01 Apr 1991

An Architecture Level Simulation Methodology

Paul D. Stigall
Missouri University of Science and Technology, tigall@mst.edu

Ram Huggahalli

Recommended Citation

P. D. Stigall and R. Huggahalli, "An Architecture Level Simulation Methodology," *ANSS 1991 - Proceedings of the 24th Annual Symposium on Simulation*, pp. 240 - 253, Institute of Electrical and Electronics Engineers, Apr 1991.
The definitive version is available at https://doi.org/10.1109/simsym.1991.151511

An Architecture Level Simulation Methodology

Paul D. Stigall
Ram Huggahalli

Department of Electrical Engineering
University of Missouri-Rolla
Rolla MO 65401

Abstract

Using the Architecture Design and Assessment System (ADAS), the processor level architecture of an example computer system is first represented as a directed graph. Then, a method of simulating instruction execution as a sequence of data transfers between the nodes of the graph is presented. The simulation methodology provides flexibility in observing the architecture dynamically at the processor level. An example application for functional verification is discussed. Development of techniques to convert programs into node sequences and, to assign appropriate delays to the nodes is necessary to further enhance the applicability of the methodology. Functional verification and performance estimation through this approach can instigate early design trade-offs and reduce system development costs.

1. Introduction

1.1. Graphical Modeling and Simulation

Computer systems are modeled [1] extensively with the aid of graphical simulation tools. The development of workstations and concepts such as distributed [2] and concurrent simulation have further increased their applicability. A number of workstation based CAD tools are available for graphical representation and simulation. These CAD tools may be general purpose tools like SIMSCRIPT and GPSS which can be used to study any representable system. No framework for computer systems modeling is provided by these software and therefore design approaches vary widely. Contrastingly, 'several Computer Hardware Description Languages' (CHDL's) [3] have been recently developed for rigorous formal descriptions of computer systems hardware. The interest in CHDL's is due to two main reasons. First, technological developments have lead to systems having high circuit level as well as organizational complexity. Design approaches therefore must be standardized and highly structured. Secondly, formalization of design descriptions facilitates the correct comprehension of these systems. CHDL's thus offer a comparitively rigid framework suitable only for the design of computer systems hardware. CAD tools consisting of graphical user interfaces (GUI's) for structural descriptions to enhance the effectiveness of CHDL's are receiving tremendous attention these days.

1.2. Computer Architecture Simulation

Computer architecture means the structure of the modules as they are organized in a computer system. Three levels of computer systems hardware [4] may be identified as,

Processor level
Register level
Gate level

The methodology presented in this paper is directly applicable to processor level modeling and extendable to the register level.

1.2.1. Existing Methods of Simulation

Many informal methods have been used for the description and simulation of architectural designs. Owing to the lack of a structured approach, these methods addressed only sections of the architecture that had significant impact on the architecture's performance. For example, memory system design is a critical issue that has been studied by simulating the interaction between the processor and the memory system alone. The processor can send tokens to the memory modules and the tokens can return to the processor from the memory modules after a delay. If a cache miss or a page fault occurs, the memory latency can be made to increase greatly. Simple time-based or event-based

0272-4715/91/0000/0240$01.00 © 1991 IEEE

simulations can provide important insights on memory hierarchy and its potential impact on the architecture's performance. To consider the architecture's performance in the presence of an I/O controller, which contends for a global memory thereby imposing a restriction on processor-memory communications, further programming effort is required to extend the model's capabilities. No general scheme for simulating data transfers between all the processor level architectural modules has been proposed primarily because of the complexity involved in describing the entire architecture. Thus, such methods fail to provide an overall view of the dynamic activity of the architecture.

Architecture Description Languages (ADL's) [3] are a sub-class of CHDL's for the formal description of architectures. A prime example is the VHSIC Hardware Description Language (VHDL - IEEE standard 1076), a US government supported and currently the most widely used ADL. In CAD and simulation tools that are ADL-based, it is observed that applications attain complexity very early, even in top-down design procedures by considering both control flow along with data flow within an architecture.

The Architecture Design and Assessment System (ADAS - RTI) [5, 6, 7, 8, 9] though classified as a CASE tool known for its software/hardware integration capabilities, can be used for hardware description in a manner similar to VHDL. ADAS offers a graphical user interface for the structural description of computer systems hardware. Component behavior can be described using ADA, C, ISPS (Instruction Set Processor Specification) or HELIX HDL. A processor level simulation methodology has been built on ADAS using an example computer architecture as a test environment. The methodology is to act as a platform for design evaluation on general purpose tools and ADL's. It facilitates early design trade-offs by making, the extraction of the maximum possible information on the computer system at the highest levels of abstraction, feasible.

1.2.2. Need for a Simulation Methodology

For the architecture analysis of a computer system with a design description and simulation tool like ADAS, a methodology must be developed. The objective of the methodology is to answer questions such as,

(i). To what depth must the behavior of the entities in the architecture be described ? For example, is it necessary to write a program that behaves exactly like a CPU ?

(ii). How do we model the interaction between a set of modules in the architecture ? How do we relay data through a specific sequence of modules ?

(iii). How do we test our architecture for a certain algorithm or program, or set of program statements ?

(iv). How can actual delays be assigned to the modules ?

In the following sections, one such approach built for ADAS but applicable to all similar software, is presented using an example computer architecture. The approach can be extended in a straightforward manner for simulating a variety of computer architectures.

1.2.3. Outline of the Approach

First, the given processor level architecture is represented as an ADAS directed graph model. Then, to simulate data flow during instruction execution, it is proposed that instructions be mapped to node sequences. A node sequence gives the sequence of delay elements that are involved in the execution of an instruction and the order in which the delays account for the elapsed time. The token that is exchanged between the nodes of the model is given a data structure consisting of a node sequence segment and an information segment. The node sequence segment is used to route the token through the nodes specified in the sequence and the information segment is used to affect any specific treatment of the token. The designer can enter any desired token values in a text file which is accessed for execution in the model. Using combinations of input stimuli the designer may then explore various features of the architecture and, propose and test protocols for dealing with issues such as bus arbitration and hierarchical memory systems. The functional aspects of the model may then be refined to accommodate these protocols. The model can also be refined to lower and finer levels.

Once the model is fully functional, the architecture's performance can be estimated by running application programs that are translated into token values. Simple schemes for such translation and assignment of proper delays can be developed for obtaining reasonable performance estimates. Figure 1 shows the scope of this work in the overall design evaluation methodology. The methodology can be extended to a variety of architectures and to explore any processor level design issue. It can be implemented on general purpose simulation tools as well as ADL's such as VHDL.

Although there is no limit to the refinement of the model to lower levels, the methodology is more suited for only preliminary evaluation of the architecture, facilitating early design trade-offs.

2. ADAS

The Architecture Design and Assessment System is a set of computer aided design tools, for the architecture level synthesis and analysis of software algorithms and their hardware implementations. During the construction of an ADAS model, attributes are specified to characterize the individual elements of the model. In this process, a database that all ADAS facilities use is created. These facilities include a directed graph editor called EDIGRAF and a functional simulator called CSIMGEN. All CAD tools which also support simulation can be expected to have similar features for structural and behavioral descriptions.

In designing a system, either the top-down or the bottom-up design methodology can be used. ADAS supports both design methodologies by allowing hierarchical design facilitites.

2.1. ADAS Design Approach

2.1.1. Directed Graphs

In ADAS, designs are represented by marked directed graphs, a sub-class of Petri nets [10, 11]. In an ADAS directed graph, nodes represent individual software operations or hardware functional elements and arcs represent data flow paths between the nodes. Each node has at least one inport or outport through which it is interfaced to other nodes. Nodes are interconnected by arcs from outports of source nodes to inports of sink nodes. Attributes are specified to assign special characteristics to nodes, inports, outports and arcs.

Data flow is simulated by passing discrete elements called 'tokens'. A token represents a unit of information. Each inport of a node is characterized by a *firing_threshold* attribute and a *token_consume_rate* attribute. The *firing_threshold* of an inport determines the minimum number of tokens that must be available on the associated arc, to enable the associated node to execute. If the threshold is satisfied, the node 'primes' and 'fires'. As the node primes, it removes as many

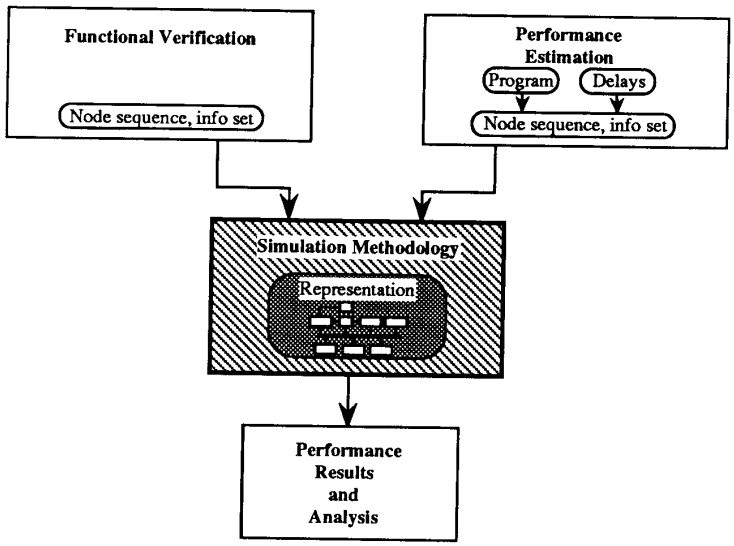

Figure 1. A Design Evaluation Methodology

tokens as specified by the *token_consume_rate* attribute value of the inport from the queue of tokens in the connected arc. When the node fires after the expiration of a delay specified in the *firing_delay* attribute, the node places as many tokens as specified by the *token_produce_rate* attribute at an outport. This process of firing is equivalent to the execution of software operations or hardware components.

2.1.2. Simulation of Directed Graphs.

Various directed graph simulation methods are provided in ADAS. Besides, a Petri net simulator for verifying the connectivity and proper attribute assignment in the graph, programs in C, ADA and hardware description languages such as HELIX and ISPS can be assigned to each ADAS node at the lowest level in the graph hierarchy for functional simulation. The *package_file_name* attribute of these nodes is assigned the name of the source code file of a program that executes as the nodes prime and fire. Once the graph is simulated, performance statistics can be generated through ADAS' in-built commands.

3. Example Computer Architecture

To develop a computer architecture simulation methodology with ADAS, the militarized reconfigurable processor (MRP) architecture [12] was used as a test environment. This architecture adheres to the von Neumann type of organization. Figure 2 shows the block diagram of the MRP.

3.1. Functional Architecture

The Central Processor (CP) and I/O Controller (IOC) modules form the core of the MRP's functional architecture. Both have their own microprogrammed control structure, are completely independent and asynchronous. They communicate with each other through a CP command instruction and an interrupt structure. A common bus structure serves as the data transfer medium between all major elements of the MRP. IOC's communicate with peripheral equipment and other computers through the IOC adapters. Memory references are made through memory interface modules which are connected directly to the bus.

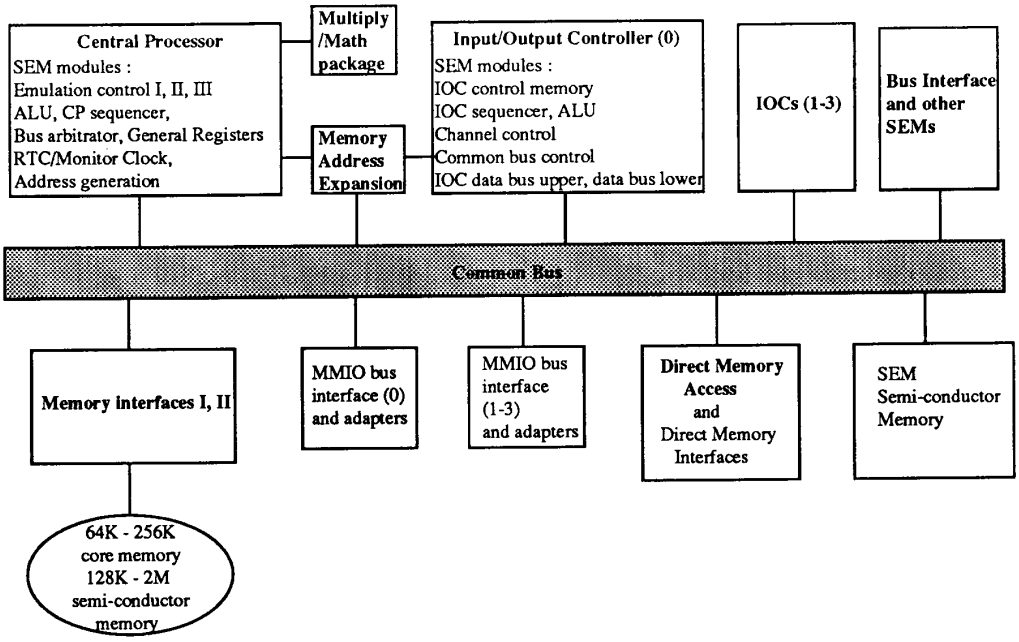

Figure 2. MRP Functional Architecture Block Diagram

The main memory is random-access 16-bit word magnetic core memory (non-volatile) or semi-conductor memory or a combination. Other modules include the Direct Memory Access (DMA), the bus interface (BusIF), Memory Address Expansion (MAE), the memory mapped I/O (MMIO), and the MATH co-processor. The DMA is an optional module that allows an external controller to read from and write into the main memory. Contention among CP, IOC and DMA is resolved by pre-assigning priorities to the three modules. The IOC has the highest priority followed by the CP.

3.2. Instructions

A variety of instructions are executed by the CP in coordination with the rest of the architecture. These instructions give us the sequence of data transfer and control between the processor level modules introduced in the previous section. As in most von Neumann architectures, the entire instruction processing sequence involves the following steps [13]:

1. Instruction fetch : A CP-Memory-CP process.
2. Instruction decode: A CP process.
3. Address generation: A CP process.
4. Operand fetch : A CP-Memory-CP process.
5. Instruction execution: A CP-'any other module'-CP process.
6. Operand store: A CP-Memory-CP process.
7. Update prog. counter: A CP process.

The description of instruction processing in terms of module interaction shows the dominance of the CP in a computer. All steps are initiated by the CP and terminated at the CP. It must be noted that the execution of many instructions does not include some of these steps. Instruction execution in step 5 need not involve any other processor level module.

IOC instructions also have a similar sequence, with the control always being returned to the IOC at the end of the sequence. However, the IOC instruction streams are invoked by the CP 'command instruction'. During the execution of the command instruction, in step 5, CP communicates with IOC.

4. ADAS Representation of the MRP Architecture

The first step in the development of the simulation methodology is the representation of the MRP functional architecture on ADAS. The functional block diagram of the MRP architecture shown in the previous section is constructed using EDIGRAF to obtain a single level hardware graph named 'uyk.hwg' (after AN/UYK-44 the technical name of the MRP). The 'uyk.hwg' alone represents the top level of the model (Figure 3). The 'sub-graph' feature has been exploited to expand each of the top-level nodes into separate second level hardware graphs. Thus, the model is a hierarchical one, comprising of the uyk.hwg and other sub-graphs.

The top layer graph consists of a total of 14 nodes of which 9 nodes are directly derived from the functional architecture diagram of the MRP. These nodes will be referred to as the 'main' nodes. A number of arcs in the graph have been made to overlap for closeness to the functional block diagram in appearance.

The In0, In1, Out0 and Out1 nodes act as interfaces to the peripherals and other external environment. The BA node resolves contention between the CP and IOC. All nodes that do not contain any sub-graphs and are mapped to corresponding C programs for functional simulation. Each sub-graph consists of a node to check the relevancy of incoming data and a delay node.

5. Simulation of the MRP Architecture

Given the directed graph of Figure 3, in order to functionally verify and obtain performance statistics for the graph, data flow through the nodes must be simulated. When instructions are processed, data is transferred between a number of modules of the computer architecture. At each module the data is manipulated and the results are used to affect further processing at other modules. To obtain statistics such as node utilization, number of times a node fires, busy times and latencies, actual information manipulation need not be performed for realistic results. These statistics depend on the order or sequence of nodes that the data flows through and the time delays at each of the nodes. A method to relay a token that represents data, through a specified sequence of nodes is now presented. A sequence of nodes can correspond to the execution of an instruction or any operation that can be interpreted in terms of processor level module communication. The CSIMGEN facility of ADAS has been used for this purpose. During the development of this method only unit delays have been used to be able to verify the correct sequencing of the token easily. Application of this type of simulation for performance analysis with actual delays is also discussed in section six.

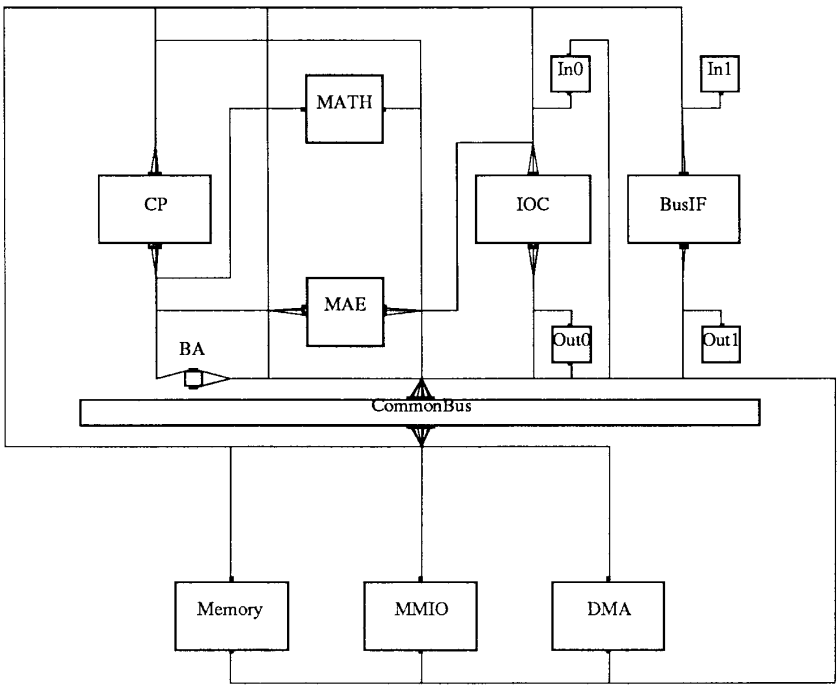

Figure 3. ADAS Representation of the MRP - uyk.hwg

5.1. The Methodology

5.1.1. Some Basic Assumptions

The methodology has been constructed with the assumptions that,

(i). the processing of an instruction always begins at the Central Processor and ends at the Central Processor. That is, the CP node is where processing is initiated and terminated,

(ii). the IOC's instruction processing can be invoked only by a CP command instruction and,

(iii). the MRP can be modeled as a data flow computer (i.e. control flow is not modeled explicitly).

These assumptions are valid for most instructions and do not restrict our model to a narrow class of instructions. Also, the assumptions are supported by the fact that the CP is the master of the MRP and that the CP needs to know whether a certain operation hasbeen completed or not, before it can initiate another operation. This is particularly so with a single bus connecting most of the modules of the MRP.

5.1.2. Assignment of Reference Numbers

In order to direct the token to a certain node from another node, it is convenient to first, assign all major nodes with reference numbers. Table I shows eight important nodes with their reference numbers.

Table I. Main nodes and their reference numbers

Node name	Reference number
CP	1
MATH	2
MAE	3
IOC	4
BusIF	5
DMA	6
MMIO	7
Memory	8

A node sequence now can be represented by a concatenation of these reference numbers. For example, the node sequence CP-MATH-CP will be '121'. An instruction fetch or an operand fetch, involves the node sequence CP-Memory-CP, which can be represented by '181'. This type of representation of node sequences is

exploited for the simulation of data flow through various modules in the MRP architecture.

5.1.3. The Token Structure

The token in functional simulation can have actual values which can be manipulated at the nodes. If the node is mapped to a program, the program uses the value of the token and performs the required function and if there is an outport to the node places a token with the resulting value at the outport. A tremendous amount of manipulative power is achieved with the attribution of a data structure to a token and the use of multiple inports and outports. The data structure, a token represents, has two segments,

1. The node sequence segment and
2. The information segment ('info' for short).

The node sequence segment of a token consists of a concatenation of node reference numbers and is used to relay the token through a specific sequence of nodes. The info segment can be used to indicate any special type of processing that must be performed during the simulation of the current node sequence. The use of the info segment provides additional flexibility to the model by allowing specific situations to be simulated. Examples of the use of the info segment are given in section six.

The most significant digit (msd) in the node sequence segment of the token structure is the reference number of the next node the token must be passed to. If the current node sequence is 'xyz' (where x, y and z are three different reference numbers), the token must be relayed through nodes x, y and z, in that order. When the token reaches node x, the msd is removed from the node sequence, leaving a current node sequence of 'yz'. Now, the token only has to go to node y and node z.

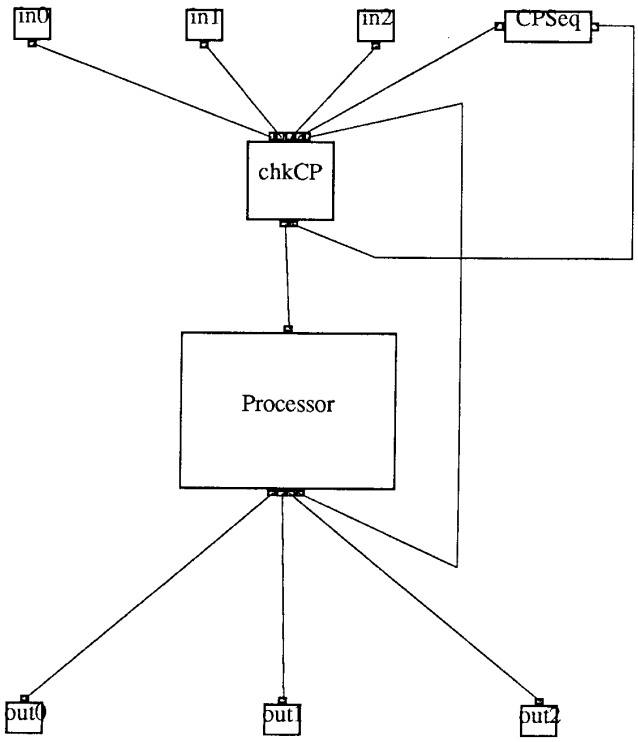

Figure 4. The Sub-graph of the CP Node - CP.hwg

5.2. Functions of Nodes

The sub-graphs of each of the main nodes contain the following types of nodes, classified based on their functions. The sub-graph of CP is shown in Figure 4.

5.2.1. 'chk' Nodes

These nodes (example: chkCP in Figure 4) perform the function of checking for the reference number of the parent node. The C programs mapped to these nodes observe the following algorithm:

```
if (any PortisActive(INPORT NO.))
/* if any inport is active */
{
        Extract msd of token's node sequence segment.
        if (msd == parent node's reference number)
        {
                Remove msd from node sequence.
                Place new node sequence with the info
                segment (a new token value)
                at the outport.
        }
        else DeActivatePort(OUTPORT NOS.)
        /* disallow data flow through outports */
}
```

5.2.2. 'Core' Nodes

All processing that contributes significantly to overall delay is represented by core nodes (example: Processor node in Figure 4). These nodes are the time delay causing elements in the model. All performance analysis results depend on the value of the firing_delay attributes of these nodes. The closer this value is to actual processing time the more accurate our results will be. However for the verification of the methodology, unit delays offer the most convenience.

CSIMGEN is an ADAS facility to simulate the model functionally. It uses the C language behavioral descriptions and produces an executebale functional simulation program called 'csim'.

Before beginning simulation, it must be ensured that valid node sequence and info segment pairs are entered in the CPSeq.dat file. Simulation results depend completely on the contents of this file. The CPSeq node in the sub-graph of the CP node (Figure 4) accesses each pair of sequence and info and assigns these values to a token's data structure. The token is then released for propagation through the architecture modules.

6. Application of the Methodology

A method of relaying tokens through specified node sequences has been described. An underlying assumption is that the processing of instructions can be represented by a series of token transfers between architecture modules that are involved in the executions. The assumption is valid in most cases since most instructions in an architecture's repertoire of instructions involve communication between at least two modules (typically CP and Memory) in the architecture. Also CP, as in all von Neumann structures is the master controller, implying that processing of most instructions begin and end at the processor. Two basic operations are included in the processing of most instructions, the instruction fetch and the execute cycles. It is these type of instructions that have been modeled in this manner. There are broadly two applications for the simulation methodology, functional verification and performance measurement. Functional verification implies the implementation and test of the protocols that control the execution of the architecture and the communication within the architecture. Performance measurement implies obtaining the architecture's performance for application programs by the use of actual component delays. The two applications are dealt with in the following sections.

6.1. Functional Verification

Communication at all architectural levels follow specific protocols. For example, the CP communicates with the IOC whenever there is I/O processing to be performed. It does so by executing a command instruction which invokes the IOC which then begins processing the I/O instructions. By observing the dynamic behavior of the system for different sequences, one can define such protocols for correct execution of the architecture. By changing the node sequences for consecutive simulation runs, bottleneck situations, contentions and other undesirable events can be observed and the proposed schemes can be verified and refined. The simulation methodology in its current state can be readily used for functional verification. The use of unit delays renders simplicity to this process. It is recommended that a formal verification methodology be used for proper utilization of the simulation methodology. We now present a detailed example of this application.

6.1.1. Bus Arbitration

The CP and IOC function completely independently and asynchronously, and communicate through the CP command instruction and an interrupt structure. The IOC relieves the CP of the peripheral communication burden. When an IO function needs to be performed , the main program initiates an I/O chain through the CP command instruction. The I/O operations depend on the stored program defined for the specified channel. Memory is accessed by both the IOC and CP on a time-shared basis, with the IOC having the priority. After the command instruction, the CP suspends its use of the bus until the IOC relinquishes it. This process of time-sharing the memory and other optional modules connected to the bus has been modeled in the uyk.hwg with the help of the BA (Bus Arbitration) node.

a. The BA node. CP and the IOC send tokens to the CommonBus node only through the BA node. The function of the BA node is to check for requests for the CommonBus node and resolve contention for the node by giving the IOC node the priority. The BA node consists of two 'core' nodes, the CPArb and the IOCArb which are interconnected to each other (see Figure 5). The two nodes are mapped to CPArb.c and the IOCArb.c C programs respectively. The algorithms followed by these two programs can be presented as follows,

CPArb
CPArb.c uses static variables for maintaining knowledge of current status of bus contention.

IO_wait = 1 if IOC is requesting use of the bus.
= otherwise.
CP_seq is a buffer to hold a CP originated node sequence.
CP_info is a buffer to hold the corresponding info segment.
CP_wait = 1 if CP is waiting to use the bus.
= 0 otherwise.

CPArb algorithm:
For the CPArb node to be active, either of the two inports must be active. Inport 0 is connected to the CP node, while inport 1 is connected to the IOCArb node. Once activated the following actions take place.

(i). If inport 1 is active, set IO_wait to the value of the info segment of the token at inport 1.
(ii). If inport 0 is active, and if the incoming sequence is new, and if IO_wait = 1, save the input node sequence, and info segments in CP_seq and CP_info and set CP_wait = 1. Report CP_wait to IOCArb. If the incoming sequence is a currently executing sequence, then, continue its execution.
(iii). If IO_wait = 0, and CP_wait = 1, allow execution of the stored sequence in CP_seq. Reset CP_wait = 0.
(iv). If IO_wait = 0, and the CP_wait = 0, allow execution of the input node sequence and info segments.

Whenever there is a change in CP_wait the value of the CP_wait is reported to the IOCArb node through outport 1.

IOCArb
IOCArb.c also has its own static variables.

CP_wait = 1 if CP is waiting to use the bus.
= 0 if CP is currently using the bus.
IO_seq is a buffer to hold a IOC originated node sequence.
IO_info is a buffer to hold the corresponding info segment.
IO_wait = 1 if IOC is waiting to use the bus.
= 0 if IOC does not need the bus.

IOCArb algorithm:
For the IOCArb node to be active, either of the two inports must be active. Inport 0 is connected to the IOC node, while Inport 1 is connected to the IOCArb node. The IOCArb algorithm is executed once the node is activated,

(i). If inport 1 is active, set CP_wait to the value of info segment of the token at inport 1.
(ii). If inport 0 is active, and if token info segment indicates the need for the bus, set IO_wait = 1. If bus is not required reset IO_wait = 0.
(iii). If CP_wait = 1, and if IO_wait = 1, then allow execution of sequence and info that has been stored in stored in IO_seq and IO_info, or continue execution of previous sequence.
(iv). If CP_wait = 0, then save input node sequence and info segments in IO_seq and IO_info respectively. Set IO_wait = 1.

Whenever there is a change in IO_wait, the value of IO_wait is reported to the CPArb node through outport 2.

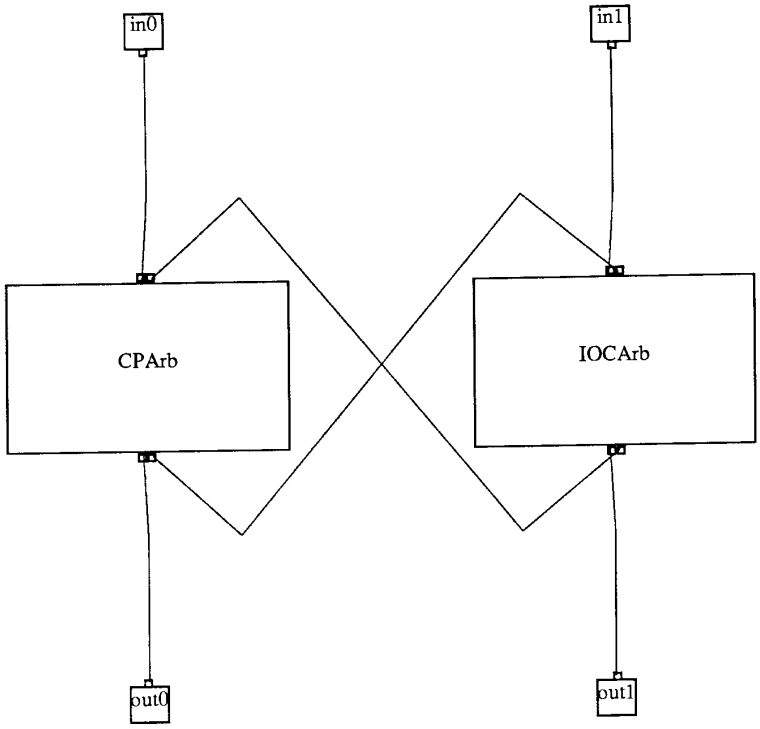

Figure 5. The BA Node sub-graph - BA.hwg

From a comparison of the algorithms it is clear that the IOC is given the priority, because, the IOC has the ability to indirectly set CP_wait (at CPArb - step (ii)) to 1, while the CP does not have this control over IO_wait (at IOCArb). IO_wait can be changed only by the IOC.

Note that at step (ii) of the IOCArb algorithm, the info segment of the token is utilized to indicate whether the particular IO operation requires the use of the bus or not. This facility is provided to take into account the function of the IOC to interact with the peripherals. During this time (a significant delay), the bus would remain idle unless the CP executed its next instruction before the IOC again requests access to the bus. Thus the info segment of token data structure provides the flexibility in being able to specify special conditions for execution.

b. Use of info segment. Just as the CP derives its sequences from the CPSeq file, the IOC derives them from the In0 node which accesses the 'In0.dat' file. It is thus assumed that the I/O instructions depend on the IO devices that the IOC interacts with. The CP command instruction is mapped to the following structure, '18141 35', indicating that it involves an instruction fetch and a CP-IOC-CP data transfer. The info segment uniquely identifies the instruction as a command instruction. The 'chk' node of the IOC, as it receives '41 35' checks for an info segment of 35 and if true, begins the IO chaining process through the Out0 and In0 nodes (Figure 3). I/O instruction node sequences are extracted by the C program mapped to the In0 node from the 'In0.dat' file. The I/O sequences begin and end with the digit '4', the IOC reference number. Also, the sequence that signals the end of the I/O chain, is given the form '40 29'. The info segment in the IO sequence-info pairs signify three types of IO operations as shown in Table II.

Table II. Info segment and its significance in the arbitration scheme

INFO SEGMENT	SIGNIFICANCE
49	IO needs bus access.
39	IO does not need bus access.
29	All IO is complete.

c. An example simulation run. Consider the following as the contents of the CPSeq.dat and the In0.dat files,

Example

CPSeq.dat:		In0.dat:	
18121381	1	48484	49
18141	35	40	29
18121381	2		

These sequence-info pairs may represent a portion of a program that involves some I/O processing. They are manually entered into the two files. The second CPSeq pair represents the command instruction which initiates IO processing at 10.0 time units (Figure 4). An assumption made for this example is that the I/O device has no delay. This assumption is impractical but here it is only intended to demonstrate the working of the arbitration scheme.

Figure 6 provides the timing diagram showing significant events during the execution of the node sequences of this example. The algorithms assigned to the CPArb and IOCArb can thus be tested for correctness.

controllers are used, the BA node scheme will be a more complex version of the present one. Thus, concurrent processing of the CP and IOC can be simulated in the model. This feature provides a larger range to themodel's application. The scheme can be expanded easily to include other contenders for the bus.

In the example of section 6.1.1 it was assumed that only one instruction is executed in the architecture at a time. Pipelined processing can simulated by letting the CPSeq node initiate a block of tokens (sequence-info pairs) rather than a single token. Pipeline design involves a number of techniques whose implementation may be verified using approaches similar to that used for bus arbitration.

6.1.2. Other Examples

The architectures performance is greatly affected by the memory latency. Hierarchical memory systems [14] have been designed to allow a greater influence on cost by cheaper but slower memories, and on speed by faster but expensive memories. The MRP architecture for example, uses both semi-conductor memories as well as

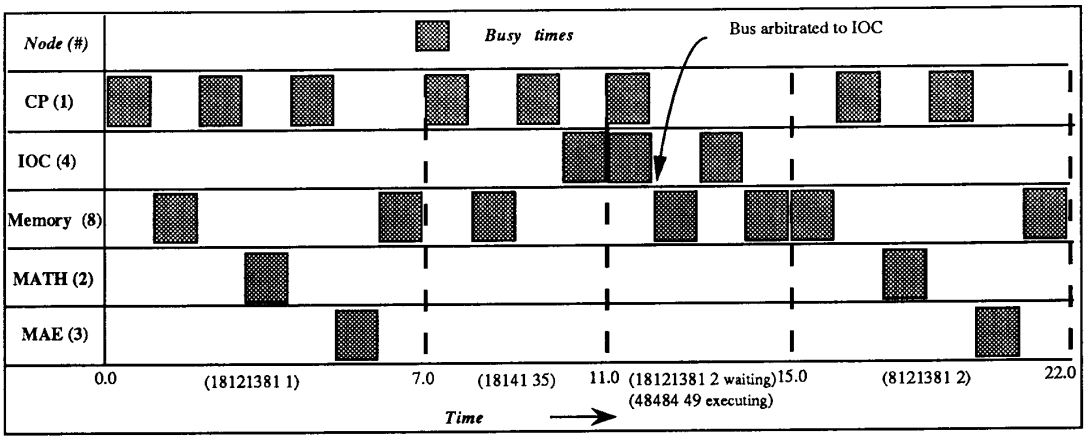

Figure 6. Timing Diagram showing the Implementation of the Bus Arbitration Scheme

The bus arbitration scheme renders the model more complete. In reality, external controllers contend for the memory even through the DMA. In the MRP, the CP and the IOC requests are given a priority over DMA request. Also, all I/O controllers have been treated as a single node called IOC. However, if additional

a magnetic core memory. The Memory node of the uyk.hwg can contain several nodes to represent different memories. Each node can have a different delay depending on the type of memory it represents. To indicate that CP is requesting data from a specific type of memory, we can use the info segment. Also, by

assigning a more complex structure to the token, page faults and memory management schemes can be simulated.

6.2. Performance Measurement

In order to analyze the performance of the architecture for a specific application program or for a small set of crucial tasks within the program, the code must be translated to a set of node sequences. Equally important is the mapping of realistic delays to individual modules. Thus, some preparation must be done before the model can be simulated for performance measurement.

6.2.1. Additional Requirements

Prior to simulation, two types of information are primarily required:

(i). a set of node sequences representing a program or part of a program for which the architecture is being tested, and,

(ii). at least the approximate time delays of the various hardware modules of the architecture. It must be noted that for different types of processing at each hardware module the delay may be different.

Methods of obtaining such information are now briefly proposed.

6.2.2. Program - Node Sequence Mapping.

The inputs to the model are the sequence and info segments. For testing the correct functioning of the model the segments can be manually typed into the files CPSeq.dat and In0.dat. A random node sequence generator program has also been used to continously generate node sequences. However, if the architecture must be analyzed for a benchmark program or any application for which good performance of the architecture is crucial, random node sequence generation is not applicable. If the program contains very few statements, they may be converted manually into node sequence. For larger programs, the conversion must be automated. Consider, as an example, the simple arithmetic operation, represented in the form a C program statement,

```
a = b + c;
```

This statement involves at the register transfer level, loading the CP's registers with the contents of the address of the variables b and c, the addition operation, placing the result in a CP register, and writing the contents of the CP register to the address of the variable a. A maximum of four instructions are required to execute this program statement. These instructions may be mapped to the following sequences (info segment is unused):

```
18181   1  - instruction fetch and read value of b.
18181   2  - instruction fetch and read value of c.
1811    3  - instruction fetch and addition of b and c.
18181   4  - instruction fetch and write value of a.
```

Thus, the four sequences listed, must be placed in the CPSeq.dat file, and the csim program executed to simulate the program statement at the architecture level. Hence, a program-node sequence converter, if built, must at first identify all the instructions corresponding to each program statement. Then, the nodes in the architecture representation that interact during the execution of an instruction, and the sequence of data/control transfer must be identified.

6.2.3. Delay Information.

A fundamental problem with the task of assigning actual delays to the nodes (particularly the Processor node) is that, the processing delays at the CP vary continously. The delay at a certain instance depends entirely on the exact nature of processing for that instance.

Suppose, however, that a specific set of program statements are to be simulated. For each gate level operation it is possible to compute the approximate total delay. This is feasible since the approximate delays for basic logic gates are known. After the computation of such delays, the *firing_delay* attribute of the Processor node on the CP parent node can be varied according to the operation being performed. Although considerable effort is required for the delay computations, the results can be quite accurate. The computations may be simplified by considering only significant delay items involved in the operations. However, this method is suitable only if the number of program statements in the set are small.

For a large set of program statements, the delays must be randomized. The range over which processing time varies can be estimated by brute force computations involving the counting of gates as mentioned in the previous paragraph. The delay within the CP can be a random variable within this range. Simulation must be carried out repeatedly to be able to converge to a certain result. The performance estimates thus obtained will be reasonable and give a good idea of the efficiency of the architecture.

The same methods of delay assignment apply to all other nodes except the Memory node where each memory module has a fixed cycle time that depends on the material and the construction of the memory. For example in the MRP architecture, the memory interface that connects the common bus to the memory modules handles cycle times from 250 milli seconds to 1 micro second.

6.2.4. Performance Studies

If the architecture is being designed mainly for a specific application, the performance for the application is of greater interest. However, if the architecture's design must encompass a large class of applications, performing with equal efficiency for all of them, its general or overall performance is important.

a. Performance for an application. Application programs can be converted to node sequences, and if using random delay assignment, these node sequences can be repeatedly executed until reasonable consistency is observed in the performance statistics. The most consistent values are adopted for performance analysis.

For some applications, the delay in the execution of some programs may be due to a few sets of large iterations within the program. For example, consider the following segment of a program,

```
/* Compute a[i] from i = 0 to 100 */
i = 0;
while (i < 100)
{
        a[i] = b[i] + c[i];
        i++;
}
/* End computation */
```

The 100 iterations performed in the program segment will take a significant amount of time as compared to any single program statement to execute. Thus, if the program contains many such segments which tend to dominate the total execution time of the program, it will not be necessary to convert every program statement into a node sequence. In this way, the task of converting programs to node sequences can be greatly simplified for some applications.

b. General performance. If the architecture must perform equally efficiently for a large variety of applications, the node sequences can be randomized and the model be simulated for a large number of randomly generated sequences. It must be ensured however, that the sequences are not completely random. For instance, there is much more CP-Memory interaction than there is CP-MAE interaction. The sequences generated must be biased to the Memory node to simulate the relatively greater number of references to this node. The CPSeq node in the CP.hwg sub-graph can be mapped to a random sequence generator instead of a program such as CPSeq.c that reads the sequences from an input file. The random sequence generator must include restriction such as,

(i). two adjacent digits must not be the same except in case of the CP, when the digits can be '1'.

(ii). the only node that can execute after the MATH node is the CP node, hence a '2' must be succeeded by a '1' in the node sequence.

(iii) the digit preceding a '2' must be '1'.

Restrictions (ii) and (iii) are due to the fact that the MATH node interacts only with the CP.

7. Future Work

7.1. Complementing Methodologies

Figure 1 showed the constituents of the complete performance analysis methodology. Thus the scope of the discussion in this paper is restrictive but is the core of any performance analysis approach on computer-aided design/engineering tools. A formal approach to design architectural protocols must be developed. This greatly facilitates functional verification and utilizes the simulation methodology correctly. As discussed in section six, techniques for application program-node sequence conversion and delay assignment are required to obtain reasonably accurate measures enabling the evaluation of application specific efficiency of the architecture.

7.2. Lower Level Design.

Building an entire top-down approach up to the gate level on ADAS involves a tremendous programming effort and time. If the approach is applied to only one architecture and then discarded, the effort may not be justifiable. Maintenance of generality while building the top-down approach will lead to a drastic simplification of the top-down synthesis of all similar computer systems in future. Once general primitive building blocks are modeled, it is simply a matter of interconnecting them differently to realize different computer architectures. A library of such basic elements is a one-time effort, and is well worth building, considering its extensive future use.

8. Conclusions

An architecture description tool has been used to build a simple but powerful simulation methodology. Preliminary evaluation of the architecture is simplified by avoiding the details of register level or gate level processing. The methodology is readily applicable for two purposes, functional verification and performance estimation. Results of the application of simulation methodology can provide a basis for trade-off decisions early in the design process. Formal approaches for functional verification and performance estimation must be developed to utilize this methodology more efficiently.

9. References

[1] Charles H. Sauer, K. Mani Chandy, *Computer Systems Performance Modeling*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[2] Usha Chandrasekaran, Sallie Sheppard, "Discrete Event Distributed Simulation - A Survey", *Methodology and Validation - Proceedings of the SCS Conference on Methodology and Validation*, Orlando, Florida, 1987.

[3] Subrata Dasgupta,*Computer Architecture - A Modern Synthesis; Volume 2: Advanced Topics*, John Wiley, New York, 1989.

[4] John P. Hayes, *Computer Architecture and Organization*, 7th edition, McGraw Hill, Singapore, 1986.

[5] Research Triangle Institute, *Architecture Design and Assessment System: User Manual (vers. 3.5)*, 1988.

[6] Geoffrey A. Frank, Deborah L. Franke, William F. Ingogly, "An Architecture Design and Assessment System", *VLSI Design*, August 1985.

[7] Geoffrey A. Frank, Connie U. Smith, John L. Cuadrado, "An Architecture Design and Assessment System for Software/Hardware Co-design", *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, June 1985.

[8] David McLin, Geoffrey Frank, John McHugh, "Use of the Architecture Design and Assessment System (ADAS) Tool Set to Evaluate a Multiprocessor Architecture for Air Chemistry Analysis", *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, New York, 1986.

[9] Jill J. Hallenback, James R. Cybrynski, Nick Kanopoulos, Tassos Markas, Nagesh Vasanthavada, "The Test Engineer's Assistant - A Support Environment for Hardware Design for Testability", *Computer*, April 1989.

[10] Tadao Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, April 1989. p541-580.

[11] T. Smigelski, T. Murata, M. Sowa, "A Timed Petri Net Model and Simulation of a Dataflow Computer", *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, 1985.

[12] UNISYS, *AN/UYK-44 Technical Description*, November 1987.

[13] Frederick J. Hill, Gerald R. Peterson, *Digital Systems - Hardware Organization and Design*, third edition, John Wiley, New York, New York, 1987.

[14] Harold S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Reading, Mass., 1987.