COMPUTER ARCHITECTURE SIMULATION APPLETS FOR USE IN TEACHING

Roland Ibbett and Frederic Mallet[1]

Abstract *- Visualisation of the activities which occur inside a computer is an important aspect of computer architecture education. At the University of Edinburgh we are using a Hierarchical Computer Architecture design and Simulation Environment (HASE) to build a number of architectural models for use in research and teaching. A new facility within HASE, JavaHASE, allows models to be translated into applets which can be accessed via the WWW. JavaHASE applets are programmable simulation models in which the code and data memory contents can be altered, the simulation re-run in the applet and the results used to visualise the activities taking place within the model (data movements, state changes, register/memory content changes, etc). These applets are being used in various ways in teaching.*

Index Terms - applets, computer architecture, simulation, visualisation

INTRODUCTION

Visualisation of the activities which occur inside a computer system is an important aspect of computer architecture education. At the University of Edinburgh we are using a Hierarchical Computer Architecture design and Simulation Environment (HASE [1]) to build architectural models for use in research and teaching [1], [5], [6]. A new facility within the HASE application, JavaHASE, allows models to be translated into applets which can be accessed via the WWW. JavaHASE applets are programmable simulation models in which the code and data memory contents can be altered, the simulation re-run in the applet and the results used to visualise the activities taking place within the model (data movements, state changes, register/memory content changes, etc).

Models currently available include a demonstration of Tomasulo's algorithm, several versions of the DLX architecture [3] and models of the Stanford DASH architecture [7]. Each model is supported by a web site describing the architecture and the model. The DLX models include a version with a simple (integer arithmetic) pipeline, a version with multiple arithmetic units and a scoreboard and a version with two pipelines and a predication mechanism. The DASH models, which include a single node version with primary and secondary caches and a single cluster version with four nodes interconnected by a snoopy bus, are used to demonstrate cache protocols. A multiple cluster, directory protocol, version is being developed.

HASE, WEBHASE AND JAVAHASE

Architectural models in HASE are described using an Entity Description Language (EDL), designed as part of the HASE project. An EDL file contains the description of entities' general properties, instances, connections and the types of data to be exchanged by entities through the connecting links. The screen layout is described by an Entity Layout File (ELF) and the behaviour is implemented in Hase++, a C++-like programming language with extensions to manage discrete-event simulations. Hase++ primitives write events to a trace file which can subsequently be used by the HASE animator to show the results of the simulation in the design window. At the start of a simulation, HASE automatically loads arrays within entities (representing *e.g.* register or memory contents) from appropriately named source files, and the contents of arrays can be viewed in windows launched from the main display.

HASE currently runs on Linux and is being used in a number of research and student projects to develop new architectural models. For many teaching purposes, however, students only need access to the simulation models, not the full facilities offered by HASE, and several mechanisms for offering HASE over the WWW have been explored. The first of these was simjava [4], a process based discrete event simulation package for Java based on Hase++ with animation facilities. simjava has become successful in its own right but from a HASE perspective it offers an alternative way of creating models, rather than a way of presenting existing HASE models over the WWW. WebHASE was therefore developed (originally as a student project) to allow HASE simulations to be viewed with a browser.

WebHASE takes as its input the HASE files used for a project and produces an XML file containing a viewer applet and all the information required to display and animate the simulation. This file can then be accessed via the WWW. Originally, users were allowed to alter model parameters (*e.g.* the instructions in the memory of a computer) and send the new parameters back to a servlet engine which would launch a servlet and re-run the simulation using the new parameters. When the simulation was complete, the servlet would return the results of this

[1]Institute for Computing Systems Architecture, University of Edinburgh, Edinburgh, EH9 3JZ, rni@inf.ed.ac.uk, fmallet@inf.ed.ac.uk

0-7803-7961-6/03/$17.00 © 2003 IEEE

simulation back to the viewer applet to be animated. Partly because the servlet engine proved unreliable and partly because of concerns over the potential for overloading the server, the interactive facilities of WebHASE were disabled and an alternative interactive system, JavaHASE, developed instead. WebHASE is still being used as a demonstration system, allowing pre-run simulations to be viewed over the Web, since it requires less sophisticated browser facilities (JDK 1.2, compared with JDK 1.3 for JavaHASE).

JavaHASE allows HASE projects to be translated into applets containing fully fledged simulation models based on simjava. Since simjava was derived from Hase++, it is able to produce compatible trace files and all Hase++ primitives have a direct equivalent in simjava. JavaHASE applets can be downloaded via the WWW and simulations run on client machines, rather than on a server. In order for students to be able to carry out exercises using the models, the applets require access to cut and paste facilities, using the clipboard, on the client machine. Although standard security managers for applets do not allow access to the clipboard, the security manager can be configured using a Java policy file to allow clipboard access to applets from specified URLs.

TOMOSULO'S ALGORITHM

Tomasulo's Algorithm was first used in the IBM System/360 Model 91 Floating-point Unit [9]. Just like the CDC 6600 Scoreboard [8], Tomasulo's Algorithm was designed to control the flow of data between a set of programmable registers and a group of parallel arithmetic units. Both are designed to ensure that constraints imposed by instruction dependencies are satisfied. Tomasulo's Algorithm is different in that it systematically attempts to minimise delays between the production of a result by one operation and the start of a subsequent operation which requires that result as an input. It is still in use today in processors such as the PowerPC 620 and is frequently taught in courses on computer architecture.

The algorithm works by using additional registers, known as Reservation Stations, at the inputs to the arithmetic units and a system of tags which direct result operands to where they are next needed, rather than necessarily to where they would have gone when the instructions which produced them were issued. The algorithm is difficult to explain to students without a dynamic demonstration.

The HASE simulation model built to demonstrate Tomasulo's Algorithm (Figure 1) closely follows the design of the Model 91 Floating-point Unit. The 360 processor and memory are represented in the model by an Instruction/Data Source Unit which stores a sequence of instructions and a set of data values.

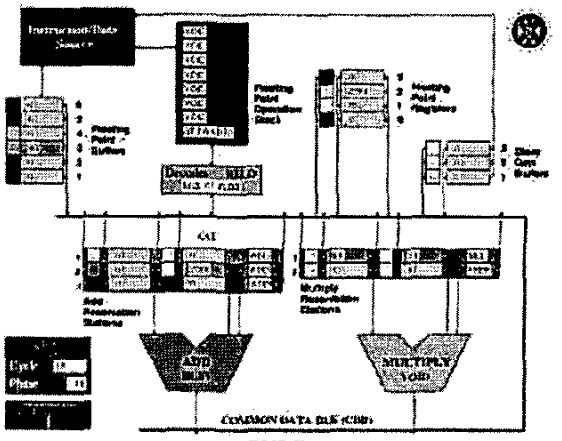

FIGURE. 1
HASE TOMASULO'S ALGORITHM MODEL

As in the Model 91, the instructions are converted into pseudo register-register instructions before being sent to the Floating point Operation Stack (FLOS). For instructions which specify a storage address, the corresponding operands are sent via a queue within the Source Unit (equal in length to the assumed memory latency) to the Floating point Buffers (FLBs). These buffers are allocated cyclicly and the corresponding FLB number is entered into the pseudo-instruction issued to the FLOS. Tags are represented in the model by different coloured icons; each FLB is displayed with the colour of its tag next to it and the reservation stations have their tag colours as background.

Demonstration Program

The demonstration program loaded into the applet forms the scalar (dot) product of two 4-element vectors. Table 1 lists the program instructions in the form in which they are held in the Source Unit's instruction memory and the corresponding pseudo-instructions which the Source Unit sends to the FLOS. The instruction format is two-address, of the form `Function Sink Source`, so the result of an arithmetic operation such as ADD, for example, is

Sink = Sink + Source

As the simulation proceeds, the user can observe the various actions which occur in the model as the program executes. The first instruction (`LD F0 FLB1`), for example, sets the tag register associated with FLR0 to the tag value of FLB1; when the data arrives in FLB1 it is copied into FLR0. Likewise, the second instruction (`LD F1 FLB2`) sets the FLB2 tag in FLR1, but before the data arrives, the third instruction (`MUL F1 FLB3`), has copied the FLB2 tag from FLR1 to the Sink register of Multiply Reservation Station 1 (RS 1) and set the Multiply RS1 tag in FLR1. The FLB2 data therefore goes directly to Multiply RS1 and never appears in FLR1.

TABLE I

PROGRAM INSTRUCTIONS	PSEUDO INSTRUCTIONS
LOAD F0 0	LD F0 FLB1
LOAD F1 2	LD F1 FLB2
MULRS F1 3	MUL F1 FLB3
ADDRR F0 F1	*ADD F0 F1*
LOAD F2 4	LD F2 FLB4
MULRS F2 5	MUL F2 FLB5
.	.
.	.
ADDRR F0 F2	ADD F0 F2
STORE F0 1	ST F0 SDB1
STOP 0	*STOP 0*

Towards the end of the simulation the store instruction (ST F0 SDB1) copies the Add RS1 tag from FLR0 into SDB1, and when the result from Add RS1 appears on the Common Data Bus, it is copied into SDB1 and sent from there to the Source Unit.

DLX APPLETS

Two DLX applets are currently being used for student exercises, one with a simple integer pipeline and one with multiple arithmetic units and a Scoreboard. Using the simple model, students are asked to write two programs which achieve the same effect by two different methods. Both programs are to reverse the order of four bytes of data taken from one word in memory and return the results to a second memory word. The first program is to use byte load and store instructions whilst the second is to load the word into a register and then use shift and logical operations to reorder the bytes before returning the register contents to memory. These programs are both run with different memory access times such that, in performance terms, the first program is better in one case, the second in the other. Students are asked to comment on the advantages and disadvantages of the two methods.

The HASE Simulation Model of the DLX with a Scoreboard is shown in figure 2. The model contains entities representing each of the components in the DLX architecture, the memory, the registers and the pipeline and execution units, together with three other entities which aid visualisation of the activities in the system: the Clock, the Scoreboard and the Pipeline Display. The Pipeline Display shows the instruction currently in each stage of the pipeline while the Scoreboard shows when data and structural hazards occur.

Data hazards are handled in the model through *Use* bits. Each register has an associated Use bit which is set when an instruction that will write to the register is issued and reset when the result is written to the register. Before accessing a source or destination register, the Instruction Decode Unit reads the relevant Use bit in the Registers entity. If the Use bit for a Register required as a source operand is set, a RAW hazard occurs and the Scoreboard shows which register is causing the hazard; if the Use bit for a destination Register is set, the Scoreboard displays a WAW hazard.

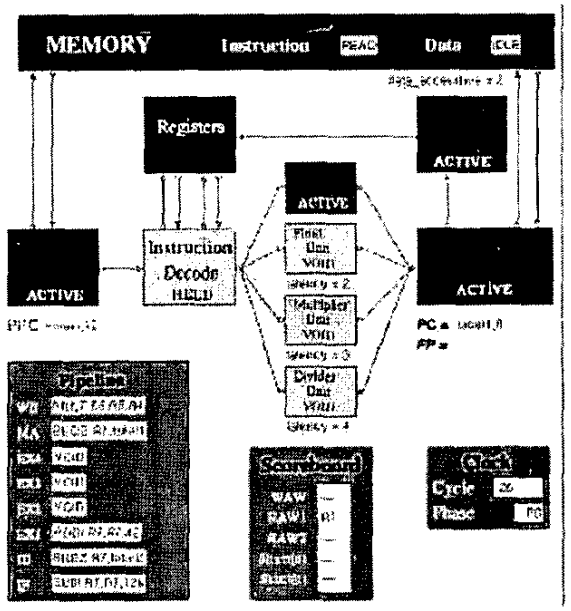

FIGURE. 2
DLX SCOREBOARD APPLET

In this model a structural hazard can occur when two instructions try to use the same pipeline stage simultaneously (the Registers, and thus the Write Back and Memory Access units can only accept one instruction per clock) or when a branch or jump instruction could overtake an arithmetic instruction. To control these situations, the Scoreboard maintains a Latency Pipeline. When an instruction is issued, its latency value is entered into the Latency Pipeline. As an instruction moves through the Execution Unit pipeline, its latency value moves through the Latency Pipeline, decremented by 1 at each move. Thus a multiply instruction has a latency of 3 when it is issued, but in the next clock its remaining latency is only 2. When the Instruction Decode Unit is ready to issue an instruction, the Scoreboard checks its Latency Pipeline for the presence of an instruction with a remaining latency value equal to that of the new instruction. If there is one, the instruction is held up and the Scoreboard displays a structural data hazard.

In order to show the deleterious effects of branches, the Program Counter is contained in the Memory Access Unit and so this is where branches are executed. To prevent a branch from overtaking a previously issued

instruction in one of the execution units, the Scoreboard uses its Latency Pipeline to monitor the presence of any instruction in the execution units and if a branch has to be held up as a consequence, this is displayed as a structural control hazard.

This model has recently been modified to accept assembler output code more easily. In previous models users were obliged to work out the branch distance and include this as an integer in the instruction. In the revised model, program addresses have two fields, one being a label (initially 'main', at memory address 0) and the second an offset. In figure 2 the Program Counter (PC) has just been updated to label1,0 whilst the Prefetch Program Counter (PPC) has just fetched the instruction at main,12 and will be updated to the new PC value at the next clock.

As a practical exercise, students are given an assembly code sequence representing a simple implementation of a scalar (dot) product loop and are asked to run the simulation and note where hazards occur. They are then asked to reorder the code to eliminate or at least reduce the effects of these hazards. As a further optimisation they are asked to unroll the loop to include two iterations of the algorithm in one program loop. Finally they are asked to note the contents of the pipeline (as shown in the Pipeline Display) in each clock period for one iteration of one of their programs and from this information draw a pipeline space-time diagram showing the progress of the instructions through the pipeline.

PREDICATION

Predication is a technique which allows branches to be removed from the code by processing both the *if* and *else* parts of a branch in parallel, removing the problem of mispredicted branches. Instructions in the Intel IA-64, for example, can have predicate tags appended to them by the compiler and an instruction will only be allowed to write its result if the corresponding predicate is true [2]. A branch construct can be rewritten using predication by making the *if* part predicated on one predicate register and the *else* part predicated on a complimentary register. Since only one of the pair of predicate registers holds true, only one of the paths through the branch will write its result.

Predication can be illustrated by the following example taken from [2]:

```
if a[i].ptr != 0 {
  b[i] = a[i].l;
 } else {
  b[i] = a[i].r;
 }
i = i + 1;
```

In a conventional architecture the value of a[i].ptr is loaded from memory and used as the condition for a conditional branch (figure 3(a)). The code is scheduled as four basic blocks; the compare, the *if* part, the *else* part, and the add instruction which follows the conditional statement. Clearly, the result of the compare is difficult to predict. Even if a prediction scheme is used, it cannot correctly predict which path will be taken every time, and the penalty for an incorrect prediction is a pipeline stall costing several clock cycles.

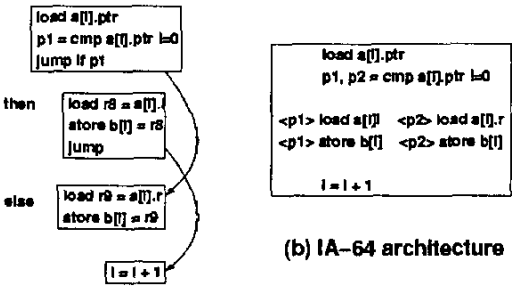

FIGURE. 3
CODE EXECUTION MODELS

Figure 3(b) shows the rescheduled code. After the load instruction, a compare instruction, cmp, compares the loaded value with zero, and sets the pair of predicate registers. The true predicate (p1) is set if the compare was true, and the false predicate (p2) is set if it was false. Only the path with the true predicate is allowed to write its result to memory, the other instruction being ignored.

The JAVAHASE Predication Applet

The JavaHASE Predication applet (Figure 4) is based on a modified DLX model originally developed as a student project. It has a dual pipeline with two simple Integer Execution units and dual data paths (A and B) in all the other units. The instruction set is essentially the same as that of the standard DLX but instructions are always accessed in pairs, simulating the effect of a double length VLIW, and each instruction in the pair also has an extra Predicate Tag field which is used to select one of 8 Predicate Registers. Predicate register 1 is permanently set to 1. Branches can only appear as the first instruction of a pair with the second being the equivalent of the delay slot in a conventional DLX.

The Instruction Fetch unit sends the instructions it has accessed from Memory along the two data paths to the Instruction Decode unit. The Instruction Decode unit accesses the registers for the required operands and sends one instruction to each of the two Execution units. The Execution units send their results to the Memory Access unit which accesses the Predicate Registers to determine whether or not each instruction is to complete. If the predicate for an instruction is true, the instruction may proceed as normal; if the predicate is false, the instruction

is terminated, *i.e.* it does not read or write memory or update a register.

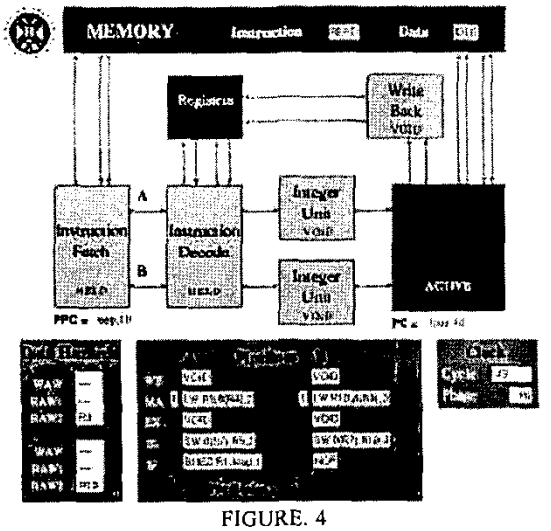

FIGURE. 4
THE PREDICATION APPPLET

New versions of the DLX compare instructions are required to set/reset the Predicate Register values. The standard DLX architecture has six instructions which compare the values in two registers and set the result in a third. The DLX-Pred instruction set includes a second set of compare instructions which set a pair of specified predicate registers, one to true and one to false, depending on the result of the comparison.

Demonstration Program

The Demonstration program pre-loaded into the applet illustrates the execution of the code fragment shown in figure 3(b). The input data for the program consists of eight triplets of data, the pointer (0 = left, 1 = right) and the left and right hand values, stored as a 24-element array. As the program runs, the selected data values of the a array, left or right according to the pointer, are copied to successive locations in the b array. As in the other DLX applets, the model also contains display entities showing the occurrence of data hazards and the contents of each stage of the pipeline. Alongside the display line for the Memory Access unit are the values of the relevant Predicate Registers so students can see which instructions will be executed and which not. The Predicate Registers themselves can be displayed in a separate pop-up window.

THE DASH APPLETS

Applet models of parts of the Stanford DASH (Directory Architecture for Shared Memory) Architecture [10] have proved valuable in helping students to understand the complexities of what might at first sight seem quite simple cache structures. The single node applet (figure 5) includes a (MIPS) processor, a Primary Cache, a Secondary Cache, the Bus used to interconnect multiple nodes, and Main Memory. The processor is modelled very simply as a source of memory addresses which are emitted in sequence. For the purposes of the exercise the cache sizes are set at 8 lines in the Primary Cache and 16 lines in the Secondary Cache. Both caches are direct mapped and each line contains 4 addresses. The Primary Cache operates a Write Through, No Write Allocate policy, the Secondary a Copy Back, Write Allocate policy.

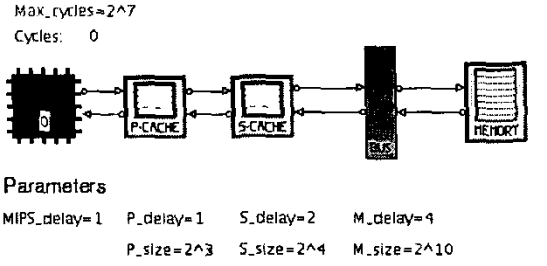

FIGURE. 5
THE DASH NODE APPLET

As part of a coursework assignment, students are asked to create files (which they can copy and paste into the processor's "memory") containing (i) the smallest number of read accesses that will create valid entries in all 8 lines in the Primary Cache (ii) the smallest number of read accesses that will create valid entries in all 16 lines in the Secondary Cache (iii) a sequence of accesses which will exercise all the possible actions which can occur in the caches.

The first two exercises are straightforward, though not all students observe that once the Primary Cache is full in the second exercise, it will then be overwritten as the second half of the Secondary Cache is filled. In the third exercise there are 12 possible actions arising from interactions between the caches and main memory; typically, only a few students find all of them.

Figure 6 shows the DASH Cluster applet which has four nodes attached to the Multi-Processor Bus. The bus displays information about actions occurring as each request is received from one of the nodes. The processors are preloaded with sequences of access patterns designed to demonstrate how the snoopy cache coherency protocols used in a DASH cluster operate.

Students are asked to display the contents of the processors and the secondary caches and, by running the animation in single shot mode, to observe what happens as the simulation proceeds. They are then asked to submit listings of the contents of each the four MIPS files,

annotated to show the responses of the caches to each access, the source of the data in the case of a cache miss and any protocol actions which occur.

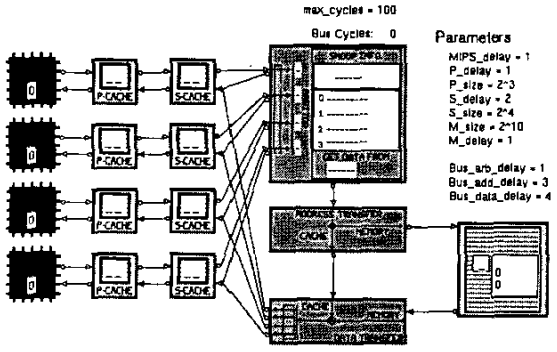

FIGURE. 6
THE DASH CLUSTER APPLET

DINERO

DineroIII is a cache simulator developed by Mark D. Hill and distributed for instructional use with a computer architecture textbook [3]. DineroIII evaluates a uniprocessor cache and produces performance metrics. It allows a variety of cache design options to be varied: copy-back v. write-through, LRU v. random replacement, demand fetching v. prefetching, etc. Although Dinero is a very fast simulator, it has a command line interface and produces text as its output. We have therefore built a JavaHASE Dinero applet which has similar functionality to the original Dinero but with a graphical interface more suited to pedagogical use. Obviously, there would be no point in animating large traces with the applet, so the applet contains predefined tutorial sessions that include default cache configurations along with small preloaded traces. Each trace has been selected to emphasise the benefits and weaknesses of a specific cache design option.

CONCLUSION

Simulation applets offer (a) a means of demonstrating visually the actions which occur inside a computer system as programs are executed and (b) virtual laboratory facilities in which students can undertake exercises which will both test and reinforce their understanding of computer architecture concepts. Several of the applets described here have been used both locally within the University of Edinburgh and remotely in the delivery of an MSc program at the Institute for System Level Integration, a joint venture between four Scottish universities and situated at Livingston, some 20 miles from Edinburgh. Students have also been able to use these applets from home and their use in distance learning is being considered. Some operational difficulties were encountered initially but these have mainly been due to browser compatibility and plug-in installation problems. Webstart technology is being investigated as a way to avoid these difficulties. The existing JavaHASE applets can be accessed from the HASE website at www.icsa.informatics.ed.ac.uk/research/groups/hase Creating applets and their associated websites make interesting student projects and new applets are continually being developed.

ACKNOWLEDGMENT

The development of HASE has been supported by the UK EPSRC through grants GR/J43295 and GR/K19716 and among the many people who have been involved in the HASE project, Stuart Buchanan, John Hawkins, Mark Sawyer and Lawrence Williams contributed particularly to the work described here.

REFERENCES

[1] Coe P.S., Howell F.W., Ibbett R.N. & Williams L.M., "A Hierarchical Computer Architecture Design and Simulation Environment", *ACM Transactions on Modeling and Computer Simulation*, Vol 8, No 4, October 1998, pp 431-446.

[2] Dulong C., "The IA-64 Architecture at Work", *IEEE Computer*, Vol 31, No 7, July 1998, pp 24-32.

[3] Hennessy J.L. & Patterson D.A., "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann, San Francisco, California*, 1996.

[4] Howell F.W. & McNab R., "Simjava - a discrete event simulation package for Java with applications in Computer Systems Modelling", *Int. Conf. on Web-based Modelling and Simulation, SCS*, Jan. 1998.

[5] Ibbett R.N., "Computer Architecture Visualisation Techniques", *Microprocessors and Microsystems, Elsevier*, Vol 23, 1999, pp 291-300.

[6] Ibbett R.N., "HASE DLX Simulation Model", *IEEE Micro*, Vol 20, no 3, May/June 2000, pp 57-65.

[7] Lenoski D., Laudon J., Joe T., Nakahira D., Stevens, L et al "The DASH prototype: Implementation and Performance", *Proc ICSA*, 1992, pp 82-103.

[8] Thornton J.E., "Design of a Computer: The Control Data 6600", *Scott Foresman & Co, Glenview, Ill*, 1970.

[9] Tomasulo R.M., " An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research & Development*, Vol 11, January 1967, pp 25-33.

[10] Williams L.M. & Ibbett R.N., "Simulating the DASH Architecture in HASE", *29th Annual Simulation Symposium, SCS*, 1996, pp 137-146.