all of the internal state and the output signals, then waits until the reset signal into the CPU becomes inactive. The process then enters a loop to fetch the next instruction from memory, decode it, and dispatch to the appropriate code section to execute it. This loop is exited when the reset signal becomes active again. When this happens, the process simply repeats from the top, to handle the reset condition. The loop body fetches an instruction by calling a procedure within a separate bus transaction package, which handles the protocol sequencing for accessing memory. The bit vector returned is stored in the IR. Decoding the instruction involves extracting the opcode, register numbers and any displacements from the IR. The opcode is then used as the selector in a case statement, each arm of which interprets a particular instruction. Because the CPU is a RISC design, decoding and executing instructions is relatively simple.

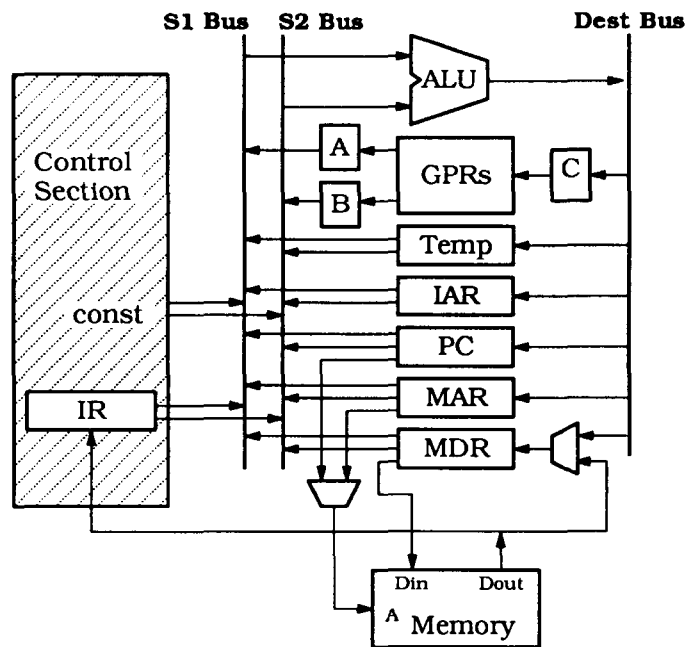
The students' task in the first stage of the project is to add instrumentation to the behavioral model of the CPU to measure dynamic instruction frequencies as the model runs a number of benchmark programs. This is best done by adding an array of counters, indexed by opcodes, to the instruction set interpreter process. Each time a new instruction is decoded, the appropriate array element is incremented, as is a count of the total number of instructions executed. Thus the relative frequencies of execution of each instruction and of the different classes of instruction can be calculated upon completion of program execution.

The students exercise their revised model by compiling a collection of different styles of programs from the Stanford small benchmark suite (viz. bubble, intmm, perm, queens, quick and towers). They use a version of the gcc C compiler ported to generate DLX assembler, and a simple assembler which generates a memory image file in the format used by the VHDL memory model. The students then run the simulations using the MINT VHDL simulator from Synthesia, and collect the statistics on instruction set usage to include and analyze in a written report. This work ties in closely with the theoretical work covered in lectures, in which the usage of the instruction set by different compilers and different kinds of programs is discussed.

## **Stage 2: Register Transfer Level Model**

In the second stage of the project, the CPU model is replaced with a register transfer level model. I supply a structural model of the datapath (see Figure 3), and simple behavioral models for each of the components it uses. The students' task is to develop a behavioral model for the controller to sequence operation of the datapath elements to fetch and interpret DLX instructions. A simple timing model is used for this datapath. Each of the registers is a transparent latch with tri-state drivers onto the source buses (S1 and S2). Datapath elements have a single propagation delay parameter, set by a generic constant in a configuration file when the component is elaborated. All data in the datapath is modelled using bit vector types. During phase 1 of the clock, source operands are enabled onto the S1 and S2 buses, and the ALU input transparent latches are enabled. The ALU is combinatorial, and produces a result on the Dest bus after a propagation delay. During phase 2, the ALU input latches are disabled, storing the operands, and the destination register is enabled to store the result.

The aim of this stage is to develop the students' understanding of datapath operations and sequencing. They must demonstrate that their controller causes correct fetching, decoding and execution of a significant subset of DLX instructions, including a number of ALU operation instructions, loads, stores and branches.



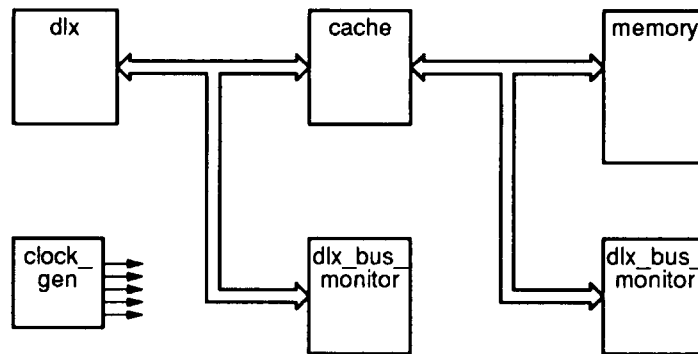
**Figure 3.** Register transfer level organization of the DLX CPU.

### Animated Version of Stage 2 Model

As part of the second stage of the project, I prepared an animated version of the register transfer level model, which displays each of the datapath elements in an X window. This version uses the X interface library provided as part of the MINT simulator. A screen dump showing the CPU in the instruction fetch state is shown in Figure 4. Each of the components in the datapath graphically depicts its state (eg, stored data, enabled inputs or outputs, ALU operation). As the simulation progresses, each element updates the graphical display of its state. The intention of this animated model is to help students understand how data path operations should be sequenced, and to provide a debugging aid for their controllers. Initially they run the model with my sample controller included in the configuration, and follow the sequence of datapath operations by clicking on the mouse button to advance the simulation one clock phase at a time. Then they substitute their controller, and check that it causes the same (or an equivalent) sequence of datapath operations. The animated version makes it easier to visualize model operation and detect errors.

The way the animated model works is as follows. The MINT X interface package provides a number of procedures for managing windows and drawing graphical objects. Firstly, an initialization procedure establishes a world coordinate system, creates a window and returns a handle to it. Next, an event polling procedure allows a model to query whether an X event (key, mouse button or exposure) has occurred, and to get event parameters such as window location, button number, etc. The remaining procedures are for drawing lines, polygons and text. The animated model includes another package to encapsulate animation management, including broadcasting of X event information to components in the model. It declares two global signals, one of a type that provides information about the animation window, and the other of a type that provides information about X events. It includes an initialization procedure that must be called by a process in the main architecture body of the model. The procedure creates the animation window and broadcasts win-





**Figure 5.** DLX computer system with cache memory included.

ments the semantics of the entity modifies the value of this state signal whenever its internal state changes as a result of VHDL events on input signals. The animation process in the architecture body is also sensitive to the state signal, and updates the graphical depiction of the instance whenever a transaction occurs. Thus the state signal provides a mechanism for cleanly separating the code that deals with model semantics from the code that deals with animation. The animation code is further separated by encapsulating it in a separate package for each architecture body, thus simplifying the animation process in the architecture body itself.

### Stage 3: Cache Memory Model

The third stage of the project requires the students to add a cache memory between the CPU and the main memory, as shown in Figure 5. The cache operates transparently to the CPU, so the same behavioral models of the CPU and main memory are used. The difference is that the simulated access time of the cache is significantly less than that of the main memory. The students must write a behavioral model of the cache, and measure the improvement in execution time over a system without the cache. The aim of this exercise is to reinforce the students' understanding of the details of cache organization and operation, and the effects on systems performance.

Figure 6 shows the entity declaration for the cache, provided to the students as a specification of the interface to which their model must conform. The cache is parameterized with respect to total size, line size, associativity (direct mapped, set associative, etc.) and write strategy (write through or copy back). The students' models must correctly handle any reasonable combination of actual values for these parameters, as actual values are not allocated to each group until after they have developed their behavioral model. Each group simulates a cache with a particular combination of parameter values, running the same benchmarks as used in stage 1. In their reports, they use their measurements along with others I provide, to determine how different cache organizations affect execution time.

### Experiences with Using VHDL

Apart from its value as a teaching exercise, this project provides an opportunity to evaluate VHDL as a language for architectural level modelling. In particular, the design of the animated model in stage 2 of the project provides some interesting insights. Firstly, the package facility in the language enabled an object oriented approach to the design, and that

```

entity cache is
  generic (cache_size : positive;           — in bytes, power of 2
          line_size : positive;           — in bytes, power of 2
          associativity : positive;       — 1 = direct mapped
          write_strategy : strategy_type; — write_through or copy_back
          Tpd_clk_out : Time);           — clock to output delay
  port (phi1, phi2 : in bit;              — 2-phase clocks
        reset : in bit;                  — synchronous reset input
        — connections to CPU
        cpu_enable : in bit;              — starts memory cycle
        cpu_width : in mem_width;        — byte/halfword/word indicator
        cpu_write : in bit;              — selects read or write cycle
        cpu_ready : out bit;             — status from memory system
        cpu_a : in dlx_address;          — address bus output
        cpu_d : inout dlx_word_bus bus; — bidirectional data bus
        — connections to memory
        mem_enable : out bit;            — starts memory cycle
        mem_width : out mem_width;      — byte/halfword/word indicator
        mem_write : out bit;            — selects read or write cycle
        mem_burst : out bit;            — tell memory to burst txfer
        mem_ready : in bit;             — status from memory system
        mem_a : out dlx_address;        — address bus output
        mem_d : inout dlx_word_bus bus); — bidirectional data bus
end cache;

```

**Figure 6.** Entity declaration for the cache model.

allowed clean separation of the code that dealt with graphics and animation from the code that modelled device behavior. There was one package that encapsulated the code for animation management, and a separate package for each animated architecture body that handled drawing and graphical state update. Secondly, the generic constant mechanism also proved useful for supporting the object oriented design of this suite. Each instance of a component was provided with different values for its generic constants, uniquely identifying the instance and specifying the location in the screen window where it should be drawn. Each component instance is responsible for updating its graphical rendition and its state when X events are broadcast or when the behavioral model updates the component's internal state. Thus there is no centralized code that needs to know about behavior of components, making the suite easier to develop, evolve and maintain.

One of the problems that arose in this model, and that will arise in any model that must use a graphical user interface and respond to user initiated events, is that there is a conflict between the two control regimes. On the one hand, the simulator kernel controls execution of VHDL code in response to changes of values on signals. Typically it may allow a model to execute for some fixed amount of simulation time, or up to a breakpoint, and then suspend execution. On the other hand, a user interface manager, such as the X windows system, needs to invoke code in response to user events such as keystrokes, button presses or mouse movements. The difficulty is that VHDL provides no way of specifying a code object as a parameter to be passed to a user interface manager. The model described in this paper circumvented these difficulties by introducing global signals to broadcast X event information, thus translating from one control domain to the other. While this approach works in this case, where the delta delays involved in responding to X events do not cause problems, it would not be appropriate for models that use delta delays as part of the behavioral description. A difficulty that was experienced in this model was unreliability of the simulator when the X interface was used in this manner. I suspect that the conflict between the X and VHDL control regimes may be a factor causing crashes, though this remains unverified.

At a more general level, there are a number of other features of VHDL that made the project feasible. Firstly, VHDL's programming language basis allows it to bridge the gap between computer science and engineering practice. (This is particularly so at the University of Adelaide, where students learn Ada as their main programming language.) Since much of behavioral modelling is similar in nature to conventional programming, the students' learning curve was less steep. They were better able to concentrate on coming to grips with structural modelling, circuit timing and sequencing, and discrete event simulation.

Secondly, the features to support "programming in the large" were very important for dealing with the complexities of the computer system models. Use of packages for abstract data types and utility operations, and of configurations for joining different parts together in various combinations, made the project significantly easier to manage.

Thirdly, the ability to use simple abstract types and a simple timing model made the language suitable for this style of architectural modelling, in which detailed electrical characteristics of signals are not relevant. The models use bits, bit vectors and enumerations for most data types, and avoid having to represent high-impedance values and different driving strengths on bus drivers by simply disconnecting drivers using null transactions. Most components use a single propagation delay generic parameter to describe timing, with the actual values supplied in configuration files. This, in combination with a simple clocking scheme, made it possible to avoid dealing with inertial delay timing semantics, since no transactions are ever pre-empted by a new signal assignment.

## **Conclusion**

One of the claims often made about VHDL is that it supports a spectrum of modelling, from low level (switch and gate) to high level (architectural). While there are numerous discussions that deal with its use at lower levels, the project described in this paper shows that VHDL can successfully be used at the architectural level, where some simplifications are made in the modelling of data values and timing. This demonstrates that the primitives provided by the language are adequate for modelling at this level. While there is scope for extending the set of language features to facilitate some programming tasks, care must be taken not to overburden the language, since it is, primarily, a hardware description language.

The project has evolved to the structure described here over a number of offerings of the course. Feedback from many students shows that they find it challenging but rewarding. They find that the "hands on" work is useful in reinforcing the theoretical aspects of the course. Indeed, for some students, the problem is to stop them doing too much work on the project. The students enrolled in the course come from two different streams. One cohort, from a computer science background, have previously had little direct contact with digital hardware, and find the project helps them understand the details of hardware sequencing and control. The other cohort, from an electrical engineering background, have had significant experience with digital system, but find the integration of this experience with more abstract computer system ideas useful. They also benefit from the experience in dealing with significantly larger software suites than they have previously had to manage.

## **Reference**

- [1] J. L. Hennessy & D. A. Patterson, *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann Publishers (San Mateo, CA) 1990.