Abstract

There is a need for a design methodology that allows the representation and simulation of a design at various levels of abstraction and interpretation. The Single Path Design Methodology presented in this paper is a possible solution to this difficult problem. The fundamental idea behind the methodology is to use one simulation language (VHDL, Version 1076) for all phases of design. The VHDL framework allows for iterative step-wise refinement of a model. A performance (uninterpreted) model can be refined to an RTL description without changing modeling environments or completely re-writing the models. As an example, the Performance Modeling phase of the Single Path Design Methodology is applied to the WM machine, a superscalar computer architecture.

1. Introduction

The size and complexity of present day digital systems has brought about the need for new design methodologies and tools that support these methodologies. The design process and tools should enable the designer to gain information about the system at every stage of its development, from initial concept to final physical design. In the top-down approach to design, a behavioral specification derived from user requirements is decomposed into less complex functional modules. These modules are then transformed into a structural form before realization in a given technology. In the bottom-up approach structural and physical modules are combined to realize a given behavioral specification. Design tools are used to verify the functionality and performance of systems through modeling and simulation.

Computer systems are modeled at different levels of detail or abstraction: circuit level, gate level, register-transfer level and system level [1]. System-level models are developed to analyze the system from a performance standpoint, and represent only those elements of the system pertinent to the performance issue of concern. The performance modeling stage of the design process has assumed increasing importance because of the large sizes of present day designs. As systems become more complex, architectural decisions made at the system level may have a significant effect on the performance of the overall system. In fact, with device density figures approaching their physical limits, it will be high-level architectural decisions that distinguish good designs from poor ones.

Performance modeling alone is not sufficient to ensure good designs. The system should be modeled and simulated at all levels of description to ensure that all specifications pertaining to the specific level of description are being met. The CAD community has developed numerous methodologies and tools for the modeling and simulation of systems. Unfortunately, these methodologies and tools work only for specific phases of the design process. Several languages and tools must be used to model and simulate a design across its several levels of representation. The translation of a representation into several languages is a time consuming and error prone process. Hence, there is a need for a design methodology and tool that allows the representation and simulation of a design at various levels of abstraction and interpretation in the same simulation environment.

Abstraction and interpretation [2, 3] are two concepts important to this paper. The level of interpretation is determined by the absence or presence
of functions that map input values to output values. There are only two levels of interpretation: uninterpreted and interpreted. The level of abstraction on the other hand, is determined by the relationship between the amount of information carried in the model and the amount of information carried in rules external to the system being modeled. It is essential that any computer-aided design methodology be capable of describing and evaluating a specified hierarchical configuration, while retaining autonomy at each level and preserving the algorithmic content during the required transformation between levels.

2. Related Work

A majority of the existing design and modeling environments support only performance level modeling. Included among these are ADAS[4], RESQ[5] and SES/Workbench[6]. ADAS uses Petri nets to model concurrent systems. However, Petri nets suffer from an exponential growth in size as system complexity increases. In addition, the user has to write code in different languages to transcend levels of representation. RESQ and SES/Workbench are queuing theory based modeling environments. Both of these systems allow efficient high-level modeling of systems but do not allow models of varying levels of interpretation to coexist. In addition, it is not clear how flexible the two systems are in describing, in a convenient manner, the low-level details of complex control structures found in computer systems. A survey of multi-level design systems can be found in [7]. Some of the early attempts at providing multi-level modeling capabilities include LALSD, LOGOS and SARA [7]. These systems require the user to use different languages to represent different levels of the design. The SABLE [7] system provided multi-level modeling capabilities but did not attempt to address performance aspects of the design process.

3. Single Path Design Methodology

As discussed previously, there is a need for a design methodology and tool that allows the representation and simulation of a design at various levels of abstraction and interpretation in the same simulation environment. The methodology presented in this paper is a solution for this difficult problem. The fundamental idea of the Single Path Design Methodology is to use one simulation language for all phases of design. VHDL (IEEE, Version 1076) [8] provides a basis from which the single path design environment is feasible. VHDL is capable of supporting models spanning varying levels of abstraction and interpretation. Both algorithmic and structural models can be designed and simulated using the VHDL language.

At the uninterpreted end of the modeling spectrum, models are constructed to predict the performance of a proposed architecture. Performance modeling essentially consists of the analysis of data flow without interpretation of function. Petri-nets and queueing networks are examples of performance models. Information flow between elements is modeled by flow of 'tokens.' In the single path design methodology models based on this view are referred to as Uninterpreted Models [9]. At the other end of the spectrum are the detailed functional models that typically map directly into hardware. Models in which functionality is of key interest and the information flowing between blocks is fully specified are referred to as Interpreted Models. The technique of simulating both uninterpreted and interpreted subsections in a model is called Hybrid Modeling [10]. The ability to include both uninterpreted and interpreted modules in the same model provides the framework for the single path design methodology [2]. This hybrid simulation capability is lacking in the tools and methodologies currently provided by the CAD community.

In essence, a good design methodology should support a combination of top-down and bottom-up design strategies. Designers initially use a top-down approach and then use their knowledge of low-level information to improve the higher level models for the purpose of more accurate performance characterization. Thus, there is a need for a facility for utilizing low-level information on the behavior of a functional model to enhance the accuracy of the corresponding higher level model [10]. This feature is referred to as back-annotation or up-annotation. The single path design environment offers this capability. Hybrid models allow the back-annotation process both to proceed in a simple straightforward manner. Low-level interpreted modules can replace uninterpreted components of the 'performance model'. The hybrid model is simulated until a predetermined confidence interval has been satisfied. At this point, the interpreted component can be replaced with a delay node whose probability density function was determined by the hybrid simulations. Thus, the timing of model has been improved without adding additional complexity to the model.

Uninterpreted models (performance models) of systems are constructed by the use of a library of primitive building blocks that can be used to model the control and data flow in computer systems [11]. Each primitive building block is written in VHDL, Version 1076 and hence a simulatable description of the system is available. Tokens are used to represent flow of
Petri nets. There are three main types of building blocks: 1) control modules which are used to control the flow of tokens through the model, 2) delay modules are used to model delays and 3) color modules are used as a mechanism to manipulate information of any kind on the tokens. For a complete description of the building blocks and the tools refer to [11] and [12].

A graphic front-end, ADEPT [12], allows the designer to construct and simulate models of systems without writing any VHDL. Simulation results are used to extract performance statistics such as utilization, throughput, or latency. The flexibility provided by the primitive modules and the front end allows modeling of highly complex architectures. An important aspect of the methodology to note is that with the use of the primitive modules, the user is not required to write any VHDL code. However, the tool allows any user-written VHDL code to be easily integrated into the system.

Hybrid modeling and simulation capability is added to the environment through the use of the Interpretation interface module. The interface allows the translation of tokens of information to real input for the interpreted modules and the back annotation of information from the interpreted module into the uninterpreted domain. For a detailed description of the technique, refer to [10].

The single path design methodology is only practical if the mechanisms required to transition between the different levels of abstraction do not inhibit the usefulness of each distinct model. Performance measures must be obtainable in a timely manner from an uninterpreted model. Likewise, functional verification must be possible for a hybrid/interpreted model. The single path design methodology achieves this goal. As will be shown in following sections, the building block approach supported by the methodology is a natural mechanism for describing data/control structures typically found in modern computer architectures. More importantly, this approach easily lends itself to the refinement process. The methodology does not use any artifacts in the uninterpreted domain that would be difficult to translate into real hardware once the uninterpreted model has been refined. For instance, the modeling methodology does not use global variables even though many performance modeling tools allow them in the model description. The rationale for this decision is that there are not any global variables in real hardware. Instead, the modeler must define the appropriate control signals or encapsulate the information inside the token (see section 6). The token handshaking protocol is a standard four stage handshaking (request, acknowledge, send, receive) often used in hardware designs. Finally, a direct mapping between the building blocks and hardware components often exists.

The remainder of this paper describes the use and applicability of the single path design methodology to model and study the performance of a novel computer architecture, the WM [12]. The example shows the use of the primitive modules in modeling complex control structures found in computer systems and the techniques for extracting performance figures. The method used to drive the model with real software is also described.

4. The WM Computer Architecture Overview

The WM computer Architecture is designed to be a high performance architecture with the hardware complexity similar to RISC machines and approximately the same density produced for CISC machines [13, 14]. WM is a byte addressable load/store architecture which achieves high performance by executing multiple operations in a single clock cycle through the use of separate asynchronous functional units. Therefore, the WM can be classified as what is termed a superscalar architecture [15].

The basic structure is shown in Figure 1. The four primary units are the Instruction Fetch Unit, the Integer Execution Unit, the Floating Execution Unit and the Vector Execution Unit. These units allow concurrency between operations that have no interdependencies. Although each functional unit must process instructions sequentially, the units may operate in parallel with respect to each other. To promote maximal concurrency, information transmitted between functional units is buffered by first-in first-out queues (FIFOs).

The Instruction Fetch Unit (IFU) is responsible for fetching instructions from memory and enqueuing the instructions into instruction FIFOs that reside between the IFU and the three execution units. These instructions are dequeued from the instruction FIFOs and executed by the execution units. In addition, the IFU contains a special execution unit. The special execution unit is responsible for executing the flow control instructions (jumps) and some 'special' instructions such as the synchronization and state manipulation instructions.

The Integer Execution Unit (IEU) is responsible for executing integer instructions, i.e. those instructions that refer to the integer registers. In addition, the IEU performs the address computations required for the load/store instructions, regardless of the type of data involved in the memory operation. All computations are
performed by a pair of pipelined ALUs available within the IEU. The destination for a result computed by the ALUs depends on the type of the computation performed. Load/store addresses computed by the ALUs are enqueued into appropriate address FIFOs. Data for store operations in the IEU are enqueued into the output data FIFO. The architecture also provides a register file with 32 registers, each 32 bits wide. Results of computations with destinations other than the output (address or data) FIFOs are sent to the register file. Incoming data resulting from integer load operations are enqueued into the input data FIFO.

The Floating Execution Unit (FEU) is responsible for the execution of operations involving floating point data. The structure of the FEU is very similar to that of the IEU, except for the fact that no address computations are performed in the FEU. Data for store operations in the FEU are sent to the output data FIFO, and data resulting from floating point load operations are enqueued into the input data FIFO.

The Vector Execution Unit (VEU) is responsible for executing vector instructions. The unit is slightly different from the other two execution units in that vector instructions perform a single operation. Therefore, only one ALU is provided in the VEU. The register file consists of vector registers, where each vector register is capable of holding a vector of elements.

5. Results

This section of the paper describes part of the uninterpreted model of the WM and the use of the model for generating performance statistics. The methodology used for driving the model is also explained.

5.1 The WM Model Overview

The chief motivation for developing high level performance models of computer architectures is to allow the designer to quickly determine the effect of design decisions on system performance. The model should be simple to allow very short turnaround times in the development and analysis phases but must also be complex enough to capture the novel features of the architecture that are expected to boost performance. This ideology of a simple yet sophisticated model was strictly followed throughout the development of the Uninterpreted Model (UM) of the WM computer architecture. As a result, no instructions are actually executed by the model. Token flow is used to duplicate the flow of information that would occur if instructions were to be executed by the model.

The salient features of the methodology are brought out in the description of a portion of the Integer Execution Unit presented in the following sections. The IEU, like the entire model, is modeled hierarchically. Before describing the model, a brief explanation of the token structure used is presented.

5.1.1 UM Token Structure

At the highest level of abstraction, the presence or absence of a token on a signal, represents the presence or absence of information. However, the token can be "colored" in order to permit control information to be distributed around the model. This control information is used for making routing decisions at several points in the model. A colored token is implemented in VHDL as a record. The data structure is shown in Figure 2.

```
type TOKEN is
  record
    STATUS: HANDSHAKE;
    TAG1: INTEGER;
    TAG2: INTEGER;
    ...
  end record

Figure 2: Token Type definition
```

The tag fields are used for carrying color information. The status field is used internally for a token-transfer protocol between the modules. The specific use of the color field will be discussed in a later section.
5.1.2 The Integer Execution Unit (IEU)

The IEU receives tokens from the Instruction Fetch Execution Unit and 'executes' the integer type instructions. The pipelined execution of the integer type instructions involves the following stages:

1) Dequeue an instruction from the input FIFO
2) Fetch the operands
3) Execute the inner ALU operation
4) Execute the outer ALU operation
5) Store the result in the appropriate register/FIFO
6) If an address was generated, send the value to memory
7) Enqueue the condition codes in the CC FIFO

Figure 3 shows the top-level block diagram of the IEU.

![Block Diagram of the IEU](image)

The IEU is activated when a data token (instruction) is present in the IEU instruction FIFO and the IEU execution unit is "ready" to accept a new token. In this context, "ready" implies that the first of the two pipelined ALUs has finished its arithmetic/logical operation. There is no buffering between the IEU instruction FIFO and the ALUs.

When the IEU receives the data token, the token is split into three identical data tokens. Each token is sent to a multiplexer designed using the low-level uninterpreted modeling primitives. The ALU inputs can come from any one of three possible sources: the register file, the R0 data input FIFO or the R1 data input FIFO. The function of the multiplexer is straightforward. A data token arrives at the input of the multiplexer, and without any simulation time delay, a data token is placed on one of three data request lines. These request lines are connected to the register file, R0 and R1 data FIFOs. The request line selected is determined by reading the value on a token color field. In this model, three color fields are being used to explicitly indicate where the source operands for the three ALU inputs originate.

For this uninterpreted version of the architecture, it is important to note how the queues and register files operate. When a request is made to any queue or register, the request is always serviced. It is assumed data is always present in the queues. When evaluating short traces, or even individual instructions, deadlock situations can occur because the system is not properly initialized. This situation prevents performance measures from being collected. By not matching requests with data, deadlock situations are avoided. It is accepted that by modeling in this manner, some timing accuracy is being lost. However, future revisions of the model will include all of the control/data dependencies as more details are added to the design.

The ALUs have been modeled as simple fixed delays. The delay of the ALU is based on the worst case operation. By coding the specific ALU operation on a token color field, a data dependent delay module could be used as opposed to the simple fixed delay.

The token, upon exiting the second ALU, is routed to one of four output FIFOs. A decode unit, similar to the multiplexer, is responsible for this function. The destination address is encoded onto a token color field. When the token arrives at the decode unit, the destination value is read from the token and placed onto a control line. The control line, in conjunction with a decider, module controls the data token routing.

A detailed description of the UM primitive modules, control and data signals is best presented by way of an example. A portion of the ALU multiplexor structure is shown in Figure 4.

![IEU Multiplexor Structure](image)

This multiplexor determines where the first source
operand for the ALU is located. When a token arrives at the input of the read color module, the value of a specified color fields is assigned to the control signal (control signals are dashed lines). The control signal is connected to the decider nodes. Each decider node compares the value on the control signal against the base value. If the two values are equal, the data token is placed on the first output line, else the data token is placed on the second output line. Thus, an integer value of “1” indicates that the data must be retrieved from the register file, while a “2” directs the request to the RO input FIFO. Other structures are modeled in a similar manner.

6. Driving the WM Model

Typically, performance or uninterpreted models are probabilistically driven[6]. Static program analysis can be performed on the benchmark programs to derive reasonable probability density functions (PDFs) of the instruction mix. The PDFs are used to define the control/data flow of the model. However, this approach is a poor solution to the modeling problem. The designer must be careful to build a model that will not reach a deadlock state. Performance measures are collected over thousands of simulated computer cycles. If the model were to deadlock, performance analysis would be difficult, if not impossible. The deadlock problem often forces the designer to make gross approximations. Secondly, PDFs do not capture either the temporal or spatial locality exhibited by programs. The performance predictions obtained from a performance model may vary greatly with subsequent behavioral models that operate instruction traces that characterize the computers workload. It would be beneficial if the designer could analyze the architecture's performance for a given workload. The architecture could then be optimized for the predicted workload.

The best performance measures will be obtained if the WM uninterpreted model is driven with a real software trace. How can this be accomplished? As stated previously, one of the benefits of building an uninterpreted model is that a preliminary analysis can be accomplished without having to write a behavioral description. Traditionally, models that operate on instruction traces require a behavioral description. Uninterpreted computer architecture models are driven by instruction traces by using the color fields of the control/data tokens.

WM instruction traces are reduced into subsets of integers that are used to color the control/data tokens of the UM model. The color information is then used by the nodes in the model to control the information flow of the model. For example, Table 1 illustrates a mapping between WM instructions and the color field values. The current version of the uninterpreted WM model requires six pieces of information; the instruction type, the location of the three operands for the pipelined ALUs, the destination address, and the memory control value (read/write).

<table>
<thead>
<tr>
<th>WM Instr</th>
<th>Token Color Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD W</td>
<td>1 4 2 2 1 0</td>
</tr>
<tr>
<td>r11:= 7</td>
<td>1 4 2 2 2 0</td>
</tr>
</tbody>
</table>

Table 1: Trace to Color Field Mapping

In the example, the first instruction, LD W, loads the constant W into the RO input FIFO. Since, load instructions are executed by the IEU, the instruction type is assigned the value “1”. In reality, when the address is computed in the pipelined ALU’s, the expression expands into (addr(W) + 0 + 0). The address of W is encoded in the instruction word while the zero constants are found in a register (register 31). This information is represented by the encoding 4,2,2. Finally, the destination of this instruction is the IEU output address FIFO, and this information is coded as a “1” to indicate the IEU. The second instruction, load integer register r11 with the value 7, is similar to the first instruction. The only difference is in the destination field. For this instruction, the result is written back to the register file as opposed to an output queue.

A “C” program has been developed that automates the process of extracting the information and encoding it. The input to the program is a WM assembly trace file, and the output is a file containing a set of integers. There is one set of integers for each instruction in the trace.

The tokens are colored in the model by using a low-level modeling primitive called “File IO”. The coloring of tokens takes place during simulation, not prior to simulation. Each time a token arrives at the input of the file_io module, a line from the reduced trace file is read, and the values are assigned to the color fields of the token. The token is immediately placed on the output line without any simulation delay occurring. For the instruction r11:= 7, tag1 is assigned 1 and tag2 is assigned the integer 2.
6.1 Simulation Results

This section describes both the results of a performance analysis of the WM architecture and the instrumentation that has been developed to collect performance statistics of an uninterpreted model.

6.1.1 Instrumentation

Performance statistics are extracted from the signal activity observed during simulation. The signal activity itself is collected in three files by the use of special primitive modules called Monitors [lo]. These modules can be connected across other WM elements (or collection of elements) to monitor their performance. The usage is similar to that of a voltmeter connected across an electrical device. Defining a special monitor module separates the data collection code from the functional code. Thus, the user need not be concerned about data collection code when a new primitive is defined. Multiple monitors may be used in a design.

Four major classes of statistics can be recovered. These are: (1) element utilization, (2) signal latencies, (3) queue length variations and (4) signal arrival and completion rates. These measurements all have the standard meanings. Statistics are calculated by the post-processing of data collected by the monitors during simulation. In the example presented in this paper, monitors are connected across the IEU instruction FIFO and across the Special Execution unit.

6.1.2 WM Performance Analysis

An initial performance evaluation of the WM architecture was conducted using the uninterpreted (VHDL-based) model previously described. For this study, the instruction trace that drove the model was the insertion sort algorithm operating on a data set of size ten. All the instructions in this trace are either executed by the integer execution unit or the special execution unit of the IFU. The performance of the architecture is directly dependent on the length of the IEU instruction queue. The deeper the IEU instruction queue, the greater the probability that the integer execution unit is operating concurrently with the special execution unit. When the IEU instruction queue fills up, the IFU stalls and the special execution unit may be unnecessarily idle. The performance of the architecture can be improved by satisfying the following conditions.

1. The utilization of the special execution unit should be maximized.
2. The token arrival rate is large enough so that if possible, the IEU instruction queue is never empty. This situation results in the maximum IEU utilization.

However, to satisfy the requirements, the IEU instruction queue should not be set to some arbitrarily large size. An excessive queue length wastes resources.

![Figure 5: WM Performance Measures](image)

Figure 5.a illustrates how the utilization of the special execution unit varies with respect to IEU instruction queue length. IEU utilization reaches its maximum achievable value when the queue length is greater or equal to sixteen. A queue length of sixteen results in the ability of the IEU to consume almost 0.1 instructions per unit time (Figure 5.b). This rate ensures that the IEU instruction queue is never empty. Finally, by fixing the IEU instruction queue size at sixteen, the utilization of the IEU execution unit can be monitored over the entire simulation run. The simulation results are presented in Figure 6.

This example based on the insertion sort illustrates a typical performance analysis possible using the methodology described. Before finalizing any parameters, multiple runs with different traces would be used. The parameters would be chosen only after averaging the different results.

7. Conclusions

This paper has presented a novel methodology for modeling complex computer architectures in a VHDL environment. System design starts at high levels
of abstraction with the system control and data flow being modeled by high level primitives. A hardware language description of the model is generated. This description is used to simulate the model and extract performance statistics like percent utilization, queue lengths and such. The WM architecture is an example of a possible architecture evaluation scenario that might be carried out very early in the design process. The process is iterative and is carried out until satisfactory performance is obtained. The methodology suitably addresses the problem of driving the model with real software in order to obtain accurate performance measures. The UM primitives have been found to be powerful enough for the task of modeling the complex control structures of the WM machine. A flexible scheme for generating performance figures has been introduced. The fact that the model and the modeling methodology are based on VHDL allows the inclusion of modules of differing levels of interpretation in the same model. We feel that with the increasing size and complexity of present day digital systems, the use of such modeling methodologies is essential for minimizing design times and permits extensive architectural evaluation without having committed any portion of the design to silicon.

8. Future Work

The goal of the refinement process is to be able to get better performance predictions in a systematic manner as the system is iteratively refined. The performance estimates for an architecture should become more accurate as the critical path is refined and the timing models are improved. However, several issues relating to hybrid modeling and the use of the interpretation interface in the context of a large computer system models have not been resolved. For example, when in the design process should the transition from the uninterpreted domain to the hybrid domain occur? What data should be placed on the token color fields and what information should be supplied by the interpretation interface?

These research questions and others, are currently being addressed by continuing the refinement of the WM computer architecture. Presently, the focus of the refinement process is the IEU pipeline. To accurately access the timing of the pipeline, all of the pipeline conflicts and inter-dependencies must be known. This requires more information than what is presently available in the model. Interpreted descriptions for several of the different stages of the pipeline are being written in VHDL. These interpreted descriptions will be simulated as part of hybrid model. It is expected that the hybrid model will provide better performance estimates for the WM architecture. This timing improvement may result from two different aspects of the hybrid model. First, the interpreted description probably will be more accurate than the corresponding uninterpreted description. Secondly, the interpreted description may add information to the model which may be utilized in other parts of the model.

9. References


