Implementation-Independent Model of an Instruction Set Architecture Using VHDL

Maximo H. Salinas  Barry W. Johnson  James H. Aylor

Center for Semicustom Integrated Systems
Department of Electrical Engineering
University of Virginia
Charlottesville, Virginia 22903-2442

Abstract

This paper presents a new methodology which allows the creation of implementation-independent functional models of systems. The methodology uses the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) which leads to the possibility of creating a unified design environment supporting system modeling at various stages of the design process. The motivation for the methodology was to assist in the analysis and specification of the WM computer architecture. An implementation-independent model of the WM architecture is provided which is intended as a first step in the development of an implementation. Additionally, architectural performance measures can be extracted from simulations using the model. The model can also serve as an architectural specification for the WM computer.

1. Introduction

The need for studying and evaluating computer architectures at the instruction set level rather than at the implementation level was recognized in the work on a new computer architecture being developed at the University of Virginia (UVa) [5]. The proliferation of new architectures whose instruction set design is closely coupled to the specific implementation, makes an architectural comparison among the various alternatives very difficult.

To study the dynamic behavior of an instruction set architecture in the execution of actual test programs, an Implementation-Independent (II) modeling methodology [4] has been developed. II models are functional models in that an instruction stream provided to them is executed, and the calculated results are available as output. The II modeling methodology uses the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) [3] which supports system modeling at various stages of the design process [1]. In addition, the methodology provides the ability to describe system architectures at a functional level without implementation details.

The II modeling methodology incorporates a novel communications and synchronization mechanism to coordinate the concurrent processes modeling the parallelisms in an architecture. The handshake mechanism takes advantage of VHDL's user-defined data types, functions, procedures, and Bus Resolution Functions (BRFs) to embed an arbitration scheme used to control communications and synchronization between VHDL processes. Additionally, the need for global system clocks is obviated by the synchronization provided with the handshake mechanism.

The II modeling methodology arose from work on the analysis and specification of the WM computer architecture [5]. The WM architecture uses distinct weakly coupled execution units for integer, floating point, vector, and control operations. To study the potential performance advantages of system architecture issues independent of particular implementation issues, an architectural model of the WM computer was desired.

2. Implementation-Independent Modeling

Implementation-Independent (II) models implement the logical structure defined by a system's architecture. The logical structure is the interconnection network required to represent all the parallelisms permitted by the architecture. In other words, the logical structure indicates the functional decomposition which preserves all possible parallelisms in the execution of an architecture's instruction set. The physical structure is the interconnection network of a system's implementation. The physical structure designated by a particular implementation will not typically duplicate the system's logical structure because constraints usually require sacrificing parallelism...
to reduce the number of implementation-dependent elements. II models are constructed by creating behavioral descriptions of the elements constituting the system's logical structure. For example, the logical structure of a computer architecture may contain an instruction fetch unit, an integer/logical execution unit, a floating point execution unit, some address and data buffers, and so forth.

Information flows between the elements of an II model through a VHDL user-defined signal type and Bus Resolution Function (BRF) which are generalized versions of those defined by Hady [2]. The use of the signal type Token and the VHDL BRF Protocol, allows an embedded handshake so that no implementation handshake mechanism is implied. The definition of Token is shown in Figure 1 and consists of a record containing two items: (1) a status field of type Handshake which can take on the values Inactive_Sink, Active_Source, Active_Sink, and Inactive_Source, and (2) a user-defined color field which is a record containing any signal types that are required by the particular model.

The definition of Protocol is shown in Figure 2, and a graphical description of its operation is supplied in Figure 3. Protocol operates on a vector of tokens and uses the status field of each token to determine which is to be passed as the resolved token. Protocol first scans the vector of tokens searching for the presence of Source-type and Sink-type tokens attempting to create a token pair consisting of one Source-type and one Sink-type token. The arbitration mechanism illustrated in Figure 3 is then used to select which of the two tokens from the pair is to be the output of Protocol; the status at the head of each arrow indicates the higher priority. The search in creating the token pair is for active-status tokens. However, if no active-status token of a particular type is found but an inactive-status token of that type is found, the inactive-status token will be selected to be used in token pair. In the case where tokens of a particular type are not found, the token of the present type selected for the token pair is selected as the output of Protocol.

### 2.1. Examples of Token Usage

The usage of tokens is simplified by user-defined VHDL functions and procedures which manipulate the status and color fields. The use of functions and procedures permits the modeler to change the operation or structure of a token without requiring that existing VHDL code be modified. An example of a status manipulation function used in the II modeling methodology is the Place-Token(T) function which changes the status of token T to Active_Source. An example of a function manipulating the token color fields (XX is used to designate an arbitrary color field) is the Change-Token-XX(T, Info) function which changes the XX color field of T to Info.

Lastly, procedures and functions may be defined to manipulate a number of color fields simultaneously along with possibly the token’s status field to simplify the use of frequently used token manipulation instruction sequences.

```vhdl
function Protocol (Input : Token_vector) return Token is
variable Source_token : Token := Def_Source_token;
variable Sink_token : Token := Def_Sink_token;
variable I : Integer;
variable No_Source, No_Sink : Boolean := TRUE;
begin
-- Search through the input token vector
for I in Input'low to Input'high loop
  if (Input(I).status = Active_source) then
    Source_token := Input(I);
  elsif (Input(I).status = Active_sink) then
    Sink_token := Input(I);
  elsif (Input(I).status = Inactive_source) then
    No_Source := FALSE;
  elsif (Input(I).status = Inactive_sink) then
    No_Sink := FALSE;
  end if;
end loop;
-- Return the value of the signal, that satisfies the
-- handshaking protocol, based on the active field of the
-- source and sink tokens
If No_Source then return Sink_token;
else return Source_token;
end Protocol;
```

Figure 1: Definition of Token

```
type Color_Type is
  record
    Direction : Access_type;
    Size : Access_Size;
    Condition : MVBbool;
    Instruction : bit32;
    Address : bit32;
    Data : bit32;
  end record;
constant Def_Colors : Color_Type :=
  (Read,Word,Invalid, x'00000000',x'00000000',
   x'00000000',x'00000000');
end record;
```

Figure 2: Protocol Bus Resolution Function
As an example, the `Place_Token_Write(T,Address,Data)` procedure manipulates the user-defined color fields named `Direction`, `Address`, and `Data` (the value of `Direction` is implied by the procedure used and does not appear as a parameter passed to the procedure) by changing the `Direction` color field of `T` to `Write`, its `Address` color field to `Address`, its `Data` color field to `Data`, and its status field to `Active Source`.

As a simple example, Figure 4 shows the sequence of VHDL instructions required to perform an access on a signal with one `source`-type token generator and one `sink`-type token generator. The example performs a read operation. The embedded handshake mechanism available by using tokens and Protocol synchronizes the two VHDL processes, one at the master and one at the slave.

### 2.2. Tokens and Hierarchy

The use of VHDL BRFs in hierarchical design may require careful attention to the order in which multiple calls to the BRF occur. Such cases arise when a signal contains multiple drivers at one level of the hierarchy and at least one of these elements contains multiple subelements and/or processes which also produce multiple drivers for the signal. If the BRF is associative, there is no danger of resolving the signal incorrectly. The Protocol BRF used with tokens, however, is not generally associative. For example, in the case in which there are two possible `source`-type token generators (masters) and a single `sink`-type token generator (slave) with a master in one element and the second master and the slave generator in another element, the handshake between the master in the first element and the slave in the second element will not succeed if the two drivers in the second element are passed through Protocol to produce an intermediate result. Two examples are shown in Figure 5 in which a slave within an element that also contains a master acknowledges an access initiated by an external master.
In the case in which an element contains both masters and slaves for a token which must be resolved in a higher level in the hierarchy, the element must provide a separate driver at its port declaration for each of the internal drivers of the token. By passing the token from each driver through its own port up through the model hierarchy, Protocol can be called only once at the highest appropriate hierarchical level to produce the correctly resolved token.

However, in the special case in which an element contains multiple token drivers which are either all masters or all slaves, Protocol can be called to produce an immediately resolved token for the element so that the element requires only a single port. This condition is possible because the search through the token vector in Protocol correctly identifies the absence of a source-type or a sink-type token in the vector so that the active status token (if any) of the type present in the vector will be the resolved token. This facility both reduces the number of signals present in element port declarations and allows the hiding of implementation-specific details as II elements are replaced by implementation dependent elements, such as in a memory system consisting of distinct memory banks.

### 2.3. Synchronization and Timing

Since the clocking characteristics of a system are issues determined by the implementation and realization of the system, II models do not include clocked elements. Rather, when required, synchronization of system elements is achieved by using tokens and VHDL wait statements. The example of Figure 4 illustrates that synchronization between a master and a slave takes place during communication between them. Tokens, therefore, may serve as synchronizers between elements as well as carriers of information.

II models may include delay modeling. Typically, delays designated to the elements in an II model are assigned to characterize some particular implementation technology. For example, a model may define a floating point division to take ten time units and an integer add to take one time unit. This type of delay modeling is necessary to study system performance at the architectural level keeping in mind that actual systems will be constrained by implementation characteristics which must be considered to accurately predict performance. In fact, the assignment of element delays is required in order to accurately observe the parallelisms available in the architecture. To understand the reason for this statement, VHDL’s timing constrasts must be analyzed [3].

VHDL’s timing model uses two types of delays, delta delays and simulation delays. Delta delays are controlled by the VHDL simulator while simulation delays are controlled by the VHDL programmer. Additionally, VHDL supports two types of storage primitives, variables and signals. An assignment to a variable results in the variable assuming its new value immediately. An assignment to a signal, however, is actually the scheduling of a future value to be stored in the particular signal. The delay between a signal assignment and the signal assuming the new value is either provided by the programmer in simulation units or is defaulted to one delta delay.

Communication between VHDL elements is performed by connecting signals between them. Therefore, any communication between elements incurs delta delays and/or simulation delays. Consequently, simulation delays must be assigned to II elements in order to separate the delays in the performance of the elements’ functions from the delays in inter-element communications. For the purposes of calculating system delays and studying parallelisms, only advances in simulation time are considered so that events which occur in the same simulation time but possibly different delta times are considered concurrent and indicate system parallelisms.

### 3. WM Architecture Overview

The WM architecture manual [5] identifies certain functional components of a WM computer and describes their operations. The architecture is characterized by a load/store memory access mechanism utilizing data first-in-first-out buffers (FIFOs), register to register operations, and distinct large register sets for integer/logical, floating point, and vector operations. The instruction set is designed to allow asynchronous operation between a number of execution units.

The architecture explicitly prescribes four execution units, the Instruction Fetch Unit (IFU), the Integer/Logical Execution Unit (IEU), the Floating Point Execution Unit (FEU), and the Vector Execution Unit (VEU). These units generally operate asynchronously with the IFU enqueuing instructions into IEU, FEU, and VEU instruction FIFOs, while each execution unit dequeues instructions from its FIFO as rapidly as it is able. For ease of understanding, the Vector Execution Unit has not been included in the model presented and will not be discussed further. In addition, separate units (referred to as Stream Control Units, or SCUs) are required to perform streaming and simple memory accesses. Additional components required by a WM computer include data FIFOs used in accessing memory and a pair of condition code FIFOs used for conditional branching.

The architecture defines six instruction classes: Integer/Logical instructions, Load/Store instructions, Floating Point instructions, Vector instructions, Control Instructions, and Special Instructions. The instructions within each class are performed by one of the execution units to allow concurrent execution of instructions from different classes. This concurrence may be asynchronous due to the instruction and data FIFOs defined by the WM architecture.

Integer/Logical instructions are executed by the IEU. Each integer instruction operates on three source
operands which may denote either five-bit literals or data from either a data FIFO or the integer/logical register file. The instruction includes two operators so that the instruction format is:

\[ \text{Int Dest} := (\text{Src1 Op1 Src2}) \text{ Op2 Src3} \]

where: (1) Dest is either an integer/logical data FIFO or a register in the IEU register file in which the result of the operation will be written, (2) Src1, Src2, and Src3 each represent either a five-bit literal or a data item read from either an integer/logical data FIFO or a register in the IEU register file, and (3) Op1 and Op2 each represent an integer/logical operation.

Because the use of two operators, Op1 and Op2, within a single instruction is characteristic of several instruction classes in the WM architecture, the architecture semantics require adherence to a data dependency rule [5]. Specifically, the result of an instruction is not available as an operand of the inner operation of the following instruction for the same execution unit. The value of an inner operand is specifically independent of the effect of the previous instruction.

The relational operations in Op1 and Op2 above are the only integer operations to set condition codes. In addition, they pass their left operand as a result to implement compare-and-operate instructions efficiently. The condition codes generated by relational instructions are enqueued in an integer condition code FIFO and are dequeued by the IFU. This mechanism frequently permits asynchronous operation between the IFU executing conditional branch instructions and the execution unit generating the condition codes.

Load/Store instructions, which are also executed by the IEU, perform memory address calculations and result in memory access requests to read or write memory using data FIFOs in the register files. These FIFOs are referred to as r0 and f0 in the IEU and FEU register sets, respectively. The format for load/store instructions is similar to the integer/logical instruction format and has the following form:

\[ \text{LSOp Dest} := (\text{Src1 Op1 Src2}) \text{ Op2 Src3} \]

where: (1) LSOp is a load/store operator specifying the type of memory access to be performed, (2) Dest is either an integer/logical data FIFO or a register in the integer/logical register file in which the result of the operation will be written, (3) Src2 and Src3 each represent either a five-bit literal or a data item read from either an integer/logical data FIFO or a register in the integer/logical register file, (4) Src1 must refer to a data item read from either an integer/logical data FIFO or a register in the integer/logical register file, and (5) Op1 and Op2 each represent a load/store arithmetic operation. The result of a load/store operation is both written in Dest and used as the address in the requested memory access.

Floating point instructions are executed by the FEU. The format for these instructions again resembles that of integer instructions in that they operate on three

source operands, from either data FIFOs or the FEU register file, and two operators:

\[ \text{Flt Dest} := (\text{Src1 Op1 Src2}) \text{ Op2 Src3} \]

where: (1) Dest is either a floating point data FIFO or a register in the floating point register file in which the result of the operation will be written, (2) Src1, Src2, and Src3 each represent a data item read from either a floating point data FIFO or a register in the floating point register file, and (3) Op1 and Op2 each represent a floating point operation. Relational instructions are included to generate condition codes written to a floating point condition code FIFO read by the IFU.

Control instructions are executed by the IFU. The instruction set includes an unconditional jump and conditional jumps based on the contents of the IEU and FEU condition code FIFOs. Alternatively, conditional branches may depend on the state of a particular streaming operation. Some instructions include stream start and stop instructions, context manipulation instructions, data transfers and conversions between the IEU and FEU register files, and other miscellaneous instructions.

4. WM Computer Model and Simulations

A block diagram of the II model of the WM computer (excluding the Vector Execution Unit) is shown in Figure 6. The figure also shows the flow of data between modules in the model. The connections illustrated represent tokens which are necessarily bidirectional due to the embedded handshake mechanism described previously. Because control information flows in both directions across each connection, the direction of the arrows denotes the direction of data flow. Note that in most cases, an address used in an access is considered control information. For example, the connection between IFU and memory points towards the IFU because that indicates the direction of data flow on that connection although an address is transmitted from the IFU to the memory module during the access request.

The WM computer model allows simulation delays to be included as FIFO access delays for each FIFO in the model, ALU delays for each ALU operation, and memory access delays. These delay parameters allow the II model to characterize particular implementation technologies by assigning appropriately normalized delays to the above modules. In fact, as described earlier, modeler-specified simulation delays must be provided to differentiate between advances in simulation time incurred in the execution of WM programs and advances in delta time incurred in the passing of information among the various modules. For example, the architectural performance impact of a technology for which an integer division requires ten times the delay of an integer addition can be studied by designating the appropriate delays to the ALUs in the model.
that this particular program allows a maximum of nine RISC-like instructions to be executed simultaneously.

5. Summary

Because of the concurrency permitted by the WM architecture by its various execution units, the module utilization achieved in executing complex programs may not be easily predicted. Therefore, the II modeling methodology was developed to model the WM computer architecture at a functional level. An II model permits a designer to simulate the execution of complex programs and extract the desired module utilization information for a particular implementation technology.

Although not specified explicitly by the WM architecture, a memory module description is required to simulate a WM computer. The II memory module provides a separate port for each of the possible memory masters in the WM architecture. These memory masters include the IFU for instruction fetches, the IEU Load/Store service module, the FEU Load/Store service module, and eight SCUs. The II model of the WM computer is programmable in that a WM microcode program is actually executed by the model.

The WM program used to create the simulation results presented here performs the multiplication of a complex vector, with real and imaginary parts, by a real constant. The program is designed such that all eight Stream Control Units (SCUs) are employed to read the necessary data from memory and to write the results to memory. Figure 7 shows simulation results derived from running the program. Although Figure 7 is intended to highlight the parallelism achieved in the execution of the programs, performance measures such as latency and percentage utilization of components may be easily extracted from this format by implementation designers. A black square indicates that the selected module was utilized at the indicated time. The IFU receives a black square only at those times in which it executes a branch instruction since non-branching instruction fetches do not contribute to the parallel execution of WM instructions. It is seen

6. Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract number N00014-89-J1699 and by the Virginia Center for Innovative Technology. In addition, we would like to thank the WM group at UVa with special recognition to Bill Wulf and Anita Jones.

7. References