Hierarchical Architectural Design and Simulation Environment

Alexander Ronnfeldt Robertson

Ph.D.
University of Edinburgh
1995

June 4, 1995

Abstract

The Hierarchical Architectural design and Simulation Environment (HASE) is intended as a flexible tool for computer architects who wish to experiment with alternative architectural configurations and design parameters. HASE is both a design environment and a simulator. Architecture components are described by a hierarchical library of *objects* defined in terms of an object oriented simulation language. HASE instantiates these objects to simulate and animate the execution of a computer architecture. An event trace generated by the simulator therefore describes the interaction between architecture components, for example, fetch stages, address and data buses, sequencers, instruction buffers and register files. The objects can model physical components at different abstraction levels, eg. PMS (processor memory switch), ISP (instruction set processor) and RTL (register transfer level). HASE applies the concepts of inheritance, encapsulation and polymorphism associated with object orientation, to simplify the design and implementation of an architecture simulation that models component operations at different abstraction levels. For example, HASE can probe the performance of a processor's floating point unit, executing a multiplication operation, at a lower level of abstraction, i.e. the RTL, whilst simulating remaining architecture components at a PMS level of abstraction. By adopting this approach, HASE returns a more meaningful and relevant event trace from an architecture simulation. Furthermore, an animator visualises the simulation's event trace to clarify the collaborations and interactions between architecture components. The prototype version of HASE is based on GSS (Graphical Support System), and DEMOS (Discrete Event Modelling On Simula).

Table of Contents

List of Figures

List of Tables

Chapter 1

Architectural Issues and the Design Problem

1.1 Introduction

Beizer [69] describes an architect's job as " the design of a hardware/software complex, subject to realistic technical, economic, operational and social constraints such that it 1) works, 2) is optimum and 3) survives." Amdahl [2] on the other hand uses the term "Architecture" "to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behaviour, as distinct from the organisation of the data flow and controls, the logical design, and the physical implementation."

The earliest architectures were limited in their instruction sets by hardware technology. As soon as hardware technology permitted, architects began looking for ways to support emerging high-level languages. In the 1960s, stack architectures became popular. They were a good match for high-level languages, given the compiler technology available. To support the 1970's trend towards high level languages and structured programming, architects aimed at hiding hardware implementation details from programmers and compiler writers. The results were both the High-Level-Language Computer Architecture (HLLCA) [32] and powerful architectures like the VAX which had a large number of addressing modes, multiple data types, and a highly orthogonal instruction set.

The last decade has seen a renewed emphasis on machine performance and a return to simpler architectures. Sophisticated compiler technology is bridging Gargliardi's "semantic gap" [27], previously narrowed by HLLCAs which ultimately lead to a semantic clash. Rapid progress in hardware technology has given architects an opportunity to design more sophisticated microprocessor architectures. Silicon real estate, for example, can currently support up to 3 million transistors on a single chip. The microprocessor industry is dominated by fast Reduced Instruction Set Computers (RISC), such as Motorola's new family of 88000s and Intel's i860 and i960 [32], that support superscalar architectures and large on-chip instruction and data caches. RISC methodology increases the semantic gap, because less frequently used instructions are compiled into a sequence of simple instructions, instead of being executed directly by hardware. Performance gains associated with a fast architecture are lost if the high-level language compilation is not optimised efficiently. Performance of the MC88000, for example, improved from 4 Million Instructions Per Second (MIPS) to 20 MIPS after optimisation of code written in assembler [61]. As microprocessor tradeoffs become more complex, a knowledge of hardware and software interaction is essential to understanding how a compiler can extract performance from an architecture. There are cases in which compiler strategies cease to be optimisations and actually slow down code execution e.g., when a temporary variable used in a global common subexpression elimination cannot be allocated to a register and requires a memory reference.

Research is required to explore how to avoid situations in which performance is degraded by optimisation techniques. To study optimisation methods requires an instrument to probe aspects of an architecture that may be exploited by a compiler. The proposal presented here involves the design and development of an interactive general purpose architecture simulator which supports an experimental platform to investigate hardware and software interaction critical to compiler optimisation techniques. This chapter identifies a number of issues where a visualising simulator could offer significant support to designers.

1.2 Defining Computer Architecture

Stone [69] defines the study of *Computer Architecture* as " the study of the organisation and interconnection of components of computer systems. Computer architects construct computers from basic building blocks such as memories, arithmetic units, and buses". The complexity of computer systems is better understood when the architecture is organised into different levels of abstraction. Analysis of each individual level can provide an orderly understanding of the system's functions. Progression from the most primitive level of the hierarchy to higher levels is accomplished by creating a series of abstractions [6]. By suppressing unnecessary details, each abstraction contains only the information relevant at the higher level. At the lowest level an architecture is described by a set of electronic circuit diagrams, the *Circuit Level.* These circuits represent an implemention in hardware of the logic circuits at the next level up in the hierarchy, the *Logic Level.* Above the Logic Level is the RTL *(Register Transfer Level* at which a floating-point arithmetic unit, for example, is represented as an interconnected set of registers and primitive ALUs. Above this is the ISP Level *(Instruction Set Processor Level)* at which a processor is described as an interconnected set of functional units (floating-point units, caches, *etc*). At the highest, PMS Level *(Processor, Memory, Switch Level)*, multiprocessor systems can be represented as ensembles of interconnected processors and memories. The current version of HASE is concerned only with the three upper levels of this hierarchy; extension to the lower levels is inherent to the HASE concept but is not currently implemented.

1.2.1 The PMS Level

The PMS *Processor Memory Switch* level defines components that interact to exchange information. They are distinguished by the kinds of operations they perform:

Memory M: a component that holds or stores information over time. Its operations are reading and writing instructions and data out of and into memory. The memory may be considered as a number of submemories.

Link L: transfers information from one component to another. The operation is that of transmitting an instruction or data from the component at one port to the component at the other.

Control K: a component that evokes the operation of other components in the system. With the exception of the processor P, all other components are essentially passive and require an active agent to set them into episodes of activity.

Switch S: each switch has associated with it a set of possible links, and its operation consists of setting some of these links and breaking others.

Processor P: a component capable of interpreting a program in order to execute a sequence of operations. It consists of a set of operations of the classes listed above (M, L, K, S) to obtain instructions from memory and interpret them as operations to be carried out.

1.2.2 ISP Level

At the *Instruction Set Processing Level* each instruction specifies its operation (or operations) and the data structures that it is to act upon. Superimposed on this is a control structure that specifies which instruction is to be executed next.

Normally this is done in the order in which the instructions are given, with a *jump* out of sequence specified by a branch instruction.

At the logic level the computer system is composed of parallel devices, with all components active simultaneously. At the ISP level, computers are represented essentially as serial devices, executing one instruction after another (or 2 or 4 depending on whether the machine is superscalar). The ISP is essentially linguistic in nature, the logic level is not. At the ISP level objects can be labelled, decisions made and instructions interpreted. The ISP level does not implement decision or interpretation mechanisms, only their functional characteristics are of importance to ISP simulation.

1.2.3 RT Level

The components of an RT (*Register Transfer*) level system are registers and functional units which operate on data as it is transferred between registers. The system undergoes discrete operations, whereby the values in various registers are combined according to the appropriate function and are then stored into some other register. The laws of combination may be anything from the simple unmodified transfer (A=B) through logic combinations (A = B *OR* C) to arithmetic functions (A = B+C). Thus a specification of the behaviour, equivalent to the boolean equations of sequential circuits or the differential equations of electronic circuits, is a set of expressions which give the conditions under which such transfers will be made. Register transfer level systems are usually visualized as having two components: Control and Data. The data part is composed of registers, operators and data paths. The control part provides the sequence of timing signals that evoke activities in the data path. This may be implemented as a hardwired state machine or a microprogrammed sequencer.

1.3 Architecture Experiments

This section identifies some architectural issues that are critical to a microprocessor's performance and that are addressed in chapter 6 to demonstrate the *design*, *simulation* and *performance evaluation* phases of HASE. The architectural issues discussed here include memory and cache design, pipeline interlocking tradeoffs and onchip parallelism including the design of multiprocessors. A study of these particular architectural issues can suggest improved methods for extracting performance through more effective hardware/software interaction. One of the main aims of HASE is to focus on visualising these issues in order to gain further insight into achieving performance. The nature of HASE allows rapid modifications to the design of hardware to experiment with a variety of, for example, pipeline configurations, instruction and data fetch buffer sizes, or likewise examine how rescheduling a sequence of instructions can increase the utilisation of parallel function units. HASE can return the exact number of clock cycles it takes to process a given sequence of instructions.

1.3.1 Memory Latency Issues

A computer's external cache memory is critical to its throughput, but the cache's performance varies with the operating system and with the applications being run. The use of multi-tasking, and LANs to allow multiple users access to a common data bus, affects the *temporal* and *spatial* locality of data in a cache. A small cache is therefore less likely to hold the instruction set window, and cache misses, for example, are more likely to occur. One of the questions facing architects is what cache size is enough to achieve a reasonable (85 % to 95 %) cache hit rate.

Cache Associativity

There are three types of cache organisation that must be considered when carrying out realistic computer architecture simulations: direct mapped, n-way associative and fully associative [35]. Although cache design techniques have not reached the point where it is possible to predict cache performance as a function of cache organisation and external machine architecture, simulation experiments of cache environments can provide the foundation for proposed designs of on- and off-chip caches. Each memory location of a direct mapped cache is mapped to one cache location. The disadvantage of this design becomes apparent when a cache miss occurs and the missing address must be loaded into the cache. Because of a one to one correspondence between cache and memory locations, the desired location will automatically replace the cache location to which it is mapped, instead of, for example, the least recently used cache entry [66]. If the next instruction immediately accesses the replaced data, the memory and cache must swap locations for a second time; this 'thrashing' process is defined to be the worst case behaviour of a direct mapped cache. However, the advantage of this placement policy is the fast hit-time associated with the simple address tag relationship between the processor and memory. Hill's simulation experiments [34] suggest that if the cache is large then the fast hit-time advantage outweighs the small probability of worst case behaviour.

The fully associative cache is the extreme opposite to the direct mapped cache. The least recently used replacement policy can be applied because each location can contain any memory address. When a cache miss occurs, the missed instruction or data can be written anywhere in the cache, preferably replacing the instruction or data item that has the lowest probability of being fetched by the processor during the next instruction cycle. Its major disadvantage is that it is complex to design, especially for large cache sizes, because a sophisticated tagging mechanism is required, involving as many comparators as there are cache

locations. On a restricted piece of silicon real estate, the percentage increase in hit-rate does not justify the extra space and design cost. Furthermore, because of its complex circuitry, a fully associative cache has a poor access time compared to direct mapped caches.

An n-way set associative cache is a tradeoff between the direct mapped and fully associative cache. Instead of each memory location being mapped to only one cache location, it is mapped to one of 2^n locations. Hence the designer may decide to use a 2-way set associative cache, or a 4-way or 8-way, depending on the behaviour of the targeted computer system and its application. Although it does not have the freedom of a fully associative cache, an n-way associative cache can execute a restricted version of a least recently used replacement policy, which compares n cache locations and replaces the location which has the lowest thrashing probability. The set associative cache is organised into sets and blocks; an n-way set associative cache will have n blocks in each set. Simulation experiments [67] have been used to determine the most efficient block size for a given processor type (i.e. RISC or CISC) and application. During the last decade, computers have supported many different types of caches and cache hierarchies. Until now a direct-mapped cache could support virtually all the microprocessors used in PCs. However, the next generation of microprocessors, such as Intel's new Pentium, will contain multiple processor units that can operate independently of each other. With one or more processors making memory calls at clock rates approaching 100MHz, a direct-mapped cache of any practical size simply cannot sustain an acceptable hit rate. As a result, set-associative cache will be used increasingly in future PC designs.

Design analysts use simulation studies to search for a cache placement compromise because the problem cannot be described mathematically. Hardware and software simulators are both used, the software simulator provides the bulk of the simulation results, while the hardware simulator verifies the software simulator's results. Researchers at Digital [33] have derived a few equations for estimating the

effectiveness of hierarchical memories by studying how parameter changes affect the overall system's performance [1] [76]. These general relationships do not appear to depend on operating system, compiler, architecture or workload. Digital emphasises however, that these equations, derived empirically from VAX hit-rate data, cannot replace thorough simulation. These rules of thumb are particularly useful to designers who have no time for extensive simulation studies. The equations include relationships such as: miss rate as a function of size, miss rate as a function of associativity, optimal block lengths, and refill bandwidths.

Cache design goals are generally to reduce processor bus activity, in the case of the 68000, for example, to between 70% and 90%, and to increase the CPU's throughput by reducing the average memory access time.

Cache Consistency Methods

HASE can provide a framework in which to consider cache consistency mechanisms. The hardware/software interaction associated with cache consistency is difficult to visualise, and contemporary high level functional behavioural tools are inadequate at capturing and visualising its mechanisms and providing a means to obtain performance evaluation of a variety of different schemes.

If a multiprocessing system is required, for example, adding another FIFO to the main memory controller FIFOs allows *reflective reads* without providing separate reflective read circuitry in each cache controller. Reflective reads help to reduce bus traffic in a multiprocessing system whenever a *snooping* cache supplies data to another processor that has experienced a cache miss. The cache controller for another processor in the system, which has been snooping the bus, observes that it has a valid copy of the requested data and inhibits the first processor's read to the main memory and supplies the requested line. In a reflective read the cache for the second processor updates main memory at the same time that it is supplying the data to the first processor and its cache. The challenge in

reflective reads is that data must be buffered to accommodate the speed of the main memory.

Caching on Decoupled Architectures

One of the experiments reported in Chapter 6 attempts to examine the effect of decoupling the address and data bus from the cache and processor. The aim of the experiment is to reduce memory latency and increase processor tolerance to memory latency by decoupling its memory address and data busses. Various current trends in computer architecture involve one or the other technique, and in some cases it has been suggested that both be used [24].

The primary time where the latency of main memory will contribute to execution times on a decoupled architecture is when the Address Processor (AP) and Data Processor (DP) must synchronise; so-called "loss of decoupling" events. Trace simulation has shown that caching can reduce memory latency caused by AP and DP synchronisation. In a multiprocessing environment, cache coherency must be considered, and this introduces further overheads. Detailed simulation studies can identify the successfulness of hardware and software based coherency schemes. The future version of HASE will provide a useful framework in which to carry out similar kinds of experiments.

The PIPE architecture [20] improved memory tolerance by incorporating short queues, of the order of 64 bytes long, between the instruction cache and the instruction unit and between the data cache and the load and store units. SPICE simulation runs indicated that the PIPE approach was 2 to 3 times faster than the MIPS and RISC-II architecture, achieving an estimated 18 MIPS. However, to achieve this performance, the PIPE's data queues must remain filled and this is only possible when data accesses can operate in advance of arithmetic operations.

1.3.2 Pipeline Issues

Pipelining is one form of embedding parallelism or concurrency in a computer system. Operations are broken up into short stages connected by interstage registers. If a pipeline has N stages, then up to N different instructions may be operated upon in parallel in an assembly-line fashion. Pipelining techniques are especially applicable to the design of very high speed systems in which interchip delays are so large relative to the clock period that the entire system must be pipelined. A pipeline architecture involves specifying control structures for data buffers, bus transfer, branching and interrupt handling. This involves defining precisely which control signals are enabled and disabled to execute the reponsibilities of a pipeline [59].

Appropriate bounds are provided which reflect the fact that sometimes the interval between pipeline completions is close to the pipeline segment time and at other times it is close to the flush time.

Definitions

Pipeline *throughput* is defined as the number of outputs per unit time. It directly reflects the processing power of a processor system. Pipeline *utilisation* reflects directly how effective a processing scheme is and is used to suggest possible improvements for removing pipeline bottlenecks.

For example, a pipeline's *Clock rate* limits the speed of data and control flows between pipeline segments. The propagation delay through each segment and possible signal skews must be carefully balanced to avoid improper gating in high performance systems [37].

Pipeline *Design Optimisation* is defined by the *cost-time* product. The speed-cost product represents a simple analytical model for comparing the relative effectiveness of different pipeline configurations. A pipeline's speed is defined in

terms of the minimum time required for a pipeline to produce a result, given the total time to execute an operation without pipelining, the number of stages in the pipeline and the propagation delay for each pipeline stage (equation 1.1). The cost of the pipeline is given in terms of the initial cost to implement the operation plus the extra hardware cost for each stage of the pipeline (equation 1.2).

$$segment\ time = T/k + \tau \tag{1.1}$$

$$cost = \alpha.k + \beta \tag{1.2}$$

where

T = time for non-pipelined case

τ = latch time

α = extra cost of each segment

β = initial cost

k = number of pipeline stages

The cost-time product is given by:

$$(T/k + \tau) * (\alpha.K + \beta) \tag{1.3}$$

This analytical model is acceptable for simple pipelines but for complex problems involving hazards and penalties it cannot express a bound for execution time and efficiency. Experimenting with the relative efficiencies of various pipelines is therefore a useful target for architecture simulation.

Pipeline Configurations

A variety of pipeline classes may be studied using different memory configurations [47]. One configuration may involve an off-chip on-package instruction cache

and an off-chip on-package data cache, another may consist of an off-chip on-package instruction cache and an off-package data cache. Each configuration can be characterised by a ratio of data access delay to datapath delay. Experiments involve pipelining the memory to handle a particular memory latency. Some of the candidate pipelines may be obviously better than others but an important question for designers is how much better they are and if so, whether the performance improvement is sufficient to justify the increased implementation cost. Furthermore, performance evaluation can measure the utilisation of different pipeline configurations and estimate the impact of compiler optimisation mechanisms. If there is considerable improvement then it may be useful to invest in more optimisation strategies.

Interlocks and Interleaved Pipelines

Dependency conflicts between successive instructions can be classified [41] into three types: first order, second order and third order.

A first order conflict occurs whenever an instruction which is about to be issued requires the use of an arithmetic unit or a result register which is already in use or has been reserved by a previously issued, but as yet uncompleted instruction. Each instruction in the following pair, for example, requires the use of the Floating Add Unit

```
R6 = R1 + R2
R5 = R3 + R4
```

while in the next example each instruction requires R6 as its result register

```
R6 = R1 * R2
R6 = R4 + R5
```

Although this latter example is unlikely to arise in normal programming practice, it must nevertheless give the correct result. Without proper interlocks the add

operation would complete first and the result in R6 would then be overwritten by that of the multiplication.

A second order conflict occurs whenever an instruction which is about to be issued requires the result of a previously issued but as yet uncompleted instruction. An example of such a conflict is the following

```
R6 = R1 + R2
R7 = R5 / R6
```

Here the second instruction can be issued, but must not be allowed to start until the result of the first instruction has been entered into R6.

A third order conflict occurs when an instruction which has just completed its operation wishes to store its result in a register which is waiting to supply an input operand for a previously issued, but as yet unstarted instruction. Such a conflict occurs in the following sequence

```
R3 = R1 / R2
R5 = R4 * R3
R4 = R0 + R6
```

Because of the length of a pipeline, an operation like $R0 = R1 + R2$ will not update $R0$ until N clock cycles after it is issued, assuming that there are N stages and that each stage completes in one clock cycle [37]. Thus the sequence:

```
R0 = R1 + R2
R3 = R0 + R4
```

will produce an erroneous answer if the two instructions are dispatched in consecutive cycles. There are two basic solutions to this dependency problem: 1) detect data dependecies and *stall* the execution of instructions as required or 2) rearrange and pad instructions by "software pipelining" at compile time to eliminate the dependency problem.

With both pipeline interlocks and instruction resequencing, the effect of data dependencies is to reduce the throughput of the processor by introducing gaps in the pipeline execution. With interlocks, these gaps are generated by a hardware mechanism; with resequencing, they are generated by inserting NOOP instructions. These gaps may represent a significant loss of throughput.

A related problem arises from the presence of jump instructions in an instruction stream. In highly pipelined processors, the next instruction fetch may begin long before the current instruction has been fully decoded and executed. Thus it may be impossible to update the machine's program counter correctly before the next few instructions are fetched. If one instruction is issued per clock, for example, and a jump instruction takes N cycles to fetch and execute, then $N-1$ instructions following the jump will always be processed, since they will have been fetched before the program counter was updated. Thus straightforward program coding could yield incorrect results.

The traditional solution to this problem is to implement in hardware a pipeline flushing mechanism to discard unwanted instructions after the jumps. Flushing these instructions represents a considerable performance penalty since flushing wastes the time spent processing the unwanted instructions and the extra main memory accesses used to fetch the unwanted instructions are wasted. Chapter 6 demonstrates how HASE visualises the pipeline flushing penalty. Research at the Institute of Technology in Chicago is currently developing a processor architecture DEMUS (Delay Enforced MUltiStreaming) [37] to eliminate both pipeline interlocks and the jump problem. DEMUS adopts a novel dynamic stream interleaving technique to fill gaps in execution required for the correct operation of one stream with useful work for another. This technique is similar to that proposed in the Context Flow architecture [52]. An example of this technique is demonstrated using HASE in Chapter 6, where a number of independent instructions are inserted into a correlation program containing a large number of data and control hazards.

1.3.3 Internal Parallelism

In addition to exploiting parallelism at the instruction level through pipelining and *superpipelining*, further performance improvement can be achieved through VLIW (Very Long Instruction Word) and superscalar architectures.

Definitions

For a conventional scalar pipeline processor the total time required for a program to execute, (E), is given by:

$$E = N * C * T \tag{1.4}$$

where:

N = number of instructions in the test program

C = average number of processor cycles per instruction

T = time period of one clock cycle

One method of reducing the total program execution time is to issue more than one instruction per clock cycle [21]. Machines that issue multiple independent instructions per clock cycle are called *superscalars*. The Single Instruction Multiple Pipeline (SIMP) [47] is a superscalar processor that supports reordering and data forwarding methods similar to the IBM 360/91. For example, to allow out-of-order execution the E-stage of all four identical instruction pipelines contains a *Wait Reorder Buffer* corresponding to the IBM 360/91's reservation stations. The SIMP supports a Dependency Handling Register File which receives register identifiers and operation types from each instruction unit pipeline, updates its data and control dependency tables and issues instructions waiting execution to the Wait Reorder Buffer. On every clock cycle the Wait Reorder buffer is checked and if there are no flow or data dependencies, the instruction is executed by one of the five pipelined functional units. The total program execution time for a

superscalar without data dependencies is given by:

$$E = \frac{N * C}{P} * T \tag{1.5}$$

where P = number of pipelines

Rather than untangling data dependencies using hardware, the Very Large Instruction Word (VLIW) processor relies on the compiler to create a package of n instructions that can be issued simultaneously. The compiler is programmed with the knowledge of the internal architecture and is responsible for scheduling the order of operations strictly in order to prevent any hazards from incorrect results at runtime. The VLIW, for example, may include two integer operations, two floating point operations and a branch. The instruction would have a set field for each functional unit yielding an instruction length of up to 168 bits. The total execution time now becomes:

$$E = \frac{N}{F} * C * T \tag{1.6}$$

where F = number of operations included in one instruction

The VLIW approach is limited by the amount of instruction-level parallelism available. For example, to keep a VLIW architecture with 7 functional units busy, the compiler would have to package 15 to 20 independent operations. This is because it is necessary to find a number of independent operations roughly equal to the average pipeline depth times the number of functional units. The behaviour of such a superscalar architecture is clearly visible during a HASE animation.

When the parallelism comes from unrolling simple loops, the original loop could probably have been run more efficiently on a vector machine. It is not clear whether a VLIW is preferred over a vector machine for such applications [32]. An open question is whether there are large classes of applications that are not suitable for vector machines, but still offer enough parallelism to justify the VLIW approach.

A general *superscalar* architecture is shown in Figure 1–1. In a superscalar machine, the hardware can issue a small number (i.e. 2 or 4) of independent instructions in a single clock. A general *superscalar* architecture is shown in Figure 1–1. The general *superscalar* architecture shown in Figure 1–1 shows n execution pipelines, comprising D_n decode stages and E_n function units. In addition, there are two new stages: the instruction window and the result window.

The instruction window acts as a buffer between the decoder and the functional units to minimise stalling. The result window buffers the result values to make sure they are placed into the register file in program order, even if they are completed out of order by the functional units. For each stage of the superscalar processor it is possible to identify a number of advantages that are available and disadvantages that must be overcome.

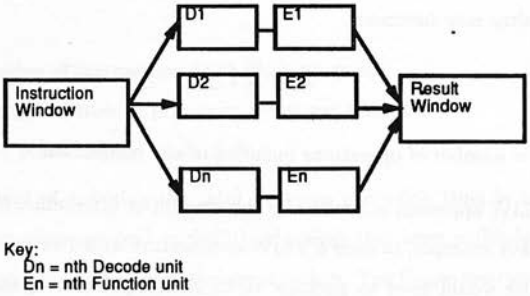


Figure 1–1: A Superscalar Architecture

For example, since the goal of a superscalar processor is to execute multiple instructions per cycle, the *decoder stage* must also be able to decode several instructions per clock. The number of instructions that it can decode per cycle is referred to as the *decoder width.* However, due to branches, not all the instructions fetched by the decoder may be valid for execution on that cycle, so mechanisms must be developed, tried and tested using simulation models to *align* and *merge* instructions to keep the decoder busy.

Similarly, an instruction cannot always be issued immediately, due to resource conflicts or dependencies. To avoid stalling the decoder an instruction window is placed between the decoder, and the functional units. This buffer can be implemented either as one central window as in the MC88110, or as a set of reservation stations, one for each functional unit.

Comparison of Superscalar Processors

Although there are several commercially successful supercalar microprocessors, for example, Digital's ALPHA, MetaFlow Lightening Sparc, Intel i960, IBM RS/6000 and Motorola MC88110 RISC family, computer architects still search for the most cost effective configuration [44].

Computer architects have experimented with varying the number of instructions fetched per cycle. For example, the Metaflow and IBM designs can decode four instructions per cycle, whereas the Intel chip decodes 3 and the MC88110 and ALPHA currently only decode 2 instructions per cycle.

The Metaflow and 88110 instruction window is a "central design" rather than "reservation stations" system, whereas the RS/6000 could be classified as either. This is because the RS/6000 has only one functional unit of any particular type, while the Metaflow and MC88110 have multiple units.

The Metaflow and MC88110 predict branches based on a branch history table, whereas the ALPHA and i960 allow the compiler to perform the prediction. The ALPHA has few deeply piplined function units, in contrast to the Intel i960 which has more relatively slow, functional units than the Metaflow.

The RS/6000 has 4 highly optimised function units to allow a new instruction to be issued on each cycle in most cases. The Metaflow architecture is the most scalable of the five because of the generic design of its DCAF (Dataflow Content-Addressable FIFO) unit. The DCAF is a scheduler; adding more functional units

to the Metaflow design will only require improving features that already exist, for example, adding more control logic to the scheduler.

The RS/6000 and the ALPHA, on the other hand, will have to be redesigned significantly to allow multiple function units of the same type and provide the necessary synchronisations between them. As the microprocessor industry improves the speed of its functional units and/or adds more of them, the task of evenly assigning instructions to function unit becomes a complex design issue.

A goal of HASE is to provide a large library of software objects to simulate the execution of superscalar architectures, for the purpose of investigating the performance of superscalar configurations under a variety of runtime scenarios. For example, the technique of "delayed branching" can sometimes degrade the performance of a *superscalar* processor because the compiler must seek N independent pairs of instructions to feed the execution pipeline.

1.3.4 Code density Experiments

RISC processors offload more complex and/or frequently used instructions onto a high level compiler for emulation by simpler instructions. CISC (Complex Instruction Set Computers), on the other hand gain performance by building into the processor high semantic content instructions executed by microcode. CISC processors try to reduce the number of external instructions that the microprocessor must fetch, in an attempt to avoid the common von Neumann bottleneck. RISC processors have lower density instruction sets and demand a higher memory/processor bandwidth, which is one reason why there has been extensive research evaluating cache performance. Because RISC instruction set formats are simple to decode it is possible to achieve, with caching and pipelining, an average of one instruction execution per clock cycle or even more with superscalar architectures [68]. CISC processors have higher density instruction sets and require a lower memory/processor bandwidth, but may sometimes require up to 20 clock cycles to execute one in-

struction. The RISC processor has a simple architecture, however it relies on a clever optimising compiler to achieve its performance. The CISC architecture is very complex but it requires less sophisticated compiler optimisation to attain maximum instruction throughput.

One compromise between the RISC and CISC methodology is the **enhanced** instruction set processor. The MC68030 adopts CISC type addressing modes and microcoded variable length instructions, but only the most frequently issued instructions are microcoded. The 68030 is defined as an enhanced instruction set processor for which the compiler reconstructs more complex instructions. By reducing the number of microcoded instructions the memory to processor traffic is increased. To increase memory to processor bandwidth the MC68030 for example, adopts a RISC approach, and includes two 256-byte instruction and data caches.

At the University of Stanford simulation experiments have attempted to measure the effect of code density on cache miss performance [39]. Intuitively, if the code is very dense then a larger proportion of the executed code is likely to be in the cache and therefore the miss rate will be lower than in a less dense, more reduced instruction set.

HASE will support the software objects infrastructure for a computer architect to investigate tradeoffs between increasing the code density to decrease the number of instruction cache misses, and decreasing code density to reduce the complexity and cost of a microprocessor's implementation.

1.3.5 Overlapping Register Windows

RISC architectures generally have a large number of registers, to reduce the number of possible 1st, 2nd and 3rd order conflicts. The size of the register set is an issue for simulation study because it is another tradeoff between the amount of storage space available on the processor's chip, the memory/processor bus traffic and compiler complexity. Clearly, the more storage there is on a microprocessor

chip the less traffic there is between memory and processor traffic. The RISC-I designed at U.C. Berkeley, for example, has 138 32-bit registers and uses (overlapping) register windows for parameter passing [48] [32]. To invoke a procedure, it is normally only necessary to update the window pointer and change the program counter. Other RISC processors restrict the instruction set to only 32 32-bit registers, and consequently require fewer CPU cycles for instruction decode.

1.3.6 Impact of Compiler Optimisation

High level optimisation has made compiler technology a major feature of microprocessor system design. Compilers take a high-level language and translate it into a universal code.

A code optimiser is written specifically for a particular architecture. A code optimiser rearranges an application's source code to take advantage of the hardware's architectural features and improve the runtime performance.

For example, current C optimising compilers use methods such as strength reduction, constant folding, least common subexpression and peephole improvement. Strength reduction [32] replaces time expensive operations, for example power operations, with cheaper operations, in this case multiplication. Constant folding optimisation reduces constant expressions into constants to save repeating the calculation. Least common subexpression [27] assigns variables to evaluated expressions that are likely to be used again in the same routine. Peephole improvement [3] takes a sequence of target instructions and replaces them by a shorter, faster set of instructions [39].

Another common optimisation method known Register Colouring reduces the number of variables and temporary variables, where usage scopes do not overlap. This means that less register context saving/restoring is required for function invocation. The simple example below demonstrates that because the scope of i and j do not overlap, they can re-use the same register.

```
C Source:
    for (i=0;i<n;i++)
       {......code.......}

    for (j=0;j<n;j++)
        { .....code.....}

Result:
 "i" and "j" will require only one register for both variables.
```

A compiler writer can test the effectiveness of a new optimisation strategy by running handcrafted optimised sequences of instructions on the target architecture simulated in HASE. A required feature of HASE is to provide visual feedback on how efficiently an optimisation technique utilises an architecture's resources.

1.4 The Architecture Design Problem

Although there is considerable discussion about top-down design, most tools available today start from the middle, at the Register Transfer Level. Designers are left to their own devices at higher levels of abstraction, especially when building hardware/software systems [72]. This section identifies how existing simulation tools are used to visualise the architecture issues described in the previous section. It attempts to distinguish the main design, conceptual and presentation difficulties.

1.4.1 Analytical Approach

Trace driven simulation and hardware measurement are techniques most often used to obtain accurate performance figures for caches. The former takes a large

amount of simulation time and the latter is restricted to measurements of existing caches. By representing the factors that affect cache performance, an analytical model that gives miss rate for a given trace as a function of cache size, degree of associativity, block size, subblock size, multiprogramming level, and task switch can be produced. The model involves a judicious combination of measurement and analytical techniques. However for more complex systems an anaytical model lacks accuracy. It can be useful to identify aspects of a program's behaviour where effort would be justified to improve cache performance. Multiprogramming traces are difficult to obtain, and to analyse the relative performance of different coherency models discussed earlier, the event possibilities extend beyond the bounds of an analytical approach. Similarly, queueing theory analysis can be useful for describing scalar and superscalar execution pipelines, but mathematics can only return quantitative figures for Cycles Per Instruction (CPI); it does not describe the nature of interactions between hardware components and software.

1.4.2 Simulation Approach

Scope of Hardware Description languages

Hardware Description Languages, for example **Verilog** and more recently **VHDL**, are similar in principle to BLMs except rather than programming in C they express the functionality of components as "hardware descriptions". HDLs can provide fast function level simulations of custom devices, and the last 2 years have seen a move towards standardising design tools to VHDL, so logic designers can exchange and add compatible hardware modules.

Systems based on VHDL and Verilog cannot be stretched to specify complex systems at the highest level. HDLs are useful for specifying RTL descriptions and providing an environment in which to experiment at the logic or gate level. HDLs can simulate signals on the input and output pins of devices and they provide libraries of components, for example latches, buffers, registers and D-flipflops that

are useful for testing a variety of logic designs. A timing diagram verifies that the logic is generating signals at precisely the correct time. HDLs provide elaborate multi-level logic hierarchies, to hide the detail of a component in a *black box*, for the purpose of simplifying the overall design. The black box effectively acts as a truth table; for every sequence of input signals it returns a corresponding set of output signals.

There are hardware description languages available for designing microprocessors. COSMOS [36], for example is a logic level symbolic simulator that converts transistor nodes into a symbolic language file which in turn is converted into a fast executable C program. MENTOR [38], offers a more usable input and output graphical interface, but runs more slowly, especially when a complete system simulation is required. MENTOR Graphics design tool is a logic level discrete event driven simulator which maintains a library of off-the-shelf devices that a designer may block together and check for correct timing characteristics. The MENTOR system provides Behavioural Language Models (BLMs) written in C to support the facility to simulate custom devices, for example, a BLM was written to simulate a static and a pseudo static RAM.

A microprocessor architect cannot use a logic simulator, because it merely simulates the basic signal logic of a design and does not provide a platform for running architectural simulation experiments. With increasing research interest in hardware and software interaction and compiler optimisation, an equivalent simulator for computer architects is necessary to derive tradeoff equations between the aspects of internal and external microprocessor and memory system design. Existing simulators are very fast but lack usability or have an extensive graphical interface and are too slow to simulate large models realistically. An Esprit project developed at the University of Edinburgh aimed to create a consistent set of tools for general system performance modelling [56] [57] [58].

Architect's Workbench Requirements

Most of current generation functional/behavioural modeling systems are extensions of hardware description languages described earlier. HDLs are essentially procedural, high level programming languages. These languages are structure oriented and require an early selection of components with behaviour embedded procedurally in a definite structure. Both the early binding to specific components and the nature of these languages restrict top-down development of designs. An architect would expect a functional/behavioural workbench to include the following features:

- A no-programming simulation environment.
- A Component Hierarchy used to create new architecture components and to extend the object library.
- A mechanism for inserting a component into an existing architecture, including linking between neighbouring components.
- An abstraction level hierarchy that prevents illegal connects between components.
- Consistency checking to ensure that objects have been instantiated with consistent parameters.
- A General Class Order– implementing a conceptual framework for computer architecture.
- A mechanism for each component class to cope with changes to internal and external parameters
- A Front Panel– to provide: the facility to enter input parameters and measure their effect on performance; a means of navigating through different levels of abstraction; the apparatus for architecture experiments.

- A component checking mechanism to ensure valid simulation runs.
- Output simulation trace and an optional animator that describes the internal behaviour of an architecture.
- Support for Performance Statistics

Commercial High-level Simulation Tools

An architecture's behaviour can be described as a reactive system [18] [19]. Reactive systems are characterised as owing much of their complexity to the intricate nature of reactions to discrete occurrences. Examples of reactive systems include most kinds of real-time computer embedded systems, control systems, communication systems, interactive software of varying nature and VLSI circuits. Common to all of these is the notion of *reactive behaviour*, whereby the system is not adequately described by specifying the output that results from a set of inputs, a description approach associated with common HDLs. Typically, such descriptions involve complex sequences of events, actions, conditions and information flow, often with explicit timing constraints, that combine to form the system's overall behaviour.

For example, STATEMATE, designed by i-logix [31], Inc. provides a working environment for the development of complex reactive systems. The computational parts of such a system are assumed to be dealt with using other means, but it is their reactive control-driven parts that attract the focus of STATEMATE. STATEMATE provides views of the structural, functional and behavioural models of a SUD (System Under Development) through *module-charts*, *activity-charts*, and *state-charts* respectively. All three representations are based on a set of simple graphical conventions. Its analysis capabilities include being able to step one unit through dynamic behaviour at the beginning and end of which the SUD is in some legal state. STATEMATE is a good general purpose tool to test

prototype systems. Unfortunately it is too general purpose and has not caught on as a popular architect's workbench. Its language primitives are not specific to architecture components, and visualising hardware interaction on such a general environment would lose the simulation experiment focus.

SES/Workbench [65] has dataflow/control semantics that are natural to behavioural design. It is naturally hierarchical because any node in the graph can be a subgraph node. The aliasing capability of SES/Workbench allows assignments of any number of logical nodes at one level of the hierarchy to a single node at a lower level of the hierarchy. Thus multiple logical communication links can be mapped to busses, or several logical operators can be mapped to a single functional physical unit.

The idea of the SES/Workbench is to enable designers to execute system-architecture designs at the behavioural level without being encumbered by the detail of structural representations. Architectural/behavioural designs can be executed independent of technology. Design evaluation can take place at each step of the resolution of the design in a true top-down process. Unfortunately due to its implementation, the SES/Workbench restricts the architect with its language. Again the semantics of SES force the architect to think of the implementation of the simulation and not simply the implementation of the architecture. The design becomes confused with the graphical details of the simulation language. Adding new components to an architecture involves programming a **C** description of the components' input/output characteristics. The system lacks a definition of abstraction levels, it is therefore difficult to identify the focus and the purpose of an experiment. A more recent version the SES/Objectbench tries to apply a more responsibility/collaboration approach, although it has not yet been released. SES is a good example of general purpose high level simulation techniques trying to help architects and compiler writers but it does not provide a transparent modelling environment. The architect should only be aware that he is building a behavioural architecture model and not be concerned with implementation details

of its simulation. The simulation environment should have a clear specification of existing component, an abstraction level hierarchy and a means to add and modify new architecture components and statistical monitors.

Procedural Simulation Tools

Engineers at Stanford University have designed a high-level simulator CARA (Compiler-Aided Research on Architectures) [23] that supports a top-down architectural analysis of embedded, custom applications. The tool characterises more than 50 instruction-set variants and allows data cache performance, register set size, and register allocation policy to be simultaneously evaluated for all the architectures. Designers have more flexibility because thay can tradeoff among high-level design constructs. Thus it was developed so that relative architecture performance could be evaluated before having to complete the machine specification at the lower level. CARA is a good example of an *Architect's Workbench.* In CARA an application is compiled to a standard intermediate form and then simulated using an architecture/cache simulator. The simulation input is the high-level description of the proposed architecture.

CARA estimates performance, for example, by calculating the number of machine cycles a given implementation would take to execute a basic block. A block is defined as a group of instructions that have a single entry and exit. Only the first instruction is a potential branch target instruction, so no instruction can cause a branch except the last. CARA was specifically useful for estimating the expected cache miss rates for a particular size and organisation of a cache. CARA complements the previous ISPS [4] and multi- level simulator Adlib-Sable [35] in that it simulates at the ISP level and yet allows specialised alterations to the executing architecture. Due to its implementation approach however, it lacks the agility for rapid architecture alterations through graphical interfacing and does not support a clear abstraction level hierarchy for the components.

Work at the University of Florida is currently developing a processor library for multiprocessor simulation [26]. Due to the level of complexity in simulating microprocessor behaviour they are seeking a method to simulate microprocessor based (DSP) multiprocessor systems accurately. They believe the sophistication of microprocessor devices has made it impractical to simulate their behaviour accurately with such traditional methods as Markov modelling and discrete event simulation. A library of DSP96002 processors enable uniprocessor and multiprocessor simulations, for example, to take place with single clock granularity. The simulation can keep an accurate count of clock cycles executed. The real-time execution rates can be calculated by scaling the number of clock cycles used to complete the simulation. The simulated processors update all the internal and external operations of the DSP96002 microprocessor such as register and memory updates associated with program execution.

Object Oriented CAD Tools

Clearly computer aided design tools intended to help create complete hardware/software systems, are themselves large complex software systems. Just as hardware systems must be designed using software design of CAD systems, CAD systems must take advantage of advances in software engineering to be successful and competitive. CAD tools and systems long ago stopped being adjuncts or spin-offs of hardware design efforts. They are a major investment whose capabilities determine the success of a hardware project and which require substantial engineering in themselves. Object Oriented Programming (OOP) technology offers substantial help in simplifying the design and implementation of CAD systems [78]. OOP has been discussed for decades, but has come into common use only within the last several years.

The notion of component *entities* has been used in *ADA* dynamic multi-level logic simulations [28]. These simulations focussed on Input/Output pin states

and hiding detail behind *black boxes* rather than using the *entities* to contain component detail in a hierarchy of abstraction levels.

Ptolemy, under development at Berkeley, clearly identifies some of the advantages of using an object oriented approach [9]. Ptolemy is a *heterogeneous* framework for the design and simulation of digital signal processors, communication systems, algorithms and communication strategies. Although the user has to do a lot of work in terms of connecting domains, because of the generality of the tool, it nevertheless demonstrates the agility, heterogenity and extensibiliy of using an object oriented language for the purpose of structuring a CAD environment. The tool is written in *C++*. The system is not complete in that it does not describe a graphical mapping that allows new components to be created and included into the component hierarchy. Furthermore, it does not define an abstraction level hierarchy that classifies the format of the abstraction level, for example, the detail of the items of data that are being exchanged during component cooperations. Ptolemy has suggested a framework for DSP and network simulation and has demonstrated some of the advantages of using OOP.

A similar object oriented framework is currently being developed at UMIST [19]. It shares the same focus as Ptolemy in that it attempts to create a library of useful objects that describe DSP and communication networks and algorithms. It defines how object orientation can decompose a system of interconnected processes expressed as a hierarchy of diagrams. In OO terminology, processes are defined as *agents* that encapsulate detail concerned with meeting responsibilities allocated to them. A job is done through collaborating with others. This work emphasises the use of inheritance and encapsulation to abstract data structures to express the responsibilities and interactions of architecture components described at different abstraction levels.

Both projects have still to develop simulated animation for their environments. Recent work at Chicago University [51] has shown the usefulness of visual RTL description. Although it was a simple model and was not supported by simulation

it demonstrated that there is a necessity to picture hardware data flow to explain the source of bottlenecks and poor use of architecture resources. This system describes the RTL abstraction level only and it has been suggested that the project may develop the animator to include OOP simulation.

1.4.3 Motivation for HASE

HASE started as a project [61] to investigate the feasibility of building a general purpose architecture simulator on an MIMD transputer network. In the course of this study, an Occam2 simulation of the 88100 Reduced Instruction Set microprocessor was developed on an MIMD T800 transputer Surface. A T414 graphics processor with gfx.library functions was configured to produce a visual presentation of the architecture's internal data flows, indicating, for example, the occurrence of read-after-write conflicts and providing useful information for performance analysis.

The simulation program was distributed over a grid of transputers using the software harness **tiny** in an attempt to reduce the simulation runtime. Simulated performance was verified by a direct comparison with the VAX accelerator, an 88000 system (courtesy of SUPERCOSMOS, Edinburgh Royal Observatory), that could run a sampling algorithm 10 times faster than a VAX machine, with an estimated performance of 8 to 9 MIPs. The inherent flexibility supported by Occam2 and the transputer environment was evaluated by attempting to alter the 88000 system architecture. Further performance improvement was achieved by developing a Front Panel Display to visualise internal data flow bottlenecks that were eliminated by optimising test program code.

Research carried out for this thesis adds a number of useful properties to the architect's workbench requirements. HASE aims to:

- support a library of useful objects to model components at different abstraction levels;
- provide a route for creating, linking, and inserting new objects into an existing architecture;
- provide a method to enforce architecture component parameter and linking rules;
- develop a multi-abstraction level simulation for the generation of a focussed event trace;
- visualise trace behaviour of a discrete event simulation model for the purpose of validating a model and guarding against an inefficient implementation;
- supply objects to monitor the statistics of component objects, for example throughput and utilisation for comparing architecture performance;
- support facilities to visualize the output state of interacting component objects at different abstraction levels. For example, animate the effect an instruction has on interacting components, and the effect these interacting components have on the instruction.
- provide support to follow a particular type of instruction through the system at different abstraction levels.

Figure 1–2 identifies the main services provided by HASE. The inputs to HASE are given by inward bound arrows, the outputs are denoted by outgoing arrows. The nodes represent a common collection of tools, programs and metrics.

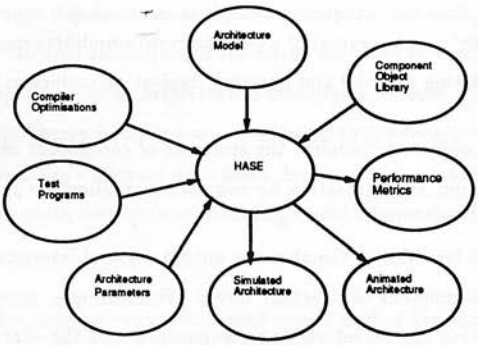


Figure 1–2: The Context of HASE

Chapter 2

Fundamental Concepts of Simulation

Overview

This Chapter addresses the basic concepts of a System, Discrete Event Process Based Simulation, and the Object Oriented Programming paradigm and provides a background to Graphical Interface programming environments.

2.1 Concept of a System

Central to any simulation study is the idea of a system [25] [14]. The term *system* can be defined generally as an orderly collection of logically related principles, facts or objects. When used in the context of simulation study, the term *system* generally refers to a collection of objects with a well-defined set of interactions among them. A classical example is the solar system. The planets and the sun form the collection of objects; gravitational force is one of the interactions among the objects in the system.

2.1.1 Definitions

Systems [29] [18] can be defined more broadly than as a collection of objects and interactions. For example, a system could involve all external factors capable of causing a change in the system. These external factors form the *system environment.* The state of a system is the minimal collection of information from which its future behaviour can be uniquely predicted in the absence of chance events. Since the inclusion of time implies that the state of a system changes, there must be some process or event that prompts this change. Such a process or event is called an *activity.* Activities external to the system are defined as *exogenous,* while activities internal to the system are referred as *endogenous.* Although it is convenient to distinguish between exogenous and endogenous activities, it is not always possible to do so. When defining a system it is not always apparent which factors are internal to the system and which are external. Therefore it is the change in system state induced by any activity that is of interest.

2.1.2 Classifying Systems

There are a number of ways to classify systems. An obvious classification distinguishes between systems that are natural and those that are man-made. For example, the solar system is a natural system, while a computer architecture system is man-made. Other classifications that can be used include continuous versus discrete, deterministic versus stochastic, and open versus closed.

Continuous versus Discrete Systems

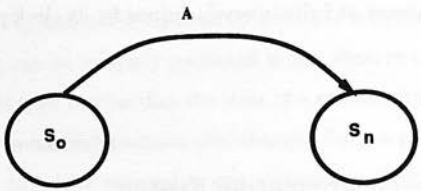
The terms *continuous* and *discrete* applied to a system refer to the nature or behaviour of changes with respect to time in the system state. Systems whose changes in state occur continuously over time are continuous systems. For example, a continuous system may describe an analogue circuit, e.g. an amplifier, feedback

loop or phase lock loop, in which the rate of change is determined by a set of differential equations. Systems whose changes occur in finite intervals, or jumps, are discrete systems, while in some hybrid systems some state variables may vary continuously in response to events while others may vary discretely. A computer architecture can be described by a set of interacting discrete subsystems. Changes to the system occur at finite intervals defined by its clock period.

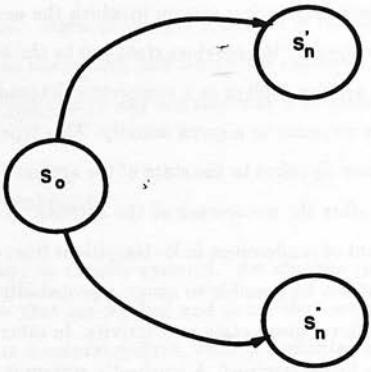
Stochastic versus Deterministic Systems

A *deterministic* system is a system in which the new state of the system is completely determined by the previous state and by the *activity.* Considered in another way, a given system evolves in a completely deterministic manner from one state to another in response to a given activity. This type of system is depicted in Figure 2-1a, where S_0 refers to the state of the system before activity A and S_n refers to the state after the occurrence of the activity. A *stochastic* system contains a certain amount of randomness in its transitions from one state to another. In some cases it might not be possible to assign a probability to the state that the system will assume after a given state and activity. In other cases these probabilities are known or can be determined. A stochastic system is shown in Figure 2-1b, where S_n' and S_n'' are two possible states that the system can enter after the state S_0 in response to activity A. Thus a stochastic system is nondeterministic in the sense that the next state cannot be unequivocally predicted even if the present state and the stimulus (activity) is known.

A computer architecture is described by both deterministic and stochastic systems. For example, high level architecture components may be described stochastically, while lower level components can be described deterministically.



a) A Deterministic System



b) A Stochastic System

Figure 2–1: Deterministic versus Stochastic System

Open versus Closed Systems

A *closed* system is a system in which all state changes are prompted by endongenous activities. In contrast *open* systems are systems whose states change in response to both exogenous and endogenous activities. As it is difficult to distinguish between endogeous and exogenous activities it is likewise difficult to clarify whether a system is open or closed. A computer architecture involves exogeneous activities, for example external interrupts from I/O, and endogenous activities, e.g. fetching, decoding and executing a program. A computer architecture is therefore defined as an open system.

2.2 Discrete System Simulation

Discrete event simulation was originally designed to solve complex queueing theory problems, which arise when a system is interpreted as a system of interacting queues. The development of simulation is most useful when applied to systems where resource utilisation is at a premium, hence the usefulness of its application for VLSI design [45]. Statistical representation of a system's behaviour is limited to considerations of gross throughout; the details of, for example, requests for services through a queue, are not amenable to close description by queueing theory.

Originally the statisticians' view of simulation as an elaborate Monte Carlo method predominated, but it has now been recognised that discrete-event simulation can reflect more closely specific details of a designed system, besides its statistical properties. This ability becomes more important as the design becomes more complex. The modern, structural view of discrete-event simulation is therefore concerned with issues outside purely statistical interest, for example, algorithms, methodology, and control structures.

Terminology

The characteristics of an *entity* are referred to as attributes. The collection of entities and attributes for a given system is referred to as the *system state*, and is generally expressed with reference to time. Any process that changes the system state is referred to as an *activity*. The occurrence of such a change at time t is referred to as an event. An event may for example, be the change of value of some attribute, the creation or destruction of an entity, or the initialisation or termination of an activity.

Zeigler [79] laid down the first abstract specification of a Discrete Event System (DEVS). In discrete event simulation the value of the time increment is not stipulated in advance; it is determined individually for each time step, based on the component actions in the model. Events are determined by the sequence of each entity's starting and finishing activities: the global actions of the simulation are the persuance of these sequences for several concurrent entity flows. Discretisation of time is thus implicit in the system itself, rather than being explicitly imposed by the simulator. The simulation program concentrates on the events interleaved between activities, and is relatively unconcerned about the periods of active work, leading to a distinctive *inversion* of concern. An "activity" does not necessarily imply that an entity is performing in any particular way: being gainfully employed or simply waiting for an opportunity for such an employment both qualify as activities since they extend over time.

A discrete-event simulation consists of a parallel flow of entities interacting with resources during activities. For the paradigm to be successful, every event-notice must have a predictable occurrence time when being inserted into the FES (future-event set). Not all events are directly predictable however, some depend on predictable events of other entities, or maybe some configuration of the model, but the first discrete-event paradigm is valid as long as each activation can be assumed to occur at *some* event.

2.3 Simulation Languages

Given the sophistication of available general-purpose languages, it can be argued that *simulation languages* are unnecessary. A discrete event simulation can be easily written using a procedural language, for example, a Monte Carlo simulation in C or object-oriented C++ [70]. The main difference is that the simulationist provides his/her own ES (Event set) strategy which would typically consist of an FES (future-event set) executive program to drive the simulation along. Convenience alone demands that the executive program should be part of the simulation language.

2.3.1 Programming Representation

In conventional procedural languages, a calculation program is written in a general sense, in terms of variables whose values will be actualised at run-time, through a *read* statement. In this way a program represents a mathematical formula like $f(x)$, with x the unknown variable. The program thus describes the active procedure, f, which defines the way in which the eventual x will be operated upon, while the data consists of passive x instances. The function $f(x)$ produces a value, whatever x is. This approach to modularising programs enables libraries of functions like f to be built up and invoked when required.

A mathematical description of system concepts is too limited for practical use, in that simulation requires more than mere substitution in a prescribed formula; for example, the structures of the real system must be mirrored in the simulation language. In a simulation program the executive is applicable to the total set of discrete-event simulations. However in simulation the model must be capable of change. The "data" which instantiates a specific model of interest consists not of

passive values, but program modules of different "classes" that require activation. In addition it may be required to specify data-structures and subprograms.

Simulations require a different kind of relationship between general and specific parts; the specifics are invoked by the general, which is the opposite case to conventional, algorithmic programs, where general library routines are invoked with special parameter values. A general-purpose language with sufficient flexibility for simulation is SIMULA [55], which initially started as a simulation language, but is now used as a language for writing packages, for example, DEMOS (Discrete Event Modelling on Simula) [7]. Other **object-oriented** languages inspired by Simula are capable of supporting simulation.

2.3.2 Object Oriented Programming

Simula is not only important as a simulation language, it is also the first object orented programming language. It was developed in Norway in the 1960's by Dahl, Nygaard, and their colleagues. The fact that Simula's main application area was simulation gave rise to the emphasis on the "linguistic anthropomorphism" that is often regarded as an essential part of the object-oriented style.

Object-oriented Software

Many modern statically typed languages include object-oriented features. Some of the better known are:

- **Trellis/OWL**, an early statically typed object-oriented language, building on ideas from CLU [54].
- **Modula-3**, a variant of Wirth's Modula-2 (descended, in turn, from Pascal) with automatic storage management, objects and concurrency [49].

- C++, an object-oriented variant of C, probably the most widely used object-oriented language [70].

- Eiffel, a commercial object-oriented language [54].

In addition to literature on object-oriented design and the specifics of various object-oriented languages, object-oriented programming has spawned a growing field of theoretical study, much of it concentrating on the problem of providing sound static type systems that are flexible enough to capture the full range of idioms offered by untyped object-oriented languages.

Characteristics

The HASE prototype takes advantage of three main characteristics of the object-oriented programming paradigm: encapsulation, inheritance and polymorphism [75]. These three characteristics are given below:

- **Encapsulation** refers to the practice of drawing "abstraction barriers" around collections of code and data. The encapsulated entities cooperate freely among themselves , but they interact with the outside world through a narrow and explicitly specified interface. Advantages include clean separation of concerns and mechanisms between various parts of a large program; hiding of private or irrelevent information; explicitly visible interfaces between components; and support for the notion of "programming by contract". In conventional programming languages large scale encapsulation is achieved through the use of abstract data types and module systems. Object-oriented programming languages go one step further by requiring that *every* piece of state is encapsulated in some object.

- **Inheritance** provides a convenient way of factorising the *implementation* of a data type using a class hierarchy. At each level of the hierarchy, the

behaviour associated with the attributes introduced at that level is described. The behaviour of an element of some type in the hierarchy is then obtained by composing all of the descriptions for the behaviour of the types above it.

- **Polymorphism** describes the relationship between two or more objects that respond differently to the same stimulus. For example, in HASE a cache and a memory object will respond differently when asked to read a byte of data. Similarly, a cache object at the RTL abstraction level will behave differently to a cache object modeled at the ISP, when asked to write a byte of data.

Although each of these is coherent and useful in isolation, the term "object-oriented programming" is normally reserved for situations where all three are in play.

Terminology

An *object* is represented by some private memory and a set of operations that describes some behaviour [8]. Objects that share the same behaviour are said to belong to the same *class*. A class is a generic specification for an arbitrary number of similar objects. A class can be used to build a taxonomy of objects at an abstract, conceptual level. A *message* consists of the name of an operation and any required arguments. When an object receives a message, it performs the requested operation by executing a *method*. A *method* is the step-by-step algorithm executed in response to receiving a message whose name matches the name of one of its methods. While the message consists of the name of a method and its required arguments, a *signature* is the name of a method, the types of its parameters, and the type of the object that the method returns.

An object that behaves in a manner specified by a class is called an *instance* of that class. All objects are instances of some class. Once an instance of a class is created, it behaves like all other instances of its class, able upon receiving a

message to perform any operation for which it has methods. It may also call upon other operations on its behalf. A program can have as many or as few instances of a particular class as required. Inheritance is a useful mechanism for factoring out common useful behaviour. Classes that are not intended to produce instances of themselves are called *abstract classes*. They exist merely so that behaviour common to a variety of classes can be factored out into one common location, where it can be defined once and reused again and again. A *concrete class* inherits the behaviour of its abstract superclass, and adds other abilities unique to its purpose. It may need to redefine the default implementations of its abstract superclass. Concrete classes are fully implemented classes which create instances of themselves to do the useful work in a system.

2.3.3 SIMULA

Simula started life as a preprocessor to Algol 60 for simulation programming, then branched out as a package-writing language [7], and currently sets the standards for a new batch of languages embodying object-oriented concepts. Simula added the *class* concept, coroutines, references and record structures to the Algol 60 base.

Objects come into existence by a call of **NEW** followed by the class name, followed by an optional list of parameter values by which the attributes may be initialised. The activity sequence of the object is entered, until the sequence comes to an end, when the object is regarded as terminated. Besides having many objects of the same class existing simultaneously, many simulations demand objects which partially resemble one another. Two classes may share some but not all of their features. Simula provides a mechanism for defining *subclasses* of a class where specific differences between subclasses can be defined without repeating the features which they share in a common declaration. In this way Simula implements the characteristic of *subtyping* through the mechanism of object *inheritance*.

2.3.4 Sim++ and Virtual Time

It is anticipated that a future version of HASE will be implemented in Sim++ [40] because it is fast and supports distributed capabilities. Sim++ was derived from DEMOS and C++ and facilitates an implementation of *virtual time* [43]. Virtual time reduces the amount of waiting necessary between synchronising and collaborating entities.

Sim++ is a superset of *C++*. It is a process based, discrete event driven, high performance simulation language and implements virtual time, so that it can be distributed over a large MIMD system as a network of Workstations. The last consideration is very important because this application will potentally need large amounts of memory and many computations in order to obtain useful results. Virtual time is analogous to virtual memory; it is completely transparent to the programmer. The major application of virtual time is as a synchronisation mechanism for distributed simulation. Rather than waiting to synchronise with other activated processes, each process excecutes without regard to whether there are synchronisation conflicts with other processes. Whenever a conflict is discovered, the offending processes are rolled back to the time just before the conflict. A **rollback** is similar to the philosophy of page faulting; because a message is received by a process with a low timestamp, the receive time of the message is very likely to be in the recent past and so it is assumed that the amount of rollback will cause a tolerable amount of unnecessary computation.

From the programmer's view the global clock always progresses forward at an unpredictable rate with respect to real time. However from the implementer's point of view, there are many loosely synchonised virtual clocks, one per process, occasionally jumping backwards. The advantage of virtual time is that every process is free at any time to send a message to any other process. In comparison to Lamport's clock condition [45] it is void of starvation and deadlock. It is faster and more scalable than Schneider's algorithm [64] because it does not have as

many synchronisation overheads when, for example, distributed over thousands of processes.

2.4 Diagrammatic Representation

The old adage "a picture saves a thousand words" seems to hold true for simulation as in other technical disciplines. This section considers the diagrammatic approach to system representation.

2.4.1 Attraction of Diagrams

It is from their two-dimensionality that diagrams gain over programs for expressing system phenomena. A diagram serves as a helpful bridge between vague external ideas and rigorous internal programs. A diagram may be more readily assimilated by non-technical members of a project team, for example, who may not grasp the intricacies of a program; a diagram thus helps in the dissemination of ideas about the model, enabling some consensus about the degree of detail and realism to be attained. A diagram is also a convenient way of publishing the broad outline of a simulation without the interference of syntactic peculiarities of a particular language.

2.4.2 Software Graphics

There is a variety of graphic programming environments [42] that allow a user to manipulate graphics functions, for example, SRGP (Simple Raster Display), Athena Widgets, and more recently *Motif*. The HASE prototype uses the Athena and Motif environment to implement its model and architecture animation facilities. To be as widely distributed as possible Motif was based on **X** Window System

Version 11 and designed to be used on X11 systems. X is a network-based, graphical windowing system developed at Massachusetts Institute of Technology [77].

The X server contains all the display specific code. It keeps track of all input from such devices as the keyboard and the mouse and deals with all requests from any X clients that are running. At the lowest level, X's client side includes the Xlib interface library. This library, which is part of the standard X release, provides a set of device-independent procedures for performing graphics and other window-related maintenance. Xlib passes these device dependent calls to the X-server, where they are translated into appropriate machine-specific graphics commands.

The procedures in Xlib are low-level; developers typically construct high level libraries, called tool kits, to make programming easier. The tool kit includes XT intrinsics and a set of graphical user-interface components. The XT intrinsics are built from the Xlib procedures, but at a higher level. Motif user-interface components were built upon XT intrinsics.

In X the user interface components are called "widgets" and "gadgets". A widget is a graphical user-interface component that has its own window. There is a variety of widget classes, for example *command* widgets that call user defined functions when selected, *dialog* widgets that display boxes for data collection and bulletin board widget classes that allow icons to be placed and managed within the widget's window. A gadget is similar to a widget except it does not possess a window of its own. and must be located in some other widget's window. A typical application program, or X client, will use several Motif widgets or gadgets as its user interface.

In addition to the X server, Xlib, and the tool kit, most X installations have a window manager. The window manager is a client program to manipulate client windows. For example, it handles actions like moving a window, resizing windows, changing the stacking order of overlapping windows, and representing a window by an icon.

2.4.3 Comments

This chapter has introduced the concept of a system, the principles of discrete event simulation, the terminology of object oriented programming and has provided a background on the simulation languages and graphic environment of HASE. Assuming an understanding of the fundamental concepts underlying HASE, Chapter 3 proceeds to explain the operation of the HASE prototype environment.

Chapter 3

The HASE Design Environment

3.1 Overview

HASE is both a design environment and a simulator. In the design phase the user creates a design through a graphical interface by composing together architectural objects (arithmetic units, caches, etc). During this phase the system checks the interfaces between objects to ensure that only compatible objects are being joined together. Subsequently the user may wish to define a collection of objects which constitute a processor, for example, as a higher level object to be used in a multiprocessor system. Therefore, each object is abstracted to model the behaviour of a submodel containing lower level components. To develop, manage and visualise the functional behaviour of the component object hierarchy, HASE uses four main components: an *Object Editor*, an *Architecture Editor*, a *Code Generator* and an *Animator*.

The Object Editor is responsible for creating and storing objects to describe the functional behaviour of physical hardware and software components. The Architecture Editor manages a hierarchical menu of software components that model the hardware and software components of an architecture. The Architecture Editor facilitates the selection, setting of parameters and integration of software components. An architecture model is saved as a graph of connected nodes, where the nodes represent software components and the arcs represent messages between

objects. The Code Generator is responsible for parsing the connected graph of nodes, collecting parameters for each component object, and building the source code for the simulation program. The Animator provides graphical presentation of an architecture's performance data and a means to visualise the simulation event trace and the architecture's behaviour during benchmark executions.

Figure 3–1 summaries the five main phases of HASE: Object Creation, Architecture Creation, Simulation and Animation and Performance Analysis. Each iteration of an object and architecture creation phase uses feedback from the Performance Analysis phase. This section describes the first four phases and shows how HASE provides monitors to capture a simulated architecture's performance characteristics.

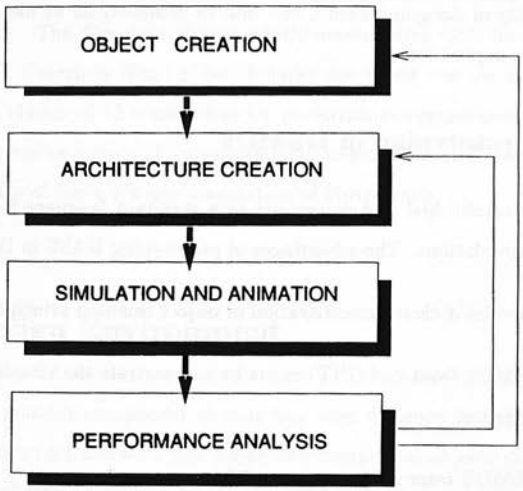


Figure 3–1: HASE Environment

3.2 Programming Environment

The factors affecting the choice of simulation language include the simulation algorithm, its programming paradigm, its syntax and its portability. *Sim++*, described in Chapter 2 supports a distributed simulation language, is based on *C++*, its syntax is popular and portable, and therefore it is considered appropriate for a future implementation of HASE. Although a number of *Sim++* simulations have been carried out to demonstrate its use on a network of Sparcs, the environment of HASE has been demonstrated in DEMOS. The objective of the project described in this thesis was not to implement a full working version of HASE but to examine the feasibility of designing such a tool and to demonstrate its use on a number of architecture experiments.

3.2.1 Prototyping in DEMOS

DEMOS is established and recognised as a standard language for implementing high level simulations. The advantages of prototyping HASE in DEMOS include:

- It provides a clear demonstration of object oriented principles.
- A DEMOS front-end (PIT) exists to demonstrate the visualisation of object inheritance.
- A DEMOS *trace* animator is available.
- Familiarity in the public domain, so it was not necessary to spend time on support problems, which were initially present with *Sim++*.

The disadvantage of using DEMOS is that it lacks *C* type ***bit-wise*** operations to express lower level (RTL) simulations. Furthermore, DEMOS does not implement a distributed simulation algorithm [11]. Previous experience indicates that

DEMOS is incapable of running large architecture simulations without becoming intolerably slow and memory intensive. Nevertheless for the purposes of a prototype it was not necessary to run large RT level simulations, but rather to demonstrate design issues and useful aspects of the operations of HASE.

3.2.2 Hardware Resources

Workstations

DEMOS and Sim++ simulations are currently running on Sun4 Workstations, on which Athena, SRGP and *Motif* are supported by X11. The *GSS* environment currently works on Openwin3 to allow the simultaneous execution of Xwindows and Suntools. The Graphics Editor which manipulates GSS functions is also written in C. Currently Sim++ benchmarks are being run on a network of 8 Sun4s and a cluster of 12 transputers for performance comparison. Future work will include running large architecture simulation programs and rewriting them to take advantage of Sim++'s implementation of **time warp**.

3.3 Design Environment

A library of reusable component objects has been designed to be configured together to support a framework into which new component objects can be inserted and connected into an architecture using standard synchronisation procedures. By specifying a set of *link refusals* and a component icon hierarchy, HASE ensures that architectural constraints between this library of component objects are not violated. This section discusses the four issues central to the operation of HASE: Component Hierarchy, Object Creation and Architecture Editing, Code Generation.

3.3.1 Component and Abstraction Hierarchy

Objects can be created *ab initio*, by composition of lower level objects, or by inheritance from generic objects already in existence in the hierarachy.

Component

If a number of classes share similar data and function members their properties can be factored out and made into an *abstract* class (Figure 3-2(a)). For example, the function units of an architecture have many properties in common, e.g. pipelines, functions to read operands and write results to register files, and these can therefore be encapsulated into an abstract class *component*. The functionality and data associated with class *component* are inherited by the *integer* and *floating point* and *data* units of the architecture.

Composition

A component class has an associated abstraction level. HASE currently includes three main abstraction levels, PMS, ISP and RTL as defined in Chapter 1. A submodel represents a number of interacting objects. For example, the *processor* class (Figure 3-2b) is *composed* of *fetch*, *decode*, *execute* and *writeback* objects. The *processor* class is not abstract, although the subclasses of *processor* inherit its data and function members. If it is instantiated it executes function members that simulate the functional behaviour of its subclasses at a higher level of abstraction.

Raising a component's abstraction level reduces the level of detail of the information that is exchanged between interacting component objects. This is a useful facility when focussing a simulation experiment on a particular aspect of an architecture. For example, at the PMS level HASE exchanges a random sequence of memory references. The detail concerning the actual address or the data returned from a read or write is omitted from the simulation. At the ISP level,

addresses and data are represented by decimal values, and results of arithmetic operations are computed and written to a *register* class. At the RTL level HASE exchanges actual binary numbers, and manipulates *bits* using shift instructions to perform arithmetic and logic operations.

Multi-level Simulation

HASE achieves multi-level *component* simulation by assigning each submodel to an abstraction level number corresponding to the PMS, ISP and RTL. The choice of abstraction level for a selected architecture component will depend on the purpose of the simulation experiment.

In addition to simulating components at varying levels of architectural detail, an architecture experiment can support multi-level *instruction* simulation. A component class is instantiated at abstraction level A_l for opcode p_i and instantiated at abstraction level A_{l+1} for opcode p_j. For example, if the purpose of a simulation experiment is to examine the RTL implementation of an "immediate" floating point multiplication instruction, the floating point function unit class instantiates its multiplication pipeline class at an RTL level and executes the remaining operation types at the higher ISP level. All binary data exchanges occurring during the simulation experiment correspond to immediate floating point multiplication instructions.

3.3.2 Object Creation

A class is a template for the functional behaviour of a hardware or software component modelled by HASE. An Object is the instantiated class. A description of the hierarchy associated with objects has been discussed in the previous section. This section lists the general attributes associated with a component object, and explains how a graphical front-end for object creation was prototyped in PIT (Process Interation Tool).

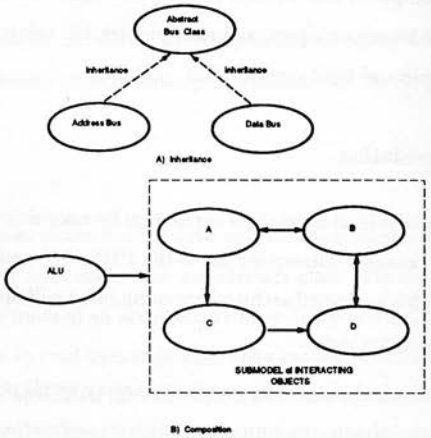


Figure 3–2: Object Hierarchy

Object Facets

Designs are created by composition of icons. At each level of the hierarchy there is a set of menus from which components can be selected and inserted into the design. Each component is represented in software by a multi-faceted object. For each object the following can exist, each represented in a unix file with the appropriate extension:

An icon: (.icon) each object in the system is represented by an icon. The icon is created using an X-windows icon editor which produces a postscript file.

A textual definition: (.txt) the textual definition of an object describes its *behaviour* (in terms of *entity entity collaborations*, *computations*), and inherited data and function characteristics of the object's class.

Simulation code: (.sim) written in Sim++ and/or DEMOS.

An interface definition: (.gdl) the interface definition allows the system to check that objects which the user wishes to link together are compatible. Compatibility may be affected by parameterisation of objects instantiated into the design from the menus. A generic cache object, for example, may need the word length, block size and degree of set associativity to be specified.

A formal definition: (.def) a planned future of extension of the HASE is the incorporation of formal definitions of objects to allow formal verification techniques to be used.

VHDL code: (.vhd) a second planned extension to HASE is the incorporation of VHDL descriptions of objects to provide a possible route to implementation of designs.

Process Interaction Tool

The Process Interaction Tool was developed by Eric Barber of BNR, as a graphical interface to DEMOS [5]. The purpose of PIT is to allow a designer to create or edit a component object from a menu of predefined icons that represent DEMOS simulation primitives. Icons symbolising nodes and links of an *activity diagram*, as defined by Birtwhistle [7], are selected from the PIT menu and inserted, linked and parameterised to define a DEMOS entity to model the interactive behaviour of an architecture component at a given abstraction level. The activity diagram is then saved under the physical component's name and inserted into a submodel which belongs to the appropriate abstraction level.

PIT includes the definition of *abstract* object procedures that are inherited by the concrete component classes. These inherited procedures for data exchange in a pipeline are *read* and *write*, and *caches* and *busses* for memory data and address exchange. Figure 3–3 shows a processor entity and its associated tree of abstraction levels. The processor class is described by three separate entity

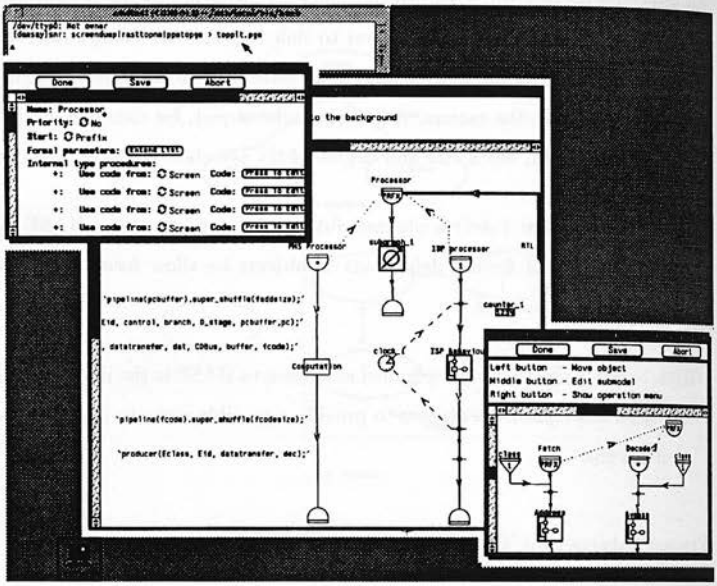


Figure 3–3: Process Interaction Tool for Component Creation

control paths defined between a start and terminate node. The PMS level models a processor as a server entity which is held in the event queue for a computation period and randomly generates read and write operations to an external bus class hierarchy. Within the ISP processor submodel further submodel trees of activity diagrams are defined, e.g., *fetch* and *decode* entities. The submodel tree for all components available in the architecture editor is defined in a Graphics Description Language (GDL) file. Only non-parent entities are defined as class nodes in GDL, because they are concrete and can be instantiated and linked. The example in Figure 3–3 shows a selection of instrumentation objects, for example the counter which counts the number of instantiations of a particular entity class during a simulation run.

3.3.3 Architecture Creation

The Architecture Editor provides a facility to browse the GDL object menu at the appropriate abstraction level and to select and insert new objects into an existing architecture without reprogramming the simulation source code. The Architecture Editor is prototyped in the GSS environment [5]. The purpose of the Architecture Editor is to raise the level of detail above entity activity diagrams defined in PIT and label each DEMOS or Sim++ object with a meaningful architecture name and icon. The language defining architecture nodes and links was prototyped in GDL, further details of which are provided in Chapter 4. In GDL a component object is represented by a node, with parameters, icons and submodels, ports and link refusal definitions. Each node is defined within a submodel graph. The submodel graphs are represented by a menu of icons, where each icon represents a node. Each submodel of a node defines an abstraction level.

Managing the Abstraction Level Hierarchy

HASE defines in GDL a unique menu of icons to display objects that can be instantiated for each submodel. A set of submodels defines an abstraction level. For example, the PMS abstraction level is defined by the set of submodels: [*processor*, *memory*, *link*, *switch*, *control*]. The *processor* submodel can be further defined at a lower ISP abstraction level, for example by a set of submodels: [*cache*, *bus*, *fetch*, *decode*, *sequencer*, *execute*, *writeback*]. The *cache* could be decomposed into a lower RTL level of abstraction in the same manner. Therefore each submodel has a set of icons that can be inserted, linked, parameterised and instantiated. The GDL description of the hierarchy enforces architecture coherency through the notion of *link refusals*. For example, at the PMS level a *processor* must not be connected to a *memory* subclass, unless via a *link*.

A node N_i defined within a submodel S_l can link with a node N_j in submodel S_{l+1}. In a submodel menu there exists an *entry* and an *exit* node. The entry

node ensures that the abstraction level of the transmitting node is lower than that of the receiving node. The token exchanged between the cooperating entities is translated to the higher abstraction level. If they are the same, then no translation is necessary; if the receiver is lower, then the *exit* node converts the format of the token to a lower abstraction level.

Edit Operations

This section describes the main operations available to setup a simulation experiment using the HASE prototype Architecture Editor.

- **Inserting and linking objects:** Figure 3–4 illustrates an architecture edit linking a pair of cooperating objects. The diagram shows 3 individual icons to represent: a cache, fetch and decode stage object. The cache icon is selected and linked to the fetch stage object. In this example, the cache object can make three different types of links: Address, Control and Data. Selecting a Data Link will attach the connection to a data bus object. Further details of how a link is created are provided in Chapter 4.

- **Parameterising objects:** Figure 3–4 shows an example of selecting and setting the decode object's parameters. In this example, the MC88100 is selected and the decode object will take a clock cycle to decode each instruction.

- **Creating a submodel:** Each icon has an option in its *pull-down* menu to *Edit Submodel.* A submodel can be edited to display its object components instantiated at the next lowest level of abstraction. Figure 3–5 shows an example of the ISP *Instruction* unit, expanded to display a submodel of its RTL pipeline stages. Each submodel has its own menu of icons. If two icons are to be connected between different submodels, then it is necessary to connect them with *exit* and *entry* icons. Each submodel therefore has a

number of external ports it uses to exchange input and output data to and from the submodel.

- **Loading and storing architectures**: After a component object, or version of an architecture is edited, the main menu provides a option to save or cancel the changes.

- **Instrumentation**: In order to assess the performance of a system it is necessary to collect data about the type and frequency of selected events during program execution. Monitor icons can identify, for example, wasteful stalls in the architecture, caused either by poor sequencing of code or by poor distribution of hardware resources. A menu of counters and various types of component monitors can be selected from the icon menu at each abstraction level to generate useful statistics on, for example, processor utilisation, throughput, frequency of data dependencies and number of pipeline stalls.

3.3.4 Code Generation

The HASE prototype, provides a main menu option to *Generate DEMOS*, simulation source code. The class declarations are retrieved from the component library and written to the simulation's source file. The HASE prototype's code generator was designed to prove the concept that a class declaration and its object parameter can be automatically linked and instantiated into a single source file. Therefore, the HASE prototype does not provide a functionally complete DEMOS code generator. The prototyped code generator only generates a DEMOS shell of the simulation's source code. The user is expected to edit the DEMOS main program of this shell, and instantiate each component object that is required by the simulation.

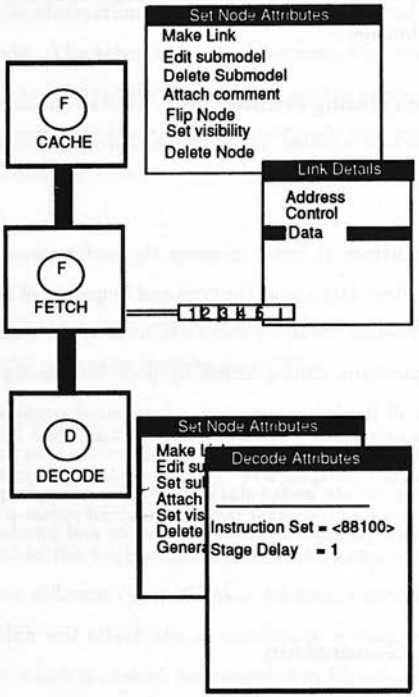


Figure 3–4: Link Creation

3.4 Simulation Phase

3.4.1 Input/Output

The Architecture Editor is responsible for writing an object's architectural parameters to a Datafile. It does this during its code generation phase. Before the code generation phase, objects for examination are selected. During the simulation phase the event trace and output states of selected objects are written to output files. Each object is responsible for the management of its own state and perform-

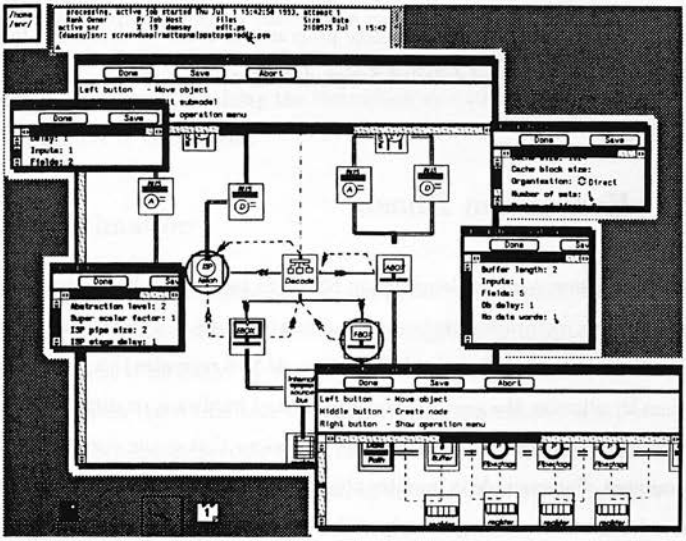


Figure 3–5: An Architecture Edit

ance data collection. In addition to display state files, performance statistics are collected and later presented as performance graphs.

3.4.2 Operation

The simulator is intended to be used in two modes: (a) with visualisation, (b) without visualisation. In mode (a) the graphical image of the currently selected part of the design is animated by the output from the simulator and the rate at which the simulation proceeds is determined by the user, in order that s/he can follow the animation on the screen. In (b) the simulation proceeds at maximum rate in order to allow performance measurement results to be gathered during the execution of complete (and potentially very large) programs. Thus the user can run the simulator slowly in visualisation mode using short test code sequences, in order to check that his/her architecture is performing correctly, and then run the

simulator in fast mode using complete programs in order to determine the effect on overall performance of detailed design decisions.

3.5 Evaluation Phase

When processors were implemented in SSI/MSI technology it was possible to attach hardware monitoring probes to appropriate points in a system in order to collect performance and debugging information. HASE recreates this facility for VLSI devices by allowing the user to attach simulated hardware monitoring devices, to alert the architect to certain patterns of behaviour that occur during a simulation experiment. For example, a monitor object may alert the architect to *1st 2nd* and *3rd* order conflicts, or may simply estimate the throughput and utilisation of a function unit pipeline.

3.5.1 Statistics and Graphic Visualisation

DEMOS provides classes to measure the average %CPU utilisation of an entity or the average time an entity waits in slave and master queues during cooperation. HASE includes *simulator* and *architecture* classes responsible for measuring the efficacy of the simulation model and an architecture's performance. The former class includes a set of *objects* (available in the hierarchical menu) for returning available stack space, number of page faults and total virtual memory required for simulation. Finally the *simulator* class is responsible for updating the total number of simulated instructions executed per second.

The responsibilities of the *architecture* class include recording statistics for the purpose of summarising architectural efficiency. For example, counting the number of stalls in a pipeline or the total utilisation and throughput of a function unit or the average (CPI) Cycles Per Instruction figure for the simulated processor. The

event trace identifies different patterns of instruction dependencies, for example, the occurrence of an exception before a control transfer instruction or the number of data dependencies that delay the instruction execution pipeline for more than a given number of clock cycles.

3.5.2 Animation

A current problem with contemporary high performance simulation tools, for example, the SES Workbench [19], is that the design and animation of an architecture experiment is obscured by the implementation of the simulation model. In HASE a separate trace animator visualises the implementation of the simulation model. An independent process is run to parse the simulation trace and animate the mechanics of the simulation model, in terms of entity scheduling, resource blocking, and entity synchronisation.

Trace Animation

The trace animator identifies which entities are interacting, or competing for common resources. A standard event trace can be toggled on or off at defined break points in the simulation source. A trace parser is responsible for reading a DEMOS trace and calling the appropriate window management functions. The animator's frontend includes options to *load* the trace, *play*, *rewind*, *step forward* and *step backwards*. The trace animator determines the workload of an entity during a simulation, in terms of scheduling, blocking and synchronisation. The trace animation example in Figure 3-6 shows two "FETCH" entities competing for a read and write port, modelled as a DEMOS resource class. $FETCH2$ is blocked in the event **ready** queue, waiting for $FETCH1$. Such an implementation could model an architecture's writeback sequencer.

HASE can provide information for the user to improve the efficiency of the model. It can show if an entity is blocked unnecessarily, waiting in the synchron-

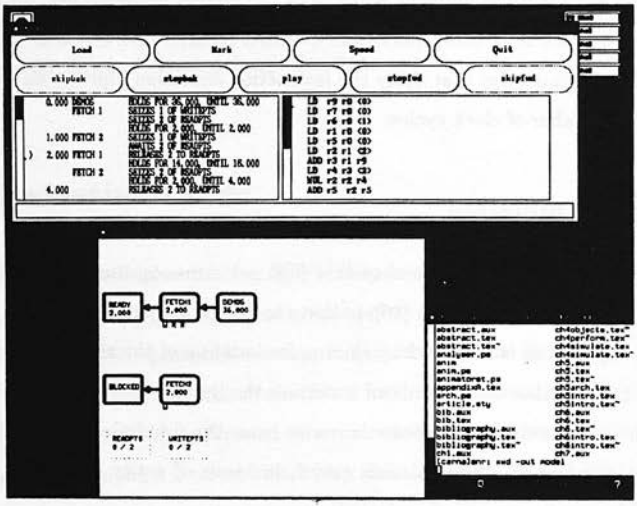


Figure 3–6: An Example of the Trace Animator

isation or resource queues, rather than computing at the head of the **ready** queue. If detail of these entity interactions is irrelevant a user can reduce the memory requirements of the simulation by abstracting the responsibilities of the entity higher up the component hierarchy.

When developing a distributed simulation interacting entities are *clustered* onto the same workstation. The trace animator can identify which entity pairs queue for the most synchronisations. Communication/computation costs of this nature were encountered with the MC88000 transputer simulation on a Meiko transputer network [61].

Architecture Animation

Architecture animation involves three interactive displays: the processor architecture, the front panel of the trace animator, and the display of the animated simulation trace. The trace animator front panel updates the current trace line

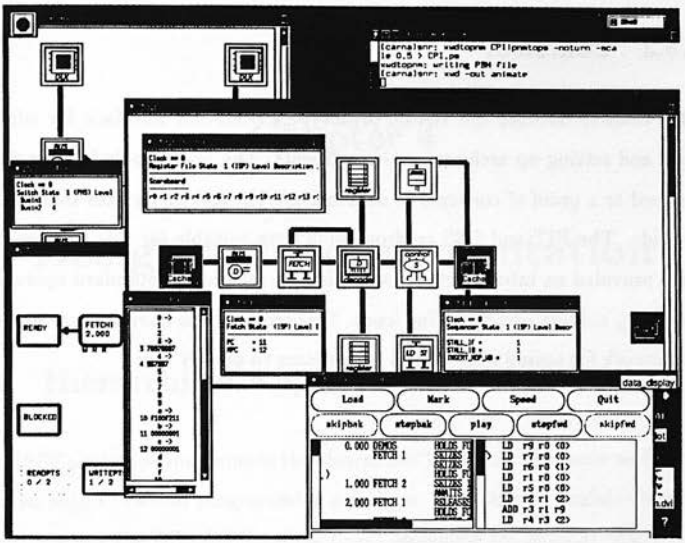


Figure 3–7: Example of an Animated Architecture

and identifies the new position of the program counter as the simulated processor steps through its program execution. Figure 3–7 shows an example of an animated architecture and trace display. The purpose of the architecture animator is to manage the simulator's output, visualise architecture performance data, and display the internal state of a selected architecture component. The trace and architecture animator have been *piped* together. As the architecture animator steps through the next instruction sequence, the trace animator is instructed to select relevent events, entities and resource interactions implementing the model of the animated component. An important feature of the architecture animator is its ability to *zoom* into a component's lower level of detail. A set of *Motif* callback functions ensures that the appropriate submodel is displayed. A linked list of *client data* referring to a component's internal state is maintained and updated on each step of the animated clock.

3.5.3 Comment

This chapter outlines the HASE prototype's front-end interface for editing objects and setting up architecture experiments. The prototype's front-end was developed as a proof of concept, to demonstrate the sort of services that HASE can provide. The PIT and GSS environments were suitable for this purpose because they provided an infrastructure for setting up menus and standard operations for selecting, moving and removing icons. Futhermore, this environment supported a framework for saving objects and parameters to output files.

Chapter 4

Design and Implementation

4.1 Hierarchical Approach

The HASE prototype implements the abstraction layers of a computer architecture using an object-oriented programming paradigm. This chapter explains how class inheritance and the DEMOS (Discrete Event Modelling On Simula) programming environment is integrated into HASE to support a framework for multi-abstraction level simulation. Furthermore, it describes how HASE was designed to specify component objects, edit architectures and evaluate and visualise an architecture's performance.

4.1.1 Classifying Abstraction Levels

The design of an architecture can be explored at different levels in the HASE hierarchy, likewise it is possible in the simulation phase to choose the level at which different parts of the design are simulated. HASE supports an hierarchy of component objects that can be instantiated to model the behaviour of an architecture at the PMS, ISP or RTL abstraction level. This hierarchical framework of HASE is based on Flynn's classification of abstraction levels of a processor [22]:

> Let P be a program. A program is defined simply as a request for a service by a structured set of resources. P specifies a sequence of

other (sub)requests, R_1, R_2, ... R_n called tasks. While the tasks appear here as a strictly ordered set, in general tasks will have a more complex control structure associated with them.

Each request, again, consists of sub-requests (the process terminating only at the combinational logic circuit level). Regardless of level, any request R_i is a bifurcated function having two rôles:

$$R_i = \{f_i^l, f_i^v\} \tag{4.1}$$

the logical rôle of the requestor f_i^l and the combined logical and physical rôle of the resource server f_i^v.

Flynn defined a resource server v to have two types: operation or storage. A storage resource is a device capable of retaining a representation of a datum after a result has occurred. An operational resource performs a binary mapping (*e.g.* add, sub, and, nor, xor, shift). Storage resources may also have operational characteristics if accessing mechanisms are included. A program is executed by means of a hierarchy of request transitions between initiating level P and the next level of tasks R_1-R_n. An actual service is provided through a combination of a tree of lower level logical requests and the physical services at the leaves of the request tree.

4.1.2 Implementing the Abstraction Level Hierarchy

An abstraction level will include a set of submodels S_n handling the same *type* of request R_i. R_i at abstraction level l requires a unique requestor f_i^l format that will include less detail than requestor f_j^{l+1} of request R_j. HASE was prototyped to include three abstraction levels, PMS, ISP and RTL. A request type has been specified for each level. A multi-level simulation not only involves hiding the details of a physical component, which is normally the case with logical simulators e.g. VHDL, but in HASE a multi-level simulation also involves abstracting the type of

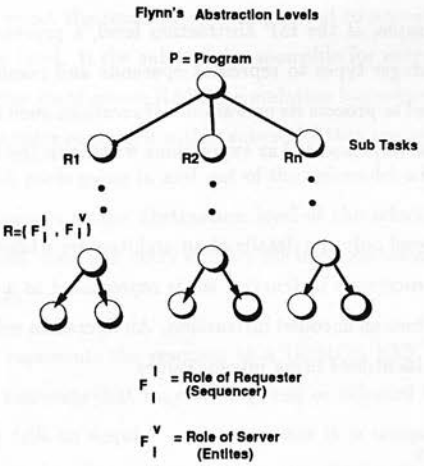


Figure 4–1: Abstraction Level Taxonomy

the requestor f_j^{l+1}. The next three sections outline the main properties of RTL, ISP and PMS requests.

RTL Requests

At the lowest level, RTL register transfer requests involve numbers to the base 2. The responsibilities of DEMOS entities, for example, involve binary operations, e.g. shift left, shift right, invert and so on. An entity may summarise the combinational logic of an architecture component by a *truth table*, in a similar fashion to the functional logic simulators described in section 1.4.2.

ISP Requests

ISP level requests focus on the flow of data between operands and results and the corresponding utilisation of function unit pipelines. Tasks handled by logical requestors at the ISP level, manipulate numbers to the base 10, translated from

base 2. For example, at the ISP abstraction level, a processor's ALU uses floating point and integer types to represent operands and results, depending on the pipeline employed to process its operations. Operations such as *add*, *subtract*, *multiply* or *divide* are implemented as expressions written in the Simula programming environment.

At the ISP level only the details of an architecture's functional properties are considered. A processor's instruction set is represented as a Simula *array*, where each element defines an encoded instruction. An operation refers to register source and destination identifiers using integer values.

PMS Requests

At the PMS level, a processor's internal architecture, memory or context switch behaviour is driven by a set of performance figures that have been collected from RTL and ISP level simulations. The PMS level processor entity is responsible for fetching a dependency graph of tasks, instead of requesting a sequence of encoded instructions. Each type of task corresponds to a delay incurred by the processor and holds onto simulated resources for an estimated interval of simulation time.

The PMS processor's instruction set is classified into a set of different instruction types; for example, *arithmetic*, *logic*, *control*, *data*, *exception handling* routines. Each dependency graph task is composed of a combination of different instruction types. The dependencies specified in the graph are configured from statistics retrieved from running real application programs.

4.1.3 Implementing Resources

The hierarchy of storage and operation *resources* are simulated in HASE by instantiating an hierarchy of DEMOS entities. An entity encapsulates data and operations to describe the behaviour of a resource server f_i^v at abstraction level l.

A task request will select the resource server, required to execute the task at its specified abstraction level. If the submodel responsible for executing this task is not instantiated in the *event* queue HASE's simulation framework will instantiate all the co-existing entities contained within submodel that are required to execute the task request. All *ports* going in and out of the submodel will convert incoming and outgoing formats to the abstraction level of the selected submodel, via *entry* and *exit* entities. Exit and entry entities are instance variables of the HASE simulation framework.

The PMS level represents the resource as a DEMOS *RES*. A RES is a user defined number of elements that may be acquired or released by competing entities. If an entity fails to acquire such a resource it is temporarily blocked in a RESOURCE queue; the Figure 4–2 shows entity $E19$ waiting in the resource queue at simulation time $t4$. Another way of implementing a resource is through the notion of a DEMOS *BIN*. A BIN is a DEMOS class similar to a DEMOS RES, except it is shared by a user defined number of entities and can therefore act as a high level server; for example, it can model a memory queue or a processor.

4.2 Implementing the Design Environment

An important feature of HASE is to provide the computer architect with a reusable set of software components. The HASE environment is designed to hide the implementation details of communication and synchronisation associated with the simulation executive program; to support a simple one to one relationship between the architect's set of reusable software components and the physical architecture components under investigation.

This section explains how the GSS environment and inheritance associated with abstract and concrete component classes are designed to support a framework

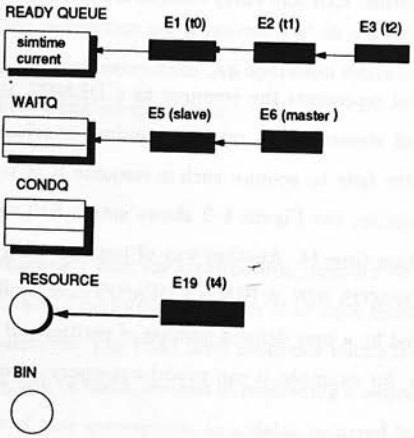


Figure 4–2: Discrete Event Modelling on Simula

for creating and editing architecture simulators using the Object and Architecture Editors.

4.2.1 Object Editor

The purpose of the *Object Editor* is to provide a front-end to the DEMOS programming environment by supporting a menu of icons to represent standard DEMOS simulation primitives. A component entity is therefore constructed from selecting simulation primitives from the icon editor and saving it as an *Activity Diagram*, as outlined in Chapter 3. This section describes the features of three types of DEMOS simulation primitives used by the Object Editor and explains how they are adopted to provide a framework for establishing connections between interacting entity components.

Features of DEMOS

A DEMOS entity is similar to a *class* defined in Simula; in HASE, an entity is used as a template of behaviour describing some architecture component. Also, a DEMOS entity can have multiple instances. However, unlike a Simula class, an entity inherits a library of superclasses to manage discrete event simulation.

DEMOS supports the mechanisms to instantiate the activation of an *entity* at simulation time t. For example, the *Fetch* entity given below is scheduled to instantiate at simulation time 0.0 and is inserted into the head of the a DEMOS event queue.

```
F :- new Fetch("Fetch", F_stage, 1, p(fetchsize), p(fetchinputs),
     p(fetchfields),p(faddsize), p(faddinputs), p(faddfields),
                                                p(fetchdelay));

F.Schedule(0.0);
```

Otherwise it can be scheduled to instantiate at 'the current time + t', where *t* is some delay in simulation time units. In the above example, the parameter *p*() refers to elements in a datafile containing parameters set for the Fetch component. It is possible to instantiate multiple instances of the same entity component. Actions of the *Fetch* entity include acquiring and competing for architecture *resources* e.g. the *Address* and *Data* bus, computing results and exchanging data between entity components.

All component entities execute a DEMOS HOLD procedure for a number of simulated time units, to model the duration of an activity. Seen from inside the calling DEMOS entity, the HOLD procedure represents a period of time when the entity component remains in the same state, locking its acquired resources until it is scheduled to release. DEMOS also provides resource and synchronisation queues for entities that compete for common resources or attempt to synchronise with the same instance of an entity simultaneously. When an entity attempts to acquire a resource and fails, it is inserted into an entity blocked queue. Similarly if an entity is waiting for a condition to be satisfied, an entity instance identifier is inserted into the CONDQ DEMOS class queue.

Entity *cooperations* are supported through a DEMOS COOPT procedure. Executing a COOPT procedure in DEMOS is equivalent to sending a message to an object and retrieving the result returned from the receiving object's method call. In DEMOS, the sending entity synchronises with the receiving entity at some defined stage in the receiver's code. The sender takes program control of the receiver's code, executes it. If the receiver is synchronised to some competing entity component, the sending entity is inserted into a DEMOS COOPT queue.

Component Inheritance

The hierarchy depicted in Figure 4–3 illustrates HASE's use of class inheritance. The *Abstract* entities defined at the top of the class hierarchy (shown with filled

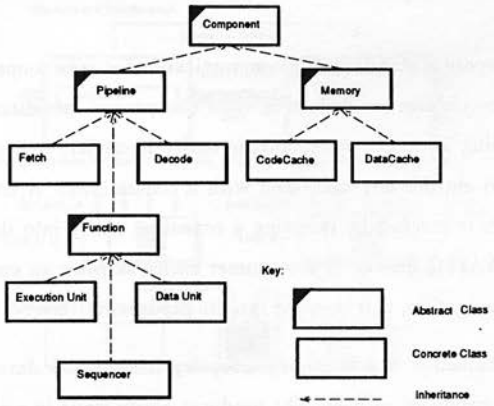


Figure 4–3: Use of Object Inheritance

triangle in figure) factor out common component behaviour. They provide useful functions inherited by instantiated child entities. For example, the *Component* entity is a parent which is inherited by all activated entities in the simulation. The class *Component* defines a standard set of procedures for entity cooperations.

The class *pipeline* defines the unit of data transfer between cooperating entities. The class *pipeline* is responsible for initialising, exchanging and shuffling data through each pipeline stage and includes standard function members to report on pipeline utilisation and throughput. When a pipeline exchange occurs between two entities, its dimensions, including its number of ports and fields, are compared in order to prevent illegal data transfers. An entity can manage an array of class pipelines. Each element of the array stores internal states of the component, for example it may define a single word control register at the RT level or it may represent a superscalar pipeline at the ISP level, depending on how the class has been initialised.

Component Cooperations

For the purpose of standardising communication between cooperating entities, two inherited procedures are defined in class *Component*: **producer** and **consumer**. By specifying an entity type, unique entity identifier, link type, purpose, and source, two entities are associated with a unique link. A cooperation between two entities is invoked by inserting a consumer entity into its producer entity's DEMOS WAITQ queue. If a consumer entity satisfies an entry condition for a specific cooperation, it is inserted into its producer's DEMOS CONDQ queue.

Synchronisation is achieved by adopting master and slave roles for the consumer and producer entities. The producer entity waits in a slave queue, until it is serviced by a consuming entity. The producer entity waits on the simulation event queue until it is rescheduled by the consumer. The consumer selects the appropriate WAITQ identifier which is defined by a tuple $T(LClass, Linkid)$, where *LClass* is the link type and *Linkid* is the link identifier. *Linkid* is referenced by indexing a destination array, with $(EClass, Eid, LClass, purpose)$ fields. *EClass* and *Eid* are the class name and instance variable of the consumer object. *LClass* is the class name of the Link object and *purpose* is the mode the link is currently operating in. For example, Figure 4–4 shows a *Fetch* object and a *Decode* object linked by a *Data Transfer* link class object. In the case shown, *Fetch* is the producer object and *Decode* is the consumer. *Data Transfer* is the name of the link class, and the *purpose* of this exchange is to transfer data between the Fetch object and the Decode object. If the *Fetch* stage was producing instructions, the *purpose* field would have been defined as *instruction* instead of *data*.

All component objects inherit source and destination arrays. The values of the source and destination arrays are initialised by the Code Generator. By assigning the same *Linkid* to both Source and Destination arrays, the producing *Fetch* object and consuming *Decode* object exchange *data* messages on a unique WAITQ channel.

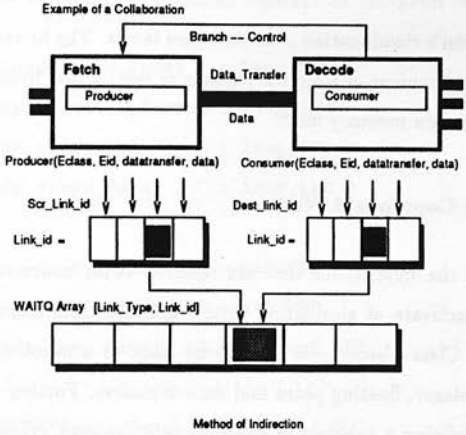


Figure 4–4: Inherited Object Communication

When cyclic entity interactions that are preemptively acquiring and releasing mutually exclusive resources are involved, *deadlock* can occur. Each component entity inherits a procedure to index a simple *semaphore* to test the state of its destination entity and to identify whether it is ready to synchronise and transfer data. If the semaphore is set, the entity will not attempt to cooperate with the blocked entity; instead it continues to process the next available entity collaboration.

Class *decode* entity is also defined as an abstract class. In addition to loading opcode tables, class *decode* is responsible for abstracting hazard management, specifically data, control and structural [32]. Class *decode* manages synchronisations between fetch, function and register file classes to sequence instructions and prevent out-of-order execution. For example, the data hazard function is responsible for holding an instruction until a function unit can bypass its result to a waiting operand. If however register *feedforwarding* is not supported by the architecture class, *decode* will disable bypassing. Class *decode* entity inherits the behaviour of the *Component* class so that all synchronisations between other stage entities execute standard producer/consumer procedures. In the same way, HASE defines

a memory class hierarchy to abstract common behaviour of storage resources as defined by Flynn's classification of abstraction levels. The hierarchy expands from modelling the behaviour of a simple register or instruction buffers, to modelling a cache or large main memory unit.

Specifying a Component Node

The leaves of the inheritance tree are referred to as *concrete* classes, they are scheduled to activate at simulation time t and are time stamped in a DEMOS event queue. Class *Component* is a parent class to a number of function units, for example integer, floating point and data transfers. Further abstraction can be achieved by defining a subclass of different function unit types. Class *decode* has a tree of subentities, each responsible for executing a different instruction set.

GDL is used to summarise the properties of a concrete entity. A concrete object has an associated GDL node that specifies its facets, including an attribute list, referenced by the graphical front-end and used to parameterise the object. The GDL file defines the number of input and output links a concrete entity can make with co-existing object components. For each link, the GDL file specifies the types of entities that it is allowed to connect. Each Link has an associated GDL *link* node that is parameterised to define its purpose, whether it is intended to be used for data transfer, control or issuing. An example of a class node defined in GDL code for ISP *fetch* entity is given below:

```
NODECLASS    Fetch
INFO         [ Iunit\
\
This node class represents the fetch stage of an
 88100 microprocesor\
]
```

```
ATTRIB
            abstraction_level    : long_int
            super_scalar_factor : long_int
            ISP_pipe_size        : long_int
            ISP_stage_delay      : long_int

PORTS       in  <- *  plain  AT north END
            out ->    plain  AT south END
            side <>*  tangle AT east west END

DISPLAY     ICON("icons/fetch.icon") AT 1 0
            SUBGRAPH ? ELLIPSE(74 74 thin double) : AT 0 0

LINKED_SGTYPE rtlgraph

END
```

The attribute section denoted by ATTRIB, defines the parameters for the object component fetch. The section PORT identifies which sides of the fetch icon can be connected to co-existing object components and each connection's associated direction. For example, a component object can be connected to the fetch icon's north side, and the connection is directed toward the fetch icon. The DISPLAY identifies the icon that will represent the fetch object component. LINKED SGTYPE denotes that the fetch object component has an associated subgraph to describe its behaviour at the register transfer level of abstraction.

HASE also uses GDL to describe the link classes that connect object components. An example of a link class specified in GDL is given below:

```
# Control link- for synchronisation
```

```
LINKCLASS  control
STYLE      dash double_arrow
PORTS      out  <> * control END
           in   <> * control END
ATTRIB     purpose: <sync ack trans enable>
REFUSE     processor * -> processor *
END
```

LINKCLASS is the name of the link, i.e. control. The STYLE attribute defines the appearance of the link. PORTS defines the connections of the link class, for example, in-going and out-going control links. ATTRIB defines the purpose of the link, for example, whether or not the link transfers data, instruction words or signals that synchronise architecture components. REFUSE defines illegal links between object components. In this simple example, the control link cannot connect a processor component object directly to another processor component object.

Linking the Submodel Hierarchy

An entity's submodel is specified in a GDL file. Each submodel defines a list of nodes that will appear in the subgraph icon menu and a list of Link type options. Each subgraph defines an abstraction level and a set of concrete entities that can be instantiated. As shown in Figure 4–5 the GDL Hierarchy is similar to Flynn's resource tree.

A submodel S_1 containing a set of interacting entity members E, which describes a component's behaviour at abstraction level l_i, can send a message to a submodel S_2 at abstraction level l_{i+1}. An entity E_n contained within submodel S_2 sends a message to another entity E_m belonging to submodel S_1, by sending its message via an *Exit* object. E_m will receive a message from E_n via an *Entry* object. Entry and exit objects are subclasses of the Component class and are responsible for translating data exchanged between two different abstraction levels;

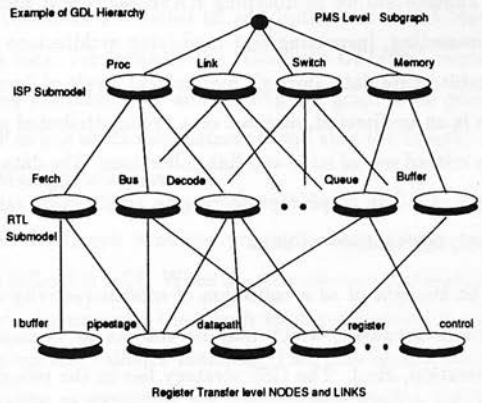


Figure 4–5: GDL Submodels Implementing Resources

for example, between decimal notation described at the ISP layer and low level binary notation described at the RTL.

4.2.2 Architecture Editor

The Architecture Editor is a front-end prototyped in GSS to provide a means to insert, link, parameterise and instantiate software components from HASE's entity hierarchy. The architecture is also described by a GDL graph which is parsed to generate the datafiles and simulation source code. This section explains the functionality of GSS, front-end implementation, graph management and the mechanism developed to generate simulation code.

Functionality of GSS

GSS was developed to support the IMSE (Integrated Modelling Support Environment) project which was a collaborative research project supported by the CEC as ESPRIT project no. 2143. It was carried out by a number of organisations including the University of Edinburgh and STC Technology Ltd.

GSS was appropriate for prototyping HASE because it provided a consistent means of representing, inspecting and modifying architecture parameters. GSS structured architecture data through hierarchical levels of "graphs" and "data". A GSS graph is an undirected, directed or a cyclic attributed graph consisting of hierarchically related sets of node and link collections. The data that describes the nodes and links of a GSS graph representing an architecture can involve complex, possibly nested, aggregations of data.

GSS can be thought of as a collection of models (activity diagrams, internal architectures and networks) which may be subject to various operations (editing, code generation, etc.). The GSS strategy lies in the recognition that all the proposed modelling paradigms use graphs (i.e. a collection of nodes and links) to represent the structure of its model. The graphs are *attributed*, that is, the objects in them have associated data, and so there is a requirement for a uniform way of defining and handling complex data types. To implement these requirements, GSS provides:

- A library of functions for handling graphs and data, known as the GSS library. GSS function library components fall into two categories: those that use the graphical facilities of the workstation (i.e. Sunwindows in the prototype) to provide an interactive tool, and those that do not require graphics.
- A Graphical Description Language, defining the types of graphs and data for specifying an architecture component.
- A file format for storing instances of graph data types.

Managing Architecture Graphs

When invoked, the Graph Manager reads a list of subgraphs from the command line and calls *gss_make_graph_window* to initialise the Suntool's window envir-

onment. GSS functions are called to add options to the Architecture Editor's front-end menu bars. For example the "Generate DEMOS" menu option was included by calling *gss_add_menu* and passing the graph code generation function *graph_analyser* as one of its parameters. It will save the Graphs and Data in the OMS (Object Management System).

The OMS is the Graph Manager program responsible for mapping data to nodes and links defined in GDL. When the user selects a *makelink*, *load*, *set_attributes* or an *icon_rotate*, for example, the Graph Manager program is responsible for executing the appropriate callback function. This may involve loading and storing files, parsing graphs, or executing low level Suntool window functions.

Traversing an Architecture Graph

The Graph Manager saves an undirected graph $G(V, E)$, where V is the number of vertices, and E is defined as the number of edges. A simple scoreboard is maintained in the form of a linked list to update the number of instances of each type of Node and Link class visited. The *graph_analyser* visits each node or link class exactly once. The *graph_analyser* calls *gss_forall_instances* recursively for each class defined in the top level submodel, for example, the PMS submodel, so that eventually all nodes are visited in each submodel. The *gss_get_values* function is called to return the values of the attributes set for the simulation. For example, one of the attributes identifies whether or not a visited node has a submodel. If a submodel exists the function *gss_get_subgraph* is called, to extract the attributes and node link identifiers for each associated entities. The entities' attributes are written to a datafile file and read later during the simulation and animation phases.

For the purpose of setting synchronisation links, *gss_forall_linked* is called to visit all outward bound links. A function is called on the destination node to

return the essential link information that is stored in the Link scoreboard and later written to the simulation code file.

Simulation Code Generation

The graph traversal phase generates simulation source files to declare entity class names, instance numbers, link types, link purposes, and parameter identifiers. The graph traversal phase writes to declaration source files that, for example, initialise the source and destination link arrays referred to by producer and consumer procedures during an entity cooperation. It is responsible for producing the source code to schedule entities chosen from the abstraction level hierarchy. Simulation source code is a concatenation of 5 main source files:

- Entity class names, instance numbers, Link Types, Link purposes, parameter names.
- Global variable declarations, including source and destination arrays, that define entity-entity collaborations.
- Entity class declarations that are included in the submodels selected for a simulation experiment.
- Main program assignments, including WAITQ initialisation, Link identifier assignments for source and destination entities; later referenced by producer and consumer procedures inherited from the Component superclass.
- Motif code file to set-up diagrams of the architecture, corresponding to the abstraction levels that have been included for each component object.

Figure 4–6, shows how the parser process maintains a log of the nodes and links at each abstraction level. In this example the PMS level has three classes associated with it. The PMS data structure maintains a list of objects instantiated

for each class. For example, Class 3 is a cache class modelled at the PMS level, and two instances of this class are instantiated into the simulation. Each entry in this list contains connection details, for example, its in-going and out-going links to other objects in the simulation. Figure 4–6 also shows a list of the Link objects that must be instantiated into the simulation. For each link object entry there is a pair of instance identifiers to for referencing the source and destination component objects. The parser program traverses this data structure for each abstraction level, to define the DEMOS synchronisation links, for example, WAITQs and CONDQs, between the instantiated component objects.

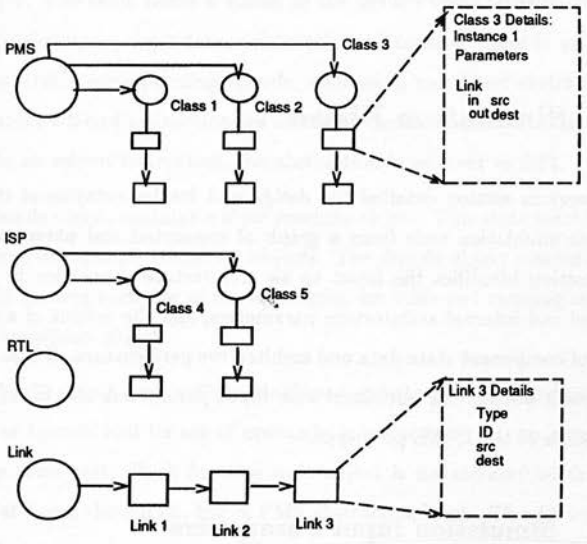


Figure 4–6: Recording Node and Link Connections

The algorithm is as follows: Node N is visited, the object component's attribute values are retrieved, and the node's class details are updated. There are two arrays that contain the unique WAITQ linkid for each entity–entity cooperation, $src\_link\_id[linkclass, linkid]$ and $dest\_link\_id[linkclass, linkid]$. When an

outward bound link is found at node N, its type and purpose have to be registered. A string is written to the main file of the simulation source code that assigns the *src_link_id* array to a unique Link class instance number. A string is similarly created for the *dest_link_id* assignment with the same *linkid*. The indices of the *src_link_id* and *dest_link_id* arrays reference the unique *CONDQ* and *WAITQ* queues for communication between pairs of instantiated entities during the lifecycle of a simulation's execution.

The declarations for entities and classes are read from file path names and concatenated into a Class/ Entity File. The Class/Entity file forms the skeleton of the simulation's source file.

4.3 Simulation Phase

The previous section detailed the design and implementation of the process to generate simulation code from a graph of connected and parameterised nodes. This section identifies the input to an architecture simulation in terms of the external and internal architecture parameters, and the output of a simulation in terms of component state data and architecture performance. It also explains how component objects are initialised with input parameters and identifies the main constraints of the HASE prototype.

4.3.1 Simulation Input Parameters

This section describes the three main types of data required to initialise a HASE simulation:

- *External Architecture* parameters
- *Internal Architecture* parameters

- *Simulation Setup* parameters

External Architecture

The term *External Architecture* refers to the components of a computer that are visible to its user, i.e. its instruction set. In HASE, the decode object is initialised by reading a decode table which represents the instruction set. The decode table defines the operation codes and the operand methods that are supported by the simulated architecture. An example of an instruction set opcode table is given in Table 4–1. The table shows a subset of the 88100's external architecture. For each instruction type, e.g. data, arithmetic and control, there is an assembly instruction with a corresponding opcode, addressing mode and abstraction level. Notice that for a **bcnd** instruction, for example, the abstraction level is set to ISP, whereas for an **addui** instruction, the abstraction level is set to RTL.

The decode object contains a state machine object. This state machine object is an ordered set of state transition objects. The decode object creates this state machine by reading each row of the instruction set table and creating an instance of a state transition object.

The decode object uses a *State Machine* object to identify the function unit to which an opcode and its set of operands is to be sent; i.e. an *integer*, *data*, or *floating point* unit. Each function unit object is instantiated to describe its operation at one of three RTL, ISP or PMS abstraction levels. The decode object's state machine is sent a message which contains the most recently fetched opcode as its parameter. On receiving this message the state machine searches for a match amongst its ordered set of state transtions. The matching state transition object will identify the function unit and the abstraction level responsible for executing the opcode within it. The decode object sends the message to an instance of the function unit responsible for executing the operation at the specified level of abstraction.

Table 4–1: External Parameters: Setting Opcode Abstraction Level

Code Class	Assembler	Opcode	Addressing Mode	Abstraction Level
Arithmetic	addu	00011	triadic	ISP
	addui	00012	immed	RTL
	subu	00013	triadic	ISP
	mul	00009	immed	ISP
Control	bra	00007	immed	ISP
	bcnd	00009	immed	ISP
Data	ldi	00001	immed	ISP
	sti	00008	immed	ISP

The behaviour of the decode object is logically dependent on a processor's internal architecture. For example, the decode must be updated to accommodate any changes to the behaviour of a particular function unit; otherwise its decoding and dispatching messages will not be recognised. In HASE, a Decode class is specialised using the object oriented principle of inheritance. The class name for a subclass of class Decode will reflect the version number of the processor that is being designed. For example, there are two separate decode subclasses for the MC88100 and MC88110 microprocessors.

In the HASE prototype a decode subclass can be represented by an activity diagram. An icon menu provides a selection of different types of instructions that can be simulated by an architecture model. For each instruction type, there is a separate menu to select different assembly instructions. Each assembly instruction can be selected to simulate at a chosen abstraction level. The array of selected and parameterised icons is converted to a table, like the example table shown in Table 4–1.

Internal Architecture Parameters

The internal architecture of a processor refers to the hardware components that are invisible to the user; for example, the associativity of its cache, the pipelining

of its function units and the arbitration mechanisms which control the return of results to an architecture's register file.

In HASE the parameters for each internal architecture component are defined during the architecture edit phase. After an architecture edit session, the parameters for each object component are stored in a datafile. At the start of the simulation phase, the internal architecture object components are instantiated with the parameters defined in this datafile. It is the responsibility of each component object to select its own parameters from the datafile.

Simulation Setup Parameters

Simulation setup parameters refer to global data the user enters to control the life cycle of the simulation. For example, the user may wish to specify the total duration of the simulation in terms of the number of simulated clock cycles. The user can parameterise the performance monitor objects, for example, to define a time interval for measuring utilisation and throughput, or setting up a monitor object to count the number of instruction stalls that match a defined pattern of behaviour.

Furthermore, a simulation run may involve executing a test program more than once. The user can specify the number of simulation runs required. The user can specify which internal architecture paramaters he/she wishes to change on each new run of the simulation. For example, a user can specify 10 simulation runs executing the same test program, but can specify that after each successful execution, a new stage is added to the floating point unit.

4.3.2 Simulation Execution

This section describes the principal types of output generated by a HASE simulation. These are:

- *Trace Events*: an event trace reporting all simulation events up to and including the current simulation time

- *Object Component State Data*: the state of a component object's attributes for the current simulated clock cycle

- *Performance Statistics*: the performance statistics, specifically: utilisation and throughput of an object component.

All component objects involved in a simulation provide services to report their own state during a simulation and, when requested, return statistics to objects that are dedicated to monitoring the performance of the processor. An object component inherits these services from the abstract superclass *Component*. The next three subheadings detail each type of simulation output.

The Event Trace

A Standard DEMOS output trace can be toggled on or off at defined break points in the simulation source.

Trace information includes:

- Initialisation of entities; i.e. the simulated start time of an instantiated entity.

- Entity cooperations; including how often producer/consumer entity pairs are inserted into DEMOS CONDQ and WAITQ queues and the time (in simulation units) that a slave entity waits for its master to consume its data and resume execution.

- Blocking entities; how long an entity waits for the release of a resource.

- The termination time of an entity.

This data is parsed on a "line by line" basis by the Trace Animator. The details of the Trace Animator are described in Section 4.4.2.

Object Component State Data

The class Component provides a service to output state information during the course of a simulation to a state display file. This method is inherited by all internal architecture component objects instantiated in a simulation. The object component time stamps its state before being written to an output file. The HASE prototype reads the state display files for the purpose of animating an object component's behaviour. Further details of the Architecture Animator are described in Section 4.4.3.

Performance Statistics

The services that return statistics about the performance of all the object components in the simulation are provided by the superclass Component. These services are inherited by all component subclasses and can be overwritten if the user wishes to redefine the service, e.g. to examine more specialised activity about the component's behaviour.

The monitor objects are responsible for sending messages to the object components to ask for statistics during a simulation. When a monitor object receives a reply it timestamps the statistics returned by the object component and writes it to an output file.

There are two main types of monitor object defined in the prototype version of HASE:

- *Component Probes*: these measure architecture performance
- *Simulation Probes*: these monitor the efficacy of the simulation.

A *Component Probe* monitors the performance of selected component objects instantiated in an architecture simulation. Typically the types of services they provide are to:

- count of the number of stalls that occur in a pipeline
- calculate the total utilisation of a function unit
- count the number of occupied stages in a pipeline
- calculate the total number of control, data and structural hazards during the simulated execution of a test program.

A *Simulation Probe* monitors the efficiency of the simulation program. Specifically, a simulation probe provide services to:

- calculate the average number of clock cycles simulated per second
- report the CPU utilisation during simulation, identifying the percentage of idle CPU time
- report on the memory usage during a simulation run.

4.4 Evaluation Phase

This section outlines the communication pattern between HASE's front-end components: the Trace Animator, the Architecture Animator and the Graph Displayer. It explains how each component is designed to manage the data collected during the architecture editing and simulation phase, and when requested, display it to the HASE user.

4.4.1 Overview

The HASE front-end is designed and implemented to visualise the behaviour of a component or collection of components during and after an architecture simulation experiment. As described in Chapter 3 the HASE front-end is composed of five main tools: an Object Editor, an Architecture Editor, a Trace Animator, an Architecture Animator and a Graph Displayer. Although each component is prototyped here as a separate process, ultimately HASE will encapsulate all five processes as one seamless application. The purpose of this implementation is to demonstrate the concepts for evaluating the performance and visualising the behaviour of selected architecture components during a simulation experiment.

Figure 4–7 illustrates the framework that controls the front-end processes. The HASE animator process is the central process to the front-end. It may be executed during or after an architecture simulation experiment. The simulator's output files can be piped directly to the front-end during a simulation run or read separately after the simulation has successfully completed.

The HASE Animator process is responsible for reading and displaying the contents of the test program and spawning the Trace and Architecture Animator processes. The Trace Animator is responsible for fetching and displaying the Event Trace file generated during the simulation phase. The Architecture Animator is responsible for retrieving and displaying the architecture configuration file that specify the components the user has selected for animation. The Architecture Animator is also responsible for retrieving and displaying state, parameter and component code files. When the user wishes to animate a simulated clock cycle the Architecture animator is responsible for broadcasting the updated clock cycle and updated simulation time to the Trace Animator. The Graph Displayer is responsible for retrieving and displaying the graph data files when requested by the HASE user.

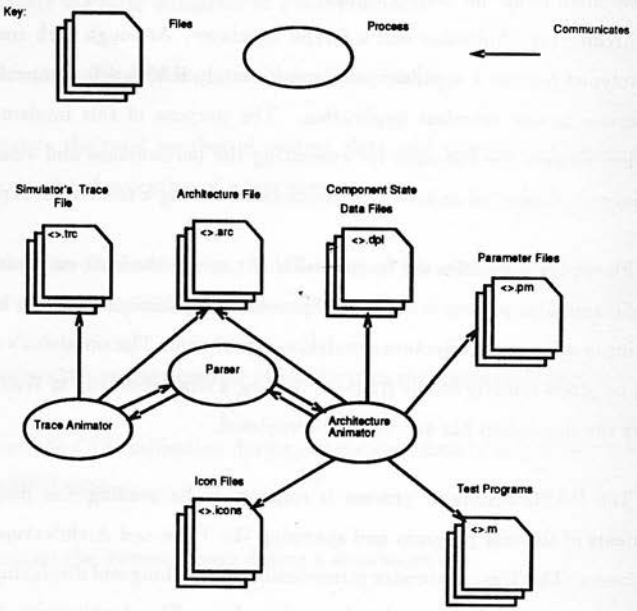


Figure 4–7: Animator Structure

The next three sections detail the design of the Trace Animator, the Architecture Animator and the Graph Displayer.

4.4.2 Trace Animator

The purpose of the Trace Animator is to identify the interactions of DEMOS entities competing for resources, during a simulation. The Trace Animator is designed to give the HASE user an indication of how efficiently the object model is behaving during a simulation run. For example, describing a 88110 microprocessor's floating point unit pipeline at the RTL level will require a considerable number of entities. The management of the interaction between these pipeline and control entities will slow the performance of the simulation. If the user does not require the detail given at RTL, the user can model the entire floating point pipeline as one entity. The model will therefore not waste valuable CPU time to produce redundant performance information.

The *Trace Animator* is spawned by the HASE Animator process. It receives, as an argument, an Event Trace File, produced by a successfully completed simulation. The Trace Animator provides functions to *load* an event trace, *play*, *rewind* *step forward* and *step backwards* through an event trace. The Trace Animator calls a parser function to read a single line of the output DEMOS trace file. It then calls functions to translate the word description of the trace file into a moving picture of boxes and lines activated by SRGP(Simple Raster Graphics Program) routines.

4.4.3 Architecture Animation

The Architecture Animator is designed to capture the behaviour of an architecture's components during a simulation experiment. The Architecture Animator is spawned by the HASE animator process. The Architeure Animator is supported

by the Motif environment. Motif is a superset of Athena widgets, also written in C. It is possible to call *Xt* functions inside a Motif application program.

Motif Support Environment

The main reason for implementing the Architecture animator in Motif is because of its high level functions for supporting icon widgets. For example, it provides a **bulletinboardwidget** which allows an icon to be positioned anywhere on a widget, just by simply specifying a set of $x - y$ coordinates. Motif also provides a simple interface for managing scrollbars and other high level interface facilities.

Functions of the Architecture Animator

Initialising the Architecture Animator process involves calling a function to read the architecture configuration files saved during the Architecture Edit phase. The configuration file contains a list of component objects and their subcomponents to be included in the animation. The Architecture Animator iconises the component objects and displays them to the HASE user.

The process of iconising a component object involves referring a set of callback function, that are presented to the user in the form of a pull down menu. The prototype version demonstrated the use of five main callback functions specified for each class of object component: animate, display state, display parameters, display code, zoom, and quit. Each instance of a component object has its own version of this set of callback fuctions. For example, zoom will display a different set of lower level objects in a new *popup* shell, depending on the component icon that has been selected. The section below describes the main callback functions in further detail.

Animate Callback Function

The Animate Callback function is executed when the Animate item on an icon's pulldown menu is selected. This event triggers an increment of the current simulated clock cycle number and simulation time. The new clock state is updated for both the abstraction levels above and below the abstraction level of the selected component. The Animate Callback function is designed to update the current state of all listed object components the HASE user is viewing. Furthermore, a message, containing the address of the currently decoded instruction, the clock cycle executed and simulation time, is sent to the HASE Animator process. The HASE Animator broadcastes this message to its Event Trace Animator process if it is running. Based on the value of the current simulation time, the Event Trace Animator updates itself.

In addition to updating component object displays, the Animate Callback function requests the Graph Displayer process to refresh itself.

Zoom Callback Function

The Zoom Callback function is executed when a mouse down event is received on the Zoom item of an object component pulldown menu. The function refers to a configuration file produced during the Edit Architecture phase, to identify the list of object components that must be iconised. The configuration file is indexed via the object component's class name and instance identifier.

Display Callback Function

The Display Callback function responds to a mouse down event by retrieving the state file for the selected iconised component, produced during the Simulation Phase. If the simulation and animation processes are running concurrently, then

the data is piped directly to the component's display window, every simulated clock cycle.

Parameter and Code Callback Functions

The Parameter Callback function retrieves the parameter file for the iconised object component selected. This file is produced by the Architecture Edit phase. Similarly, a callback function retrieves the file containing the class description for the iconised object component.

4.4.4 Graph Displayer

The Graph Displayer process is provided by the X Window application **gnuplot**. This application is spawned by the Architecture Animator process. Once the application is running the Animate Callback function will ensure that the gnuplot display is refreshed to cover the next set of simulated clock cycles. The HASE user can invoke further callback functions from the user interface to zoom into a gnuplot to investigate, for example, an unusual pattern ocurring on the pipeline utilisation graph.

Chapter 5

Results and Discussion

Overview

The purpose of this chapter is to demonstrate the use of the HASE prototype to experiment with internal and external architectures. The technical background for the architecture experiments presented here is explained in Chapters 1 and 2. The description of each architecture experiment includes:

- A brief description, to explain the purpose of the experiment.
- A flow of events, to describe how to carry out the experiment.

Each architecture experiment is illustrated by one or more **scenarios**. A scenario consists of:

- A brief explanation of the specific aspects of an architecture that require investigation.
- a step by step description of the scenario, supported by references to screen snapshots taken from the HASE prototype.
- a comment on the results of the experiment and a critical assessment on the usability of the HASE prototype.

This chapter demonstrates the use of HASE through the four different types of architecture experiment listed below:

- Section 5.1, *Experiment 1: Investigating Internal Architecture* - enhancing the performance of a microprocessors's internal architecture
- Section 5.2, *Experiment 2: Investigating External Architecture* - enhancing the performance of a microprocessor's external architecture
- Section 5.3, *Experiment 3: Investigating Hardware/Software Interaction* - optimising the usage of a microprocessor's internal and external architecture
- Section 5.4, *Experiment 4: Investigating Network Traffic* - determining the relationship between a processor's architecture and its external network traffic.

5.1 Investigating Internal Architecture

This section describes how to use HASE to investigate the impact of adding a new component to an internal architecture.

5.1.1 Brief Description

The investigation begins when an architect wishes to integrate a new component into an existing internal architecture. The architect uses HASE to identify whether the new component will improve the performance of the microprocessor's internal architecture.

If the architect observes a significant performance improvement, the expected cost of including this new component to the architecture, will be determined.

If there is no significant performance improvement, the architect will save the latest version of the new component into the HASE component library for future reference.

5.1.2 Flow of Events

To investigate the effect of adding a new component to an existing internal architecture, the architect performs the following steps:

1. The *Object Editor* is used to create an object to model the behaviour of the new component. This involves identifying the attributes of the new component, the services the new component supports and the protocols for interacting with components belonging to the existing architecture.

2. The *Architecture Editor* is used to insert the new object into the existing architecture. This involves parameterising the new component and linking it to its neighbouring components.

3. The *Simulation and Architecture Probes* are used to measure the performance of the enhanced architecture.

4. The *Architecture Editor* is used to link and compile the generated simulation code.

5. A set of test programs is prepared for the enhanced microprocessor. It is assumed that for this experiment, the modification to the internal architecture does not effect the external architecture.

6. The simulation is run to collect trace and architecture events and performance statistics.

7. The *Graph Displayer* is used to provide an overall view of the new component's performance, in terms of utilisation and throughput.

8. The *Architecture Animator* is used to observe the new component's state during the simulated execution of a test program. Simulation runs are repeated for the remaining set of test programs.

9. The architect modifies the parameters of the new component and reruns the set of test programs.

10. The set of parameters for the new component are optimised until the performance improvement of the new component is maximised, or until the architect decides the new component will not add value to the existing internal architecture.

5.1.3 Add a History Buffer to an Existing Architecture

The purpose of this section is to demonstrate a specific example of the type of experiment described above.

Scenario Description

The scenario begins when the architect observes that there is a data dependency between the fetch stage and the decode stage. The fetch stage is waiting for the decode stage to accept the next word of data. The architect animates the execution of a convolution test program and notices that instruction pre-fetching cannot continue until the target address of the previously fetched instruction is calculated by the execution unit.

The architect decides to insert an *Instruction History Buffer* between the pre-fetch stage and the execution unit, which can be filled while the execution unit calculates a branch instruction's target address.

The architect searches for an Instruction History Buffer in the HASE component library, but does not find one. The architect defines the attributes and

services for a history buffer and adds it to the component library. The architect incorporates the history buffer into the existing internal architecture and attaches a probe to the fetch and execute pipelines to measure the architecture's simulated instruction throughput.

Step Description

This section describes the steps taken to add a history buffer using the HASE prototype.

1. View the *Graph Displayer* and observe a low throughput associated with the fetch and execution unit. Figure 5–1 shows the average number of instructions processed per clock cycle for the Decode, Integer, Floating Point and Data unit pipelines. At the beginning of the simulation the Decode pipeline peaks at approximately 2 instructions per cycle, and then steadly decreases to an average of 0.75 instructions per cycle. This is because the Decode pipeline is blocked waiting for the execution unit pipline to resolve data dependencies between both arithmetic and control transfer instructions.

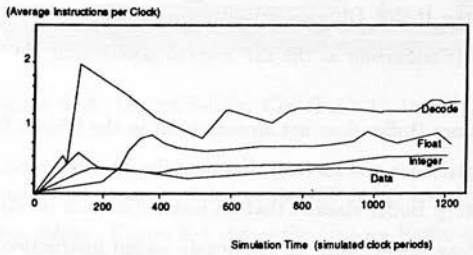


Figure 5–1: Graph Displayer: Poor Throughput of Execution Unit

2. View the *Architecture Animator* over the period immediately after the Decode pipeline peaks at 2 instructions per cycle. Figure 5–2 shows a typical display of the Architecture Animator's Assembly Code window (bottom right

window) which helps to identify the instructions that are blocked due to data and control dependencies.

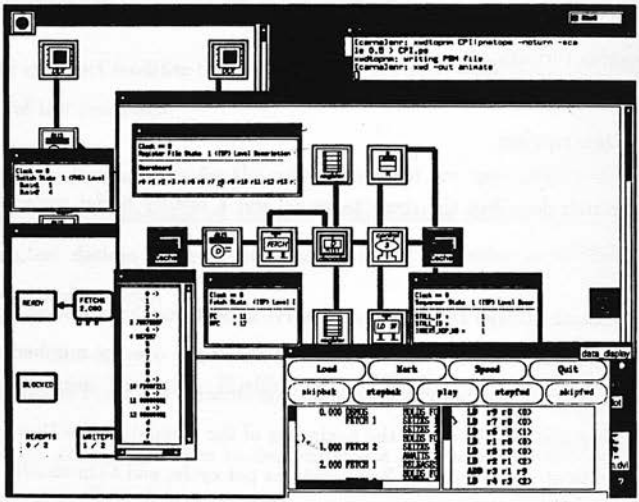


Figure 5–2: Architecture Animator: Identifying Control Transfer Latency

3. Access the HASE Object Repository for a History Buffer component that models its behaviour at the ISP level of abstraction.

4. If a History Buffer does not already exist in the Object Repository, define a set of attributes and services that describe its component behaviour. Define the History Buffer class so that an instruction and its state data remains in the history buffer, until all previously issued instructions have successfully completed. If an issued instruction fails to execute; for example, due to a floating point exception, the History Buffer class will flush all of its out-of-order instructions and associated state data. Similarly, the History Buffer class will flush all pre-fetched instructions after a control transfer instruction, if the branch is not taken.

5. An Icon is created to represent the History Buffer component. Link Refusals are defined for the history buffer icon in GDL, as explained in Chapter 4. Figure 5-3 shows an example of the *Object Editor's* typical display layout for inserting the History Buffer component into the Decode component's activity diagram. In addition, *Monitor Probes* are attached to measure the utilisation and throughput of the decode, fetch and execution unit pipelines.

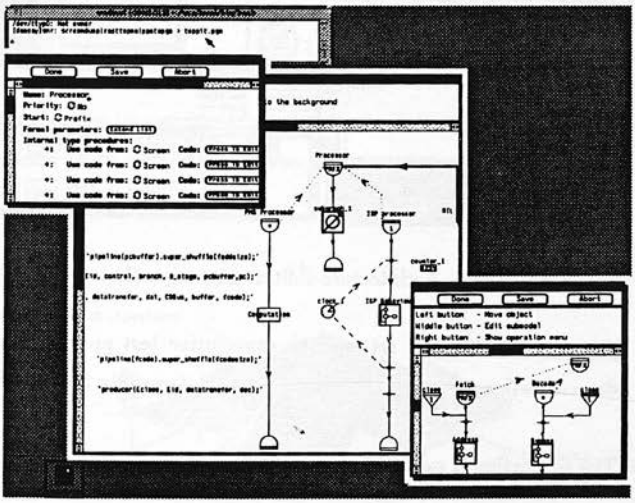


Figure 5-3: Object Editor: Creating a History Buffer

6. The existing architecture is edited at the ISP level of abstraction using the *Architecture Editor*. Figure 5-4 shows the History Buffer component being linked between the Decode Stage's instruction pipeline and the instruction's data bus. The history buffer length is initially set to 2 and each element in the buffer is initialised to contain two instructions and associated state data.

7. A set of test programs, including a dot product and a convolution program are selected to test the enhanced architecture. (An example of the simulated

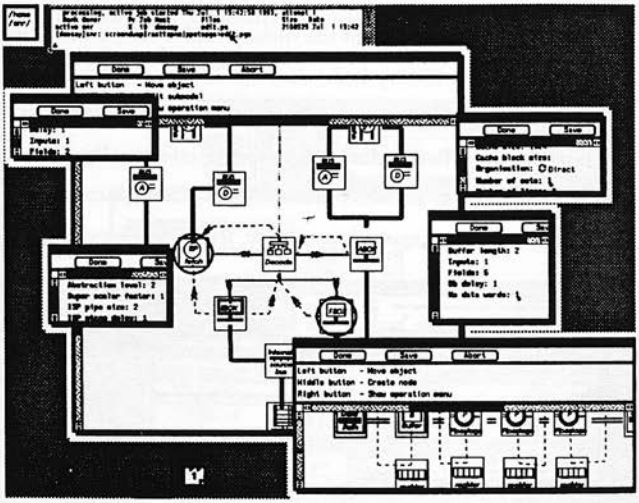


Figure 5–4: Architecture Editor: Linking a History Buffer

processor's instruction set and the convolution test program used for this demonstration are given in Appendix A and Appendix B respectively.)

8. The simulation is executed.

9. The throughput of the fetch and execution unit pipelines is compared against the old performance measurements of the architecture. Figure 5–5 shows that during the execution of the convolution test program, the average throughput of the fetch and execution unit pipelines improved by approximately 10%.

10. The performance graph for throughput shows that when the branch is wrongly predicted, for example the first time a control transfer instruction is encountered, there is a delay while the flushed history buffer is refilled. (Figure 5–6) shows the Decode unit pipeline filling its history buffer between

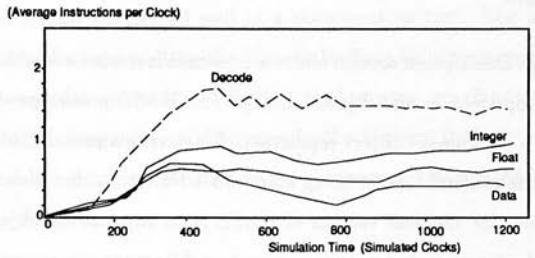


Figure 5–5: Graph Displayer: Improved Function Unit Throughput

clock cycles 10 and 20. After clock cycle 20 the execution pipeline contains a fresh set of issued instructions.

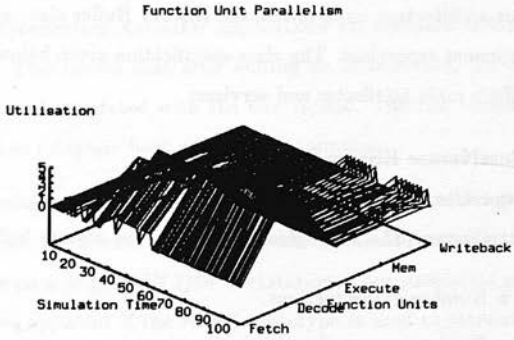


Figure 5–6: Graph Displayer: Flushing History Buffer

11. From simulation and animation results the history buffer appears to improve the average throughput of the execution unit pipeline.

Comments

In the Step Description section above a reference is made to searching for a History Buffer component in an Object Library. The HASE prototype does not provide access to a component object repository. Retrieving a reusable object component requires the manual task of using standard UNIX **grep** functions. An important requirement for the real version of HASE is to support an object repository to contain the latest tested versions of each architecture component and its associated documentation. The architect will browse through the object repository for components that resemble the desired object component behaviour. Once a new object is designed, implemented, tested and documented, the architect inserts the new component into the object repository.

In this architecture experiment, the History Buffer class was subclassed from the Component superclass. The class specification given below describes the History Buffer's main attributes and services:

ClassName: History Buffer

SuperClassName: Component

Attributes: The attributes provided by the history buffer include:

- Number of instructions.
- Delay to return the next instruction.

Services: The services provided by the history buffer include:

- Clear contents of History Buffer.
- Insert a pre-fetched instruction.
- Dispatch an instruction to function unit.
- Save the result of a completed (out of order) instruction.
- Flush all instruction and their associated state information that are not committed for completion.

The *Object Editor* performed well as a documenting tool. The activity diagrams were useful for representing the History Buffer's behaviour graphically and provided a means for conveniently loading and storing an object component. However, the GDL language, used to specify the History Buffer's icon symbol, parameters and linking characteristics, is not conveniently integrated into the Object Editor application.

The Code generator provided by the *Object Editor* was poor at linking instance variables into the component object's source code. The turnaround time for creating a new tested History Buffer object took approximately 8 hours, 10% of this time was spent recompiling the code generated by the Object Editor.

On the HASE prototype, the *Object* and *Architecture Editors* are linked to the *Trace* and *Architecture Animator* applications via standard UNIX pipes and file descriptors. This means that after editing an architecture, the *Architecture Animator* needs to be updated with the new layout. The real version of HASE will be designed to integrate both editing and animation.

When animation was turned off the simulation ran at an average of 34 simulated clock cycles per second. This figure was calculated running the DEMOS HASE prototype on a single SUN 3/80 workstation. Poor simulation performance will become more apparent if the HASE prototype is used to simulate large application specific test programs, instead of small handcrafted test programs; like the convolution program used for this simulation experiment.

5.2 Investigating External Architecture

This section describes using HASE to investigate how the density of an instruction set effects the performance of a processor's architecture.

5.2.1 Brief Description

This experiment begins when an architect wishes to add a new operation or addressing mode to a microprocessor's instruction set.

5.2.2 Flow of Events

To perform this type of architecture experiment the following procedure is carried out:

1. The *Object Editor* is used to edit the decode object and add a new entry to its opcode table.

2. The *Architector Editor* is used to parameterise the new opcode to a PMS, ISP or RTL abstraction level.

3. The *Architecture Editor* is used to link the decode object to the execution and memory units.

4. A test program is prepared to test the enhanced external architecture. It is assumed in this experiment that the architect runs the simulation on a small set of handcrafted set of test programs.

5. The *Architecture Editor* is used to add monitor probes to appropriate memory, execution and bus objects.

6. A framework of the simulation code is generated using the *Architecture Editor*. The source code is compiled, linked and executed against a set of test programs. During execution, the event traces, state data and performance metrics for the architecture simulation are collected.

7. The performance of the enhanced architecture, in terms of function unit throughput and utilisation is viewed using the *Graph Displayer*.

8. The states of the execution pipeline are diplayed using the *Architecture Animator* to identify potential bus contention, execution and memory unit stalls.

9. The architect decides whether or not the instruction set enhancement adds value to the performance of the architecture.

5.2.3 Adding a New Addressing Mode to an Operation

This scenario uses HASE to investigate how adding a new addressing mode to an operation, impacts a microprocessor's performance.

Scenario Description

The architect wishes to increase the code density of an external architecture by adding a register-memory addressing mode to an ADD operation. For example, instead of writing:

```
LOAD  R1, #xxxx
ADD   R2, R2, R1  (or stores)
```

The code becomes:

```
ADD   R2, #xxxx
```

Where *xxxx* is some defined memory address. This new addressing mode gains the advantage of reducing the number of instructions required to write a program and therefore reducing the number of possible instruction cache misses. Due to the extra access time required to retrieve data from memory, the architect assumes that this enhanced addressing mode will cause the clock cycle to increase by approximately 10%.

The architect wishes to test the modified architecture on a set of test programs to investigate how it effects processor / memory traffic and the average throughput of the execution pipeline.

The architect observes that if there is a proportion of load instructions in the application of less than 11%, the extra latency caused by the ADD address mode decreases the throughput of the execution pipeline.

Step Description

This section describes the steps taken in HASE to add a new type of addressing mode to an existing external architecture.

1. An initial search is carried out to check if the required addressing mode is defined for the ADD instruction.

2. The instruction set opcode table is updated with the new ADD instruction's addressing mode. This table is later used by the decode object to generate a state machine for the external architecture.

3. The *Object Editor* is used to select and modify the decode object. This involves adding an additional service to the decode object. When an ADD operation with a memory reference is retrieved from the fetch stage, the decode object sends a message to the data pipeline object to fetch the ADD instruction's operand and forwards it to the integer unit for execution.

4. The test programs are rewritten to include the ADD instruction with its enhanced addressing mode.

5. The new version of the decode object is inserted into the existing architecture using the *Architecture Editor*.

6. The Decode object is parameterised to execute ADD instructions at the ISP level of abstraction.

7. The clock period delay is increased by 0.1 (i.e. 10% of the original clock period), to simulate the extra latency caused by the new instruction. This parameter is set using the *Architecture Editor*.

8. The *Monitor Probes* are set to return statistics on the throughput and utilisation of the data and address bus between the cache object and the fetch stage object. State information is collected for the Integer and Execution unit pipelines. In particular, the architect connects a probe to the decode stage to count the simulated execution of Load and Store instructions, and calculate the total percentage of load operations the test program executed.

9. The *Graph Displayer* is viewed to determine the average CPI for the Execution Unit pipeline for the simulated test program with and without the ADD instruction's enhanced addressing mode. (Figure 5–7 shows there is a 4.5% performance improvement when the proportion of load instructions in the test program that require ADDing is 20%.

10. The *Graph Displayer* indicates that there are fewer CPI stalls generated during the simulation run for the enhanced instruction set. This is illustrated in (Figure 5–8), which shows the performance of the enhanced external architecture as a continuous line.

    This experiment ran the architecture simulation against test programs with a 33% mix of Load/Store instructions. The introduction of the ADD memory

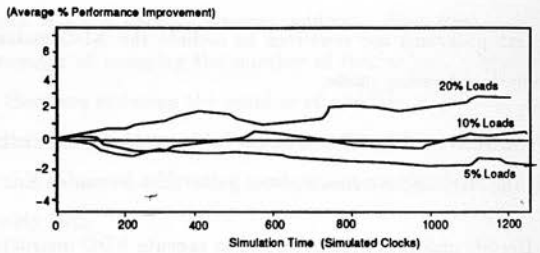


Figure 5–7: Graph Displayer: Performance Increase Against Percentage of Load Operations

addressing mode reduced the number of Load instructions required by 10%. A higher percentage of the test program was stored in the simulated instruction cache and therefore the the total number of instruction cache misses, (shown as spikes on CPI graph) decreased.

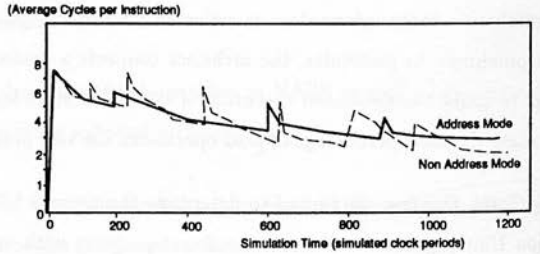


Figure 5–8: Graph Displayer: CPI Trace for Non-Addressing and Addressing ADD operation

Comments

In the simulation experiment, the set of test programs were written specifically to test a new addressing mode. Currently, simulation experiments that modify the instruction set of an external architecture restrict the HASE prototype to running small sequences of instructions. Running a large commercial benchmark

on HASE requires a flexible assembler to translate the high level grammar of the application into the new version of the assembly language program, required for the each external architecture experiment.

The external architecture experiment described here demonstrates the advantage of HASE's object oriented design. Specifically, HASE defines an architecture's instruction set as a collection of instruction objects. Each instruction object is a subclass of a particular class of instruction; for example control, load/store and arithmetic instructions. The ADD instruction described here was defined as a subclass of the arithmetic instruction. To add new behaviour to the ADD class instruction, a subclass is created which overrides the inherited behaviour of its decode method. Therefore, when the new ADD object receives the decode message from its Decode object, it performs the extra services required to handle the operand's memory address and retrieve the ADD instruction's operands.

The following attributes and services were added to the subsclass of the ADD instruction class:

SuperClassName: AddInstruction

Classname: Add(WithMemoryAddress)

Attributes:

The instance variables added to the ADD instruction object are:

- Abstraction Level
- Effective address
- Opcode value.

Services:

The services added to the ADD instruction object are:

- Calculate the effective memory address of the ADD instruction's operand

- Dispatch the operand address to the data unit object
- Dispatch the ADD opcode to the integer unit object.

5.3 Optimising Hardware/Software Interactions

This architecture experiment demonstrates how to use HASE to visualise the software/hardware interactions associated with a microprocessor architecture. Two types of optimisation techiques are investigated: Delayed Branching and Register Colouring.

5.3.1 Brief Description

The investigation begins when an architect wishes to determine how a specific sequence of assembly code executes on an architecture. An optimisation technique is examined to eliminate redundant code operations and maximise the processor's use of its resources.

5.3.2 Flow of Events

To perform this case study, the following activities are carried out:

1. The architect uses the *Arcitecture Animator* and *Graph Displayer* to identify poor uses of %CPU time.

2. The test program is rewritten to incorporate the code optimisations.

3. If a performance improvement is observed; the architect extends the implementation of the microprocessor's compiler to incorporate the code enhancement as a new optimisation technique.

5.3.3 Delayed Branching Code Optimisation

This scenario demonstrates the use of HASE to investigate a *Delayed Branch* optimisation technique.

Scenario Description

This scenario begins when a compiler writer views the performance of an architecture executing a test program. It appears that control transfer instructions are immediately followed by a drop in execution unit utilisation. The architecture under test does not have an instruction buffer, so pre-fetching cannot occur while the execution unit is calculating the control transfer instruction's target address. The compiler writer decides to insert an instruction that is independent of target address immediately after the control transfer instruction.

The compiler writer rewrites the test program and views the performance of the architecture. He writes an algorithm to seek and place independent instructions immediately after all control transfers. If an independent instruction cannot be found the compiler will insert a no-operation instruction.

Step Description

1. The *Graph Displayer* is viewed and a glitch in execution unit pipeline utilisation is observed at repeated intervals (Figure 5–9).

2. The *Architecture Animator* is used to display the state of the Decode stage and it is observed that the issuing of a control transfer instruction immediately precedes a glitch on the utilisation graph, shown in the Graph Displayer (Figure 5–10), inside the window entitled Function Unit Parallelism.

3. An independent instruction is inserted immediately after the control transfer instruction.

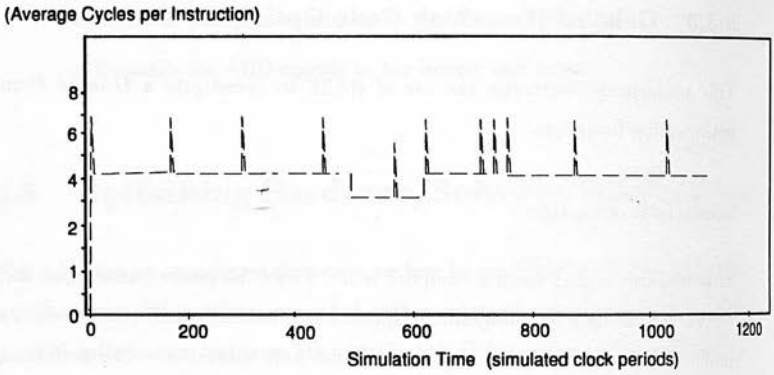


Figure 5–9: Graph Displayer: Repeated Utilisation Glitches

4. The architect re-runs the execution on the optimised code and observes that 40% of the utilisation glitches are eliminated (Figure 5–11).

5. The compiler is incorporated to include the **delayed branch** optimisation.

5.3.4 Register Colouring Code Optimisation

Scenario Description

During the animated execution of a convolution test program the architect observes that the program uses different registers for temporary variables when one can suffice. The architect applies a simple register colouring algorithm to optimise the use of the register file for temporary variables. For example, two temporary variables can share the same register if they are declared outside the scope of each other. Optimising the use of the register file during program execution reduces the memory load and store operations and therefore reduces the average number of cycles per second.

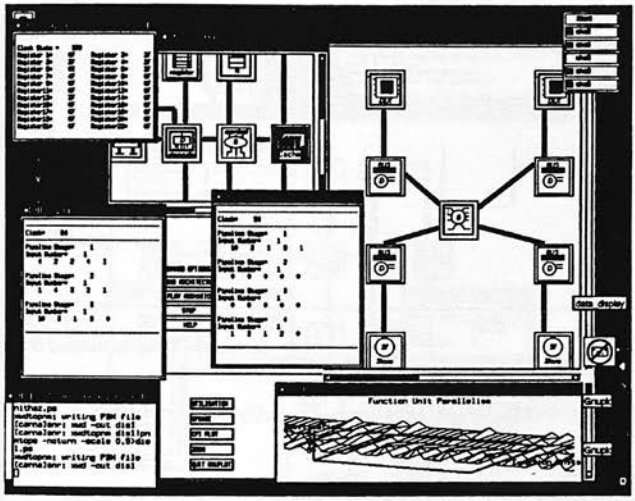


Figure 5–10: Architecture Animator: Control Transfer Instruction

Step Description

1. The *Architecture Animator* is used to view the current state of the Register File object. Figure 5–12 shows an example of using the assembler code window, shown in the bottom right hand corner of the screen, in conjunction with the register file.

2. Each step of the test program's execution is animated and the storage of each program variable is traced to its register, to determine if the allocation algorithm is making effective use of the processor's register file.

3. The simulation is rerun with an enhanced register allocation algorithm to identify whether or not the number of data cache accesses decreases during the animated execution of the test program.

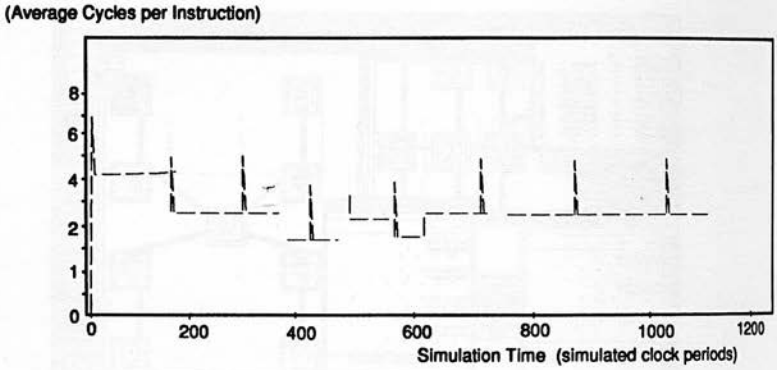


Figure 5–11: Graph Displayer: Executing Delayed Branch Instructions

Comments

During the hardware/software interaction experiments described above, the HASE prototype's *Architecture Animator* provided useful features to:

- Select a specific simulation time on the *Graph Displayer* for a low period of throughput and use the *Architecture Animator*'s assember code window to search for data dependencies in the instruction flow. For example, the first scenario used the assembler window to identify control transfer instructions creating spikes on the CPI graph, because of wrong branch predictions.

- Step through a sequence of instructions, and use the assembler code window to trace the allocation of temporary variables to registers in the register file.

- Pan across object components and display their state information during the animated execution of a test program. For example, in the second scenario the internal state of the code cache was displayed to check for load / store accesses of temporary variables that could have been saved in the register file, had the register allocation algorithm been more efficient.

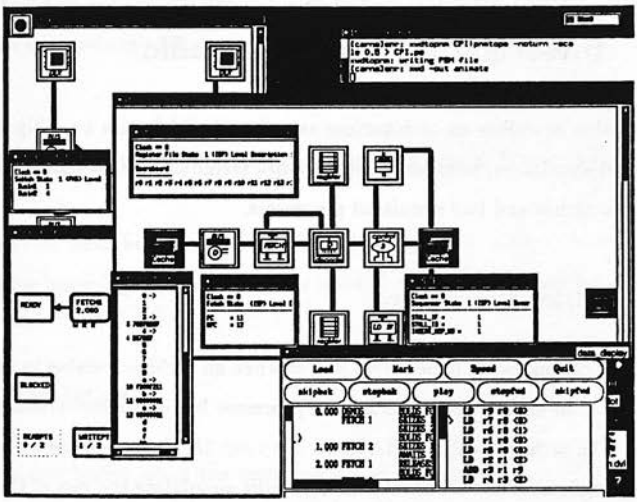


Figure 5–12: Architecture Animator: Viewing Contents of the Register File

Usage of the HASE prototype would have been more effective if:

- The *Architecture Editor* and the *Architecture Animator* were supported by the same application. Piping between the two processes made the environment slow and the *Architecture Animator* had to be updated manually after every architecture edit.

- The abstraction level of an object component was parameterised during runtime and not during the architecture edit phase. The HASE prototype requires the architect to specify the abstraction levels of each object component before compiling and linking the simulation's source code.

5.4 Investigating Network Traffic

This section describes an architecture experiment which uses a multiple abstraction level simulation, to examine the network traffic between a shared set of code and data caches and two simulated processors.

5.4.1 Brief Description

This type of simulation experiment begins when an architect wishes to investigate the impact an architectural change of a processor has on its surrounding network traffic. The architect can use HASE to carry out this investigation by simulating one processor at a low abstraction level, whilst simulating the rest of the network at a more abstract level.

5.4.2 Flow of Events

To perform this architecture experiment the following procedures are followed:

1. An aspect of a processor's architecture is selected for investigation.

2. The *Architecture Editor* is used to raise the abstraction level of component objects which do not add value to the experiment's results. The *Trace Animator* is used to distinguish the role of each entity during the simulation; suggesting which component objects should be simulated at a higher level of abstraction.

3. The *Monitor* objects are attached to component objects for collecting performance data.

4. The simulation is executed.

5. The *Architecture Animator* is used to help clarify the relationship between an architecture's internal behaviour and its surrounding Network.

5.4.3 Simulating Network Influence on a Processor

This section describes a scenario in which an architect wishes to run a HASE simulation involving component objects instantiated at different abstraction levels.

Brief Description

The scenario begins when the architect wishes to simulate two MC88110 CPUs connected to an external bus and an instruction and data cache. The architect wishes to investigate the interactions between two MC88110 microprocessors, sharing a set of data and instruction caches and a bus network controlled by a memory router component. The architect decides to simulate one of the MC88110 microprocessors at the PMS abstraction level and the second microprocessor at the ISP abstraction level.

At the ISP level, the MC88110 is simulated as six execution units operating independently and concurrently. The integer, floating-point, multiply and divide execution units perform computation operations. The data unit performs data memory accesses, while the instruction unit performs instruction fetches, sequencing and control functions.

The instruction unit fetches instruction pairs from the instruction cache, and issues instructions to their appropriate execution units. The instruction unit also executes control flow instructions, for example, branch operations. An important feature of the architecture is a history buffer, a (First In First Out) FIFO queue which records the relevant machine state at the time of an instruction issue.

The architect wishes to simulate the memory router as a simple 2 x 2 crossbar switch, which includes two input and output queues for processing incoming addresses and data.

Step Description

1. This scenario begins when the architect wishes to investigate the relationship between the history buffer size and the corresponding throughput of the crossbar switch in terms of messages per second; where each message represents either a memory address, an instruction or an operand.

2. The *Trace Animator* is used to filter out superfluous component objects from the model. The entities visible in the *Trace Animator*, will only include those that add value to the results of the simulation experiment.

3. The *Architecture Editor* is used to parameterise one of the MC88110 microprocessors to simulate at the PMS level. The second MC88110 and the crossbar switch are parameterised to simulate at the ISP level. Figure 5-13 shows an example in which two MC88110 processors are connected to their shared code and data caches. In this example, the outgoing and ingoing queue lengths for the crossbar switch are both initialised to 4 elements. The right hand window on the screen displays the icons that represent the object components simulating the MC88110 processor's behaviour at the ISP level of abstraction. The MC88110 processor's history buffer length is varied from 1 to 10 elements; where each element saves the current state of 2 issued instructions.

4. A *Measurement Probe* is attached to the bus and crossbar switch to measure the frequency of instruction fetches between the two MC88110s and their shared Instruction and Data cache components.

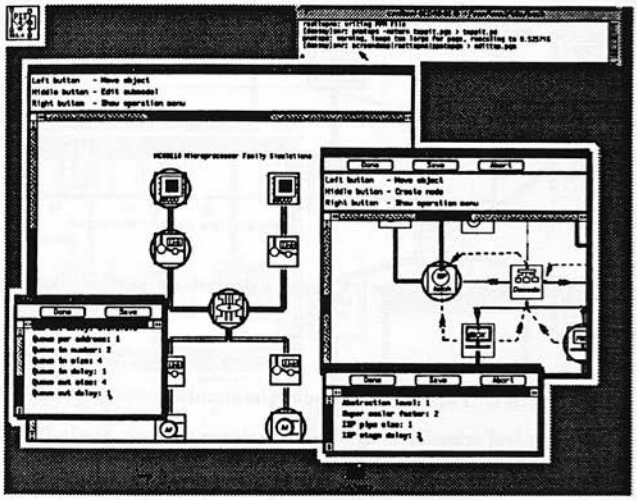


Figure 5–13: Architecture Editor: Setting a Component's Abstraction Level

5. The simulation is repeated for each history buffer size ranging between 1 and 10, and the corresponding average crossbar switch throughput is measured for each simulation run.

6. Figure 5–14 shows how the crossbar switch throughput increases from 31 simulated messages per second to 39 messages per second, as the history buffer size increases between 1 and 10 elements.

7. After the history buffer size exceeds 6 elements the throughput of the crossbar switch peaks at approximately 39 messages per second. The *Architecture Animator* is used to inspect the contents of the ISP processor's history buffer to determine any data dependencies that prevent the history buffer from progressing. Figure 5–15 displays an example of the ISP processor's history buffer receiving 2 instruction words from the fetch stage on each simulated clock cycle. The *Graph Displayer*, in the bottom right hand corner, displays

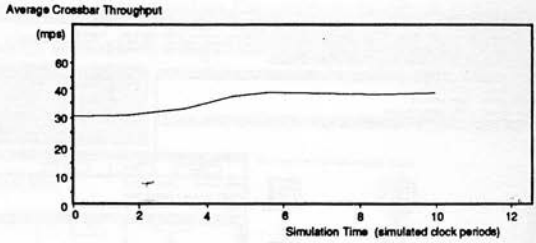


Figure 5–14: Graph Displayer: Crossbar Switch Throughput againsy History Buffer Length

the repeated CPI spikes, throughout the simulation run. The CPI spikes are due to control transfer instructions that are wrongly predicted. The failed target address predictions caused the history buffer to stall, flush its contents and refill itself, with the correct branch of instructions.

Comments

This scenario demonstrates how HASE uses polymorphism to support multiple abstraction level simulations [62] [63]. The software structure for the HASE implementation of the MC88110 network is illustrated in (Figure 5–16). This figure shows the main communications between the PMS and ISP processors, busses and crossbar switch object components.

This diagram distinguishes three types of objects: control, problem domain and interface. The interface object is responsible for returning results to the front-end animator, for example, event traces, state data, and statistics about the number of stalls in each pipeline etc. The control object shown here is responsible for coordinating all component objects to respond to the simulated DEMOS clock. A problem domain object refers to any type of object component which is relevant to the behaviour of the architecture.

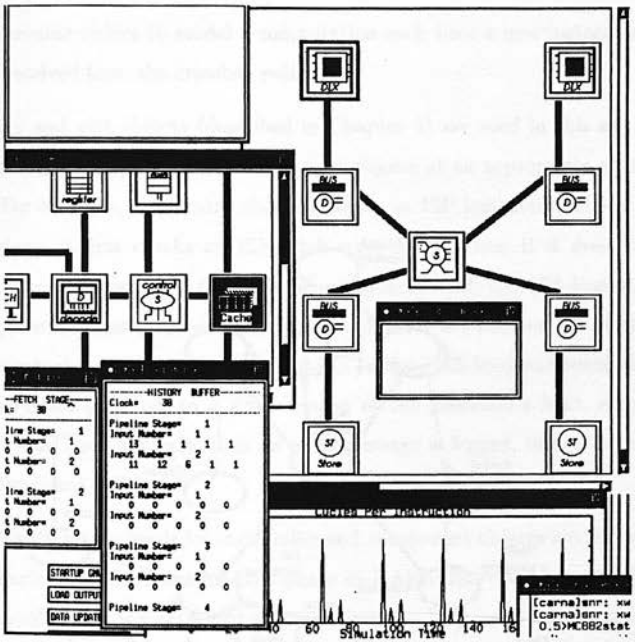


Figure 5–15: Architecture Editor: Display State of History Buffer

The control object is inherited by all the problem domain objects, but its methods are overwritten by each subclass. When the main control program of the DEMOS simulation broadcasts a start message, each component object will assign itself to an abstraction level entered from the *Architecture Editor* and behave accordingly. For example, when the ISP processor receives a start message, it instantiates all the lower level ISP component objects required to simulate its instruction execution. When the ISP processor receives the next clock message from the DEMOS simulator, it responds by attempting to fetch the next ISP instruction object from the bus object.

The PMS processor, responds to the same start and clock messages, but behaves at a higher level of abstraction. For example, it only instantiates a random

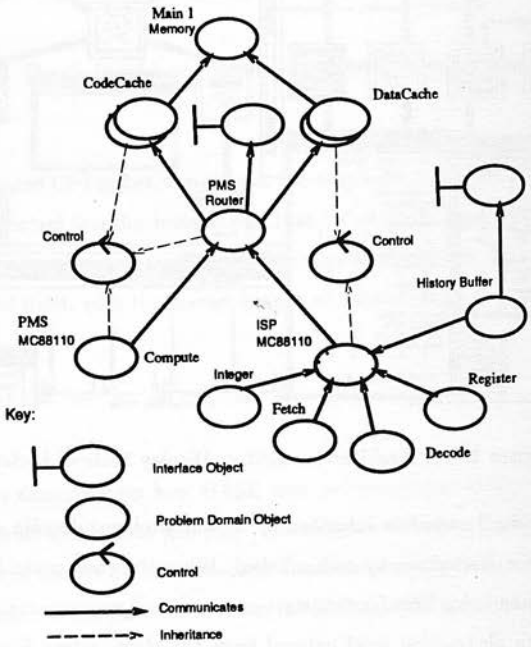


Figure 5–16: Multi Abstraction Level Simulation: Software Structure

CPI generator object to model a computation each time a new instruction word pair is received from the crossbar switch.

Entry and exit objects (described in Chapter 4) are used in this experiment to match instruction objects to component objects at an appropriate abstraction level. For example, if an entry object receives an ISP instruction object for the fetch stage, it first checks an ISP fetch component exists; if it does, it sends a *nextInstruction* message to the fetch component with the ISP instruction as an argument. If not, the entry object searches for a PMS version of the fetch component, abstracts the instruction object to the PMS level and sends the same *nextInstruction* message to a PMS version of the processor's fetch stage. If a PMS version does not exist then an error message is logged, indicating that the execution is not valid.

The abstraction levels for instruction and component objects are defined statically during the Architecture Edit phase in HASE. However, by extending the polymorphic approach described above and using a language such as smalltalk or Objective C, which supports dynamic binding, a future version of HASE can be designed to change a component's abstraction level at runtime.

This would provide the architect with the flexibility to zoom into the detail of an animated architecture component during runtime, rather than having to specify each component object's abstraction level before compiling and linking the simulation code.

Chapter 6

Conclusion

The computer architect must continue to invent new architectural techniques if he wishes to take full advantage of the recent progress in hardware technology. Specifically, the architect not only requires a thorough technical understanding of the hardware's physical limitations, he must also understand how an architecture can perform against different types of software applications. Furthermore, he must know whether his simulation is realistic and whether he can set up controlled experiments and make valid comparisons between new and existing architecture designs. Before suggesting an architecture enhancement, the architect must decide whether or not a performance improvement can justify the impact of its hardware and software cost.

This thesis contributes toward the development of a simulation environment directed specifically towards computer architects for the purpose of setting up architecture experiments and investigating hardware/software interaction.

This chapter concludes the work of this thesis and is divided into the five following sections:

- Section 6.1, *Architect Requirements* - concludes the essential features of a hardware/software simulation environment.

- Section 6.2, *Object Oriented Design* - summarises the advantages of using object oriented design for the simulation and animation of a computer architecture.

- Section 6.4, *Architecture Experiments* - summarises the set of architecture experiments used to demonstrate the operation of HASE.

- Section 6.3, *HASE Prototype Performance Evaluation* - its role as a prototype and discusses some of the advantages and disadvantages of its development environment.

- Section 6.5, *Future Work* - describes how the HASE prototype will be implemented as a production version.

6.1 Architecture Requirements

A survey of existing academic and commercial simulation was carried out. From this survey, described in section 1.4, it is clear that logic simulators are well established. Such tools, including Mentor Graphics and more recently VHDL, support a well defined environment for detailed logic design. At the logic level these tools support a logic level component library, for example providing adders, multiplexers and standard off the shelf microprocessors. However, for this survey it appears that logic design environments do not provide similar sorts of libraries for architecture components, for example, pipelines, history buffers, and general purpose function units.

The survey investigated existing academic and commercial simulators that did specialise in architecture simulation. Amongst simulation environments such as STATEMENT and i-logix, the most popular commercial architecture simulator is the SES/Workbench. The deficiency with the SES/Workbench is that the details

of an architecture's behaviour and its associated hardware/software interaction are hidden by the mechanics of the simulation. This deficiency was confirmed by the ARM microprocessor simulation demonstrated at UMIST.

In HASE, although common attributes and services are inherited from parent classes; each internal and external architecture component is directly associated with a line of code and is symbolised by an icon. This simplifies the task of establishing an object repository to store architecture components and enables the architect to identify, observe and if necessary, control the interactions between components during an architecture simulation.

6.2 Object Oriented Design

There are three main advantages of using an object oriented simulation approach rather than a procedural approach:

- For some defined abstraction level, if an architect can clearly define the attributes and services of a physical internal or external component in the architecture, then it can be mapped directly to a software object.

- If a set of component objects have common attributes and services, then these facets can be factored out and abstracted into a superclass. The superclass can be specialised to create a new set of subclasses, that directly model physical architecture components. This maintains consistency between component objects and saves the architect unnecessary effort.

- An object component's behaviour, i.e. its attributes and services, can be specialised further to model its physical component at a lower level of detail. This notion maps conveniently onto Flynn's classifying of abstraction levels, described in section 4.1.1.

The main disadvantage of using an object oriented approach is that objects are expensive on computer resources, and this sometimes impairs a simulation's runtime performance.

6.3 Architecture Experiments

The HASE prototype functionality was demonstrated on a variety of internal and external architecture, software/hardware interaction and multiple abstraction level simulation experiments. Each type of architecture experiment was illustrated by scenarios which provided a step description of how to use the HASE prototype.

6.4 HASE Prototype Performance Evaluation

The purpose of developing a prototype was to demonstrate how to apply an object oriented approach to computer architecture simulation and animation. The HASE prototype implemented an object oriented design using the DEMOS programming environment. DEMOS proved to be a suitable prototyping language because it was well established and was already supported by various graphical frontend tools; for example, PIT and GSS.

6.4.1 Advantages of the HASE Prototype

The HASE prototype has demonstrated how:

- The behaviour of an architecture can be factored out into superclasses; for example, the Component class encapsulates the synchronous and asynchronous communication protocols between architectures, inherited by all instantiated pipeline function units.

- Polymorphism can be used to design multiple abstraction level simulations. For example, a component's class can be subclassed to respond to the same message inherited from its parent but behave at a lower abstraction level.

- To use an *Architecture Editor* to symbolise component objects as a menu of icons. Each icon represents a software class which maps directly to a specific physical architecture component. This simple one to one mapping between software and hardware makes it easier to validate the correctness of an architecture simulation.

- A GDL is used to specify the behaviour of new component objects that an architect requires for a simulation.

- A class's behaviour can be described graphically in terms of an activity diagram. The activity diagram provides a convenient means of documenting a library of component objects.

- An *Architecture Animator* is useful for investigating a processor's hardware/software interaction; for example, studying the merits of a particular program's register allocation algorithm.

6.4.2 Disadvantages of the HASE Prototype

The HASE prototype programming environment is unsuitable for the following reasons:

- The support environment is fragmented. This is because the HASE prototype used and modified existing applications to prove a concept. For example, the *Object Editor* and the *Architecture Editor* are developed from the GSS programming environment and the *Architecture Animator* and *Trace Animator* are developed using Motif and SRGP respectively.

- Adding a new architecture component to the *Architecture's menu* required the use of a GDL to specify the link refusals between co-existing components in the architecture library.

- The *Architecture Editor* Code generation service only supported a framework and does not produce a compilable and linkable simulation source code.

- The *Architecture Editor* communicated with the *Architecture Animator* via a standard UNIX pipe. This added complexity is not necessary if the both tools are part of the same application.

- The DEMOS simulation code is compatible with the GSS support tool, but DEMOS is derived from SIMULA, which is characteristically slow and memory demanding, compared to more recently developed object oriented languages, such as C++ and Objective C.

6.5 Future Work

As explained in the previous section, the implementation of the HASE prototype version was fragmented into various technologies and software environments, because its purpose was primarily to prove design concepts, and illustrate an object oriented approach to simulating and animating hardware/software interaction. This section describes the main future tasks that will develop HASE from a prototype into a working environment. These tasks can be divided into four main sections:

- Section 6.5.1, *Create Architecture Components*, this involves converting the architecture class hierarchy from DEMOS to Sim++.

- Section 6.5.2, *Develop a Component Object Repository*, this involves developing a component class browser and a database to store and provide efficient access to a library of architecture object components.

- Section 6.5.3, *Develop a Frontend*, this involves developing a single GUI interface to incorporate the tools that were designed and built for the HASE prototype.

- Section 6.5.4, *Develop a Distributed Simulation Environment*, this involves setting up a distributed simulation environment to improve the performance of large, computation intensive, simulations.

6.5.1 Create Architecture Components

The HASE prototype established a class hierarchy to describe a selection of microprocessor architectures, namely the Motorola 88000 family. This class hierarchy included abstract classes that modelled the generic behaviour of memory components, data busses, fetch and decode units, execution pipelines and sequencers and register files.

The class hierarchy was implemented in DEMOS, and although SIMULA was a good prototyping language (and was already established) and worked well with the *PIT* and *Trace Animator*, the disadvantage of DEMOS is that its runtime performance is impaired by its demanding memory requirements. Consequently, the class hierarchy of the real version of HASE will be implemented in the Sim++ simulation language. Although Sim++ is less established, it is derived from the C++ object oriented programming language and provides support for distributed simulation.

6.5.2 Develop a Component Object Repository

The component objects and their associated specification documents will be stored in a component object repository. The component object repository will be developed to include the following facilities:

- A Component Class Browser which features facilities to search for a component object at a specified abstraction level. It will support facilities to filter out required facets of information about a selected component object.
- An object oriented database, which may be developed from a third party product, for example, Object Store, and provides efficient access to all component objects in the repository.
- A query language to allow an architect to ask the database if a component object already exists that matches a required class's interface and behaviour. A component object will be returned that best fits the required component characteristics.

6.5.3 Develop a Frontend

The HASE prototype is fragmented into 4 main tools; the *Object Editor*, the *Architecture Editor*, the *Trace Animator* and the *Architecture Animator*, described earlier in chapters 3 and 4. Future work is therefore required to integrate these simulation and animation tools into one consistent, application, implemented using the standard Motif programming environment.

6.5.4 Develop a Distributed Simulation Environment

During the prototyping phase of HASE, a number of component objects were created and built in Sim++ to investigate the potential performance improvement

gained from running a distributed simulation. Initial distributed simulations involved a cluster of four Sparc Workstations. The entities involved in the simulation had to be sufficiently busy before farming them to a different processor was justified. Otherwise the extra communication cost between processors impeded the possibility of performance gain.

Future work in this area may include developing a environment to distribute the large Sim++ architecture simulations across a network of workstations, in such a way that the inter-process communication cost between workstations is minimised.

Appendix A

Instruction Set

Appendix A contains a cross section of the DLX instruction set [32]. The instruction set presented here was modified by simulation experiments to demonstrate Decode object of the HASE prototype.

Table A–1: Logical Instructions

Mnemonic	Encoding				
	31 26	25 21	20 16	15 0	
andi	0 0 1 1 0 0	rs1	rd	SIMM-16	
ori	0 0 1 1 0 1	rs1	rd	SIMM-16	
xori	0 0 1 1 1 0	rs1	rd	SIMM-16	
	31 26	25 21	20 16	15 11	10 0
and	0 0 0 0 0 0	rs1	rs2	rd	0 0 0 0 0 1 0 0 1 0 0
or	0 0 0 0 0 0	rs1	rs2	rd	0 0 0 0 0 1 0 0 1 0 1
xor	0 0 0 0 0 0	rs1	rs2	rd	0 0 0 0 0 1 0 0 1 1 0

rd: Destination Register (general purpose)

rs1: Source 1 Register (general purpose)

rs2: Source 2 Register (general purpose)

SIMM-16: 16-bit Signed Immediate Operand

Table A–2: Integer Arithmetic Instructions

Mnemonic	Encoding																															
	31		29	28		26	25		21	20		16	15																			0
addi	0	0	1	0	0	0	rs1			rd			SIMM-16																			
addui	0	0	1	0	0	1	rs1			rd			SIMM-16																			
subi	0	0	1	0	1	0	rs1			rd			SIMM-16																			
subui	0	0	1	0	1	1	rs1			rd			SIMM-16																			
s_i	0	1	1	COND			rs1			rd			SIMM-16																			
s_ui	1	1	0	COND			rs1			rd			SIMM-16																			
	31					26	25		21	20		16	15		11	10													3	2		0
s_u	0	0	0	0	0	0	rs1			rs2			rd		0	0	0	0	0	0	0	1	0	COND								
mult†	0	0	0	0	0	0	fs1			fs2			fd		0	0	0	0	0	0	0	1	1	0	0	0						
multu†	0	0	0	0	0	0	fs1			fs2			fd		0	0	0	0	0	0	0	1	1	0	0	1						
div†	0	0	0	0	0	0	fs1			fs2			fd		0	0	0	0	0	0	0	1	1	0	1	0						
divu†	0	0	0	0	0	0	fs1			fs2			fd		0	0	0	0	0	0	0	1	1	0	1	1						
add	0	0	0	0	0	0	rs1			rs2			rd		0	0	0	0	0	0	1	0	0	0	0	0						
addu	0	0	0	0	0	0	rs1			rs2			rd		0	0	0	0	0	0	1	0	0	0	0	1						
sub	0	0	0	0	0	0	rs1			rs2			rd		0	0	0	0	0	0	1	0	0	0	1	0						
subu	0	0	0	0	0	0	rs1			rs2			rd		0	0	0	0	0	0	1	0	0	0	1	1						
s_	0	0	0	0	0	0	rs1			rs2			rd		0	0	0	0	0	0	1	0	1	COND								

rd: Destination Register (general purpose)

rs1: Source 1 Register (general purpose)

rs2: Source 2 Register (general purpose)

fd: Destination Register (floating point)

fs1: Source 1 Register (floating point)

fs2: Source 2 Register (floating point)

SIMM-16: 16-bit Signed Immediate Operand

COND: 000 – eq 001 – ne 010 – lt 011 – gt 100 – le 101 – ge

† Not implemented in hardware, vectored directly to *software emulation* code

Table A–3: Load/Store Instructions

Mnemonic	Encoding			
	31 26	25 21	20 16	15 0
lhi	0 0 1 1 1 1	0 0 0 0 0*	rd	SIMM-16
lb	1 0 0 0 0 0	rs1	rd	SIMM-16
lh	1 0 0 0 0 1	rs1	rd	SIMM-16
lw	1 0 0 0 1 1	rs1	rd	SIMM-16
lbu	1 0 0 1 0 0	rs1	rd	SIMM-16
lhu	1 0 0 1 0 1	rs1	rd	SIMM-16
lf†	1 0 0 1 1 0	rs1	fd	SIMM-16
ld†	1 0 0 1 1 1	rs1	fd	SIMM-16
sb	1 0 1 0 0 0	rs1	rs2	SIMM-16
sh	1 0 1 0 0 1	rs1	rs2	SIMM-16
sw	1 0 1 0 1 1	rs1	rs2	SIMM-16
sf†	1 0 1 1 1 0	rs1	fs2	SIMM-16
sd†	1 0 1 1 1 1	rs1	fs2	SIMM-16

rd:	Destination Register	(general purpose)
rs1:	Source 1 Register	(general purpose)
rs2:	Source 2 Register	(general purpose)
fd:	Destination Register	(floating point)
fs2:	Source 2 Register	(floating point)
SIMM-16:	16-bit Signed Immediate Operand	
†	Not implemented in hardware, vectored directly to *software emulation* code	
*	These bits are not decoded	

Table A–4: Control Transfer Instructions

Mnemonic	Encoding
	31 26 25 0
j	0 0 0 0 1 0 | SIMM-26
jal	0 0 0 0 1 1 | SIMM-26
	31 26 25 21 20 16 15 0
beqz	0 0 0 1 0 0 | rs1 | 0 0 0 0 0* | SIMM-16
bnez	0 0 0 1 0 1 | rs1 | 0 0 0 0 0* | SIMM-16
bfpt‡	0 0 0 1 0 0 | 0 0 0 0 0* | 0 0 0 0 0* | SIMM-16
bfpf‡	0 0 0 1 0 1 | 0 0 0 0 0* | 0 0 0 0 0* | SIMM-16
	31 26 25 21 20 16 15 9 8 0
rfe	0 1 0 0 0 0 | 0 0 0 0 0* | 0 0 0 0 0* | 0 0 0 0 0 0 0* | 0 0 0 0 0 0 0 0 0*
trap	0 1 0 0 0 0 | 0 0 0 0 0* | 0 0 0 0 0* | 0 0 0 0 0 0 0* | VEC-9
	31 26 25 21 20 16 15 0
jr	0 1 0 0 1 0 | rs1 | 0 0 0 0 0* | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0*
jalr	0 1 0 0 1 1 | rs1 | 0 0 0 0 0* | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0*

rd: Destination Register (general purpose)

rs1: Source 1 Register (general purpose)

rs2: Source 2 Register (general purpose)

SIMM-16: 16-bit Signed Immediate Operand

SIMM-26: 26-bit Signed Immediate Operand

VEC-9: Vector number from the start of the page address in the vector base register

‡ Not implemented in hardware, vectored directly to *software emulation* code

* These bits are not decoded

Table A–5: Floating-Point Arithmetic Instructions

Mnemonic	31–26	25–21	20–16	15–11	10–0 (10–3)	2–0
addf	0 0 0 0 0 1	fs1	fs2	fd	0 0 0 0 0 0 0 0 0 0 0	
subf	0 0 0 0 0 1	fs1	fs2	fd	0 0 0 0 0 0 0 0 0 0 1	
multf	0 0 0 0 0 1	fs1	fs2	fd	0 0 0 0 0 0 0 0 0 1 0	
divf	0 0 0 0 0 1	fs1	fs2	fd	0 0 0 0 0 0 0 0 0 1 1	
addd	0 0 0 0 0 1	fs1‡	fs2‡	fd‡	0 0 0 0 0 0 0 0 1 0 0	
subd	0 0 0 0 0 1	fs1‡	fs2‡	fd‡	0 0 0 0 0 0 0 0 1 0 1	
multd	0 0 0 0 0 1	fs1‡	fs2‡	fd‡	0 0 0 0 0 0 0 0 1 1 0	
divd	0 0 0 0 0 1	fs1‡	fs2‡	fd‡	0 0 0 0 0 0 0 0 1 1 1	
_f	0 0 0 0 0 1	fs1	fs2	fd	0 0 0 0 0 0 1 0	COND
_d	0 0 0 0 0 1	fs1‡	fs2‡	fd‡	0 0 0 0 0 0 1 1	COND

fd: Destination Register (floating point)

fs2: Source 2 Register (floating point)

COND: 000 – eq 001 – ne 010 – lt 011 – gt 100 – le 101 – ge

‡ Not implemented in hardware, vectored directly to *software emulation* code

‡ *Double-precision* floating point registers should be aligned to (even) floating point register pairs

* These bits are not decoded

Appendix B

Assembly Test Programs

Appendix B includes a listing of the optimised test programs that were run against the HASE prototype whilst conducting hardware / software interaction experiments.

B.1 Convolution Program: Optimised using Delayed Branching

```
LD r9 0 0
LD r7 0 0
LD r6 0 1
LD r6 0 0
LD r2 1 2
ADD r3 r1 9
LD r4 r3 2
MUL r2 r2 r4
ADD r5 2 5
SUB r1 1 1
ADD r12 6 1
```

```
ST r4 r9 r23
ST r5 r7 FF
SUB r7 r7 1
SUB r9 r9 1
ADD r14 r6 r9
BCND 0 r14 -16
ST r5 r6 FF
LD r7 0 0
LD r5 0 0
LD r2 1 2
ADD r3 r1 9
LD r4 r3 2
MUL r2 r2 r4
ADD r2 2 5
SUB r1 1 1
ADD r12 6 1
BCND 0 r12 -8
ST r4 r9 r23
SUB r7 r7 1
SUB r6 r9 1
ADD r14 r6 r9
BCND 0 r14 -16
MUL r2 r2 r4
ADD r5 r2 r5
SUB r11 r1 r1
ADD r12 r6 r1
```

B.2 Convolution Program: Optimised using Register Colouring

```
LD r6 0 1
LD r6 0 0
LD r2 1 2
ADD r3 r1 9
LD r4 r3 2
MUL r2 r2 r4
ADD r5 2 5
SUB r1 1 1
ADD r6 6 1
ST r4 r9 r23
ST r2 r7 FF
SUB r7 r7 1
SUB r9 r9 1
ADD r6 r6 r9
BCND 0 r6 -16
ST r5 r6 FF
LD r6 0 0
LD r5 0 0
LD r2 1 2
ADD r3 r1 9
LD r4 r3 2
MUL r2 r2 r4
ADD r5 2 5
SUB r1 1 1
ADD r6 6 1
```

```
BCND 0 r6 -8
ST r4 r9 r3
SUB r7 r7 1
SUB r9 r9 1
ADD r6 r6 r9
BCND 0 r14 -16
MUL r2 r2 r4
ADD r5 r2 r5
SUB r2 r1 r1
ADD r4 r6 r1
```

Bibliography

[1] A. Agarwal, M. Horowitz and J. Hennessy, *"Analytical Cache Model"*, ACM Transactions on Computer Systems, Vol. 7, No. 2, May 1989.

[2] G. H. Amdahl, *"Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities"*, Spring Dt Comp Conference, AFIPS Conference, Proc, vol 30, pp 483, 1967.

[3] N. Armitage, *Optimisation Methods*, Unisoft, 28 July, 1988.

[4] M. Barbacci, *"Instruction Set Processor Specifications (ISPS): Notation and its Applications"*, IEEE Transactions Computers, Jan. 1981, pp. 24-40.

[5] E. Barber & P. Hughes, *"Evolution of the Process Interaction Tool – A Graphical Editor for DEMOS"*, Proc. 17th Simula Users' Conference, Association of Simula Users, 1990.

[6] C. G. Bell, D. P. Siewiorek & A. Newell, *"Computer Structures: Reading and Examples"*, McGraw-Hill, 1971.

[7] J. Birtwistle, *"DEMOS: Discrete Event Modelling On Simula"*, Prentice-Hall, 1985.

[8] G. Booch, *"Object Oriented Development"*, IEEE Transactions on Software Engineering, Vol 12, No. 2, 1986.

[9] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, *"Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems"*, International Journal of Computer Simulation, August, 1992.

[10] CACI Simulation Tools, NETWORK 11.5, CACI Corporation, 1988

[11] R. D. Chamberlain & M.A. Franklin, *"Hierarchical Discrete Event Simulation on Hypercube Architectures"*, IEEE Micro, August 1990.

[12] T. C. Chen, *"Parallelism, pipelining and Computer Efficiency"*, Computer Design, vol 10, pp 69-74, 1971.

[13] R. J. Chevance, *"An Evaluation Methodology for Microprocessor and System Architecture"*, Computer Architecture News, Vol 20, No. 3, June 1992.

[14] A. T. Clementson, *"Extended Control and Simulation Language-Computer Aided Programming System"*, Lucas Institute for Engineering Production, University of Birmingham.

[15] A. Cota & R. G. Sargent, *"An Algorithm for Parallel Discrete Event Simulation Using Common Memory"*, 22nd Annual Simulation Symposium, 1989.

[16] B. S. Davie, *Hardware Description Languages: Some Recent Developments* EUCSD Report, CSR-198-86, 1986.

[17] Davie B. S., *"A Formal, Hierarchical Design and Validation Methodlogy for VLSI"*, PhD thesis, Department of Computer Science, University of Edinburgh, Oct 1988.

[18] J. B. Evans, *"Structures of Discrete Event Simulation"*, Ellis Horwood Ltd., 1988.

[19] D.G. Evans & D. Morris *"Applying Modelling to Computer Systems"*, IFIP 'Codes Workshop' May 18th 1992.

[20] M.K.Farrens & A. R. Plezkun, *Improving Performance of Small On-Chip Instruction Caches*, The 16th Annual International Symposium on Computer Architecture, 1989.

[21] M. J. Flynn, *"Detection and Parallel Ezecution of Independent Instructions"*, IEEE Tranactions on Computing, vol C-19, pp 889-895, Oct. 1970.

[22] M. J. Flynn, *"Some Computer Organisations and Their effectiveness"*, IEEE Tranactions on Computing, Vol C-21 No. 9, Sept 1972

[23] M. J. Flynn & C. L. Mitchell, *"A Workbench for Computer Architects"*, IEEE Design and Test, Feb. 1988.

[24] K. Foster, *"Uncoupling Central Processor and Storage Device Speeds"* Computing Journal, vol 14, pp 45-48, Feb. 1971.

[25] W. R. Franta, *"Process View of Simulation"*, North Holland, 1977.

[26] A.D. George, *"Simulating Microprocessor-Based Parallel Computers Using Processor Libraries"*, Simulation, February 1993.

[27] U. O. Gargliardi, *"Report of workshop 4- Software Related Advances in Computer Hardware"*, Proc. Symposiaum on the High Cost of Software, Menlo Park Calif., pp99-120, 1973.

[28] S. Ghosh, *"Using Ada as a HDL"*, IEEE Design and Test of Computers, Feb 1988.

[29] W. J. Gray, *"Simulation Principles and Methods"*, Winthrop Publishers, Inc., 1980.

[30] W. Handler, *"The Impact of Classification Schemes on Computer Architecture"*, Proc International Conference on Parallel Processing, Aug. pp7-15, 1977.

[31] D. Harel, H. Lachover, A. Naamad, A. Pnueli, & R. Sherman, *"STATEMATE: A Working Environment for the Development of Complex Reactive Systems"*, IEEE Trans on Software Engineering, Vol. 16, No. 4, 1990.

[32] J. L. Hennessy and D. A. Patterson, *"Computer Architecture: A Quantitative Approach"* Morgan Kaufmann, Inc. 1990.

[33] L. Higbrie, *"Quick and Easy Cache Performance Analysis"*, Digital Equipment Corporation, 1989.

[34] M. D. Hill *"A Case for Directed-Mapped Caches"*, IEEE Computer, Vol 21, No. 12, pp 25-40, Dec 1988.

[35] M. D. Hill *"Adlib: User Manual Technical Report"*, 177 Comp Sys Lab, Stanford University, 1970.

[36] Lawrence Huang, Kyeongsoon Cho, Lawrence Huang, Derek Beatty & Karl Brace, *"User's Guide to COSMOS"*, 20 April 1989.

[37] D. C. McCrackin, *"Eliminating Interlocks in Deeply Pipelined Processors by Delay Enforced Multistreaming"*, IEEE Trans on Comp VOL. 40, No. 10, Oct 1991.

[38] MENTOR Graphics, *"An Introduction to Digital Simulation"*, April 1989

[39] W. W. Huwu and P. P. Chang, *"Achieving High Instruction Cache Performance with an Optimised Compiler"*, The 16th Annual International Symposium on Computer Architecture, 1989.

[40] JADE, *"Sim++: A Discrete Event Simulation Language"*, High Performance Simulation Software, Release 2.3 (Beta), 1988.

[41] R. N. Ibbett, *"The Architecture of High Performance Computers"* Macmillan Conputer Science Sieries, 1982.

[42] J. Su & P. R. Ritter, *"Experience in Testing Motif Interface"* IEEE Software, March 1991.

[43] D. R. Jefferson *"Virtual Time"* ACM Transactions on Programming, Vol. 7, No. 3, pp. 404-425, July 1985.

[44] M. Laird, *"A Comparison of Three Current Superscalar Designs"*, Computer Architecture News, Vol 20, No3, June 1992.

[45] L. Lamport, *"Time, Clocks and the Ordering of Events on a Distributed System"*, ACM 21, pp 558-556, 7 July 1978.

[46] K. Marakami, N. Iric, M. Kuga & S. Tormita, *"SIMP- A Novel High Speed Single Processor Architecture"*, The 16th Annual International Symposium on Computer Architecture, 1989.

[47] C. L. Mitchell & M. J. Flynn, *"A Workbench for Computer Architects"*, Design and Test of Computers, February 1988.

[48] C. Mitchell, M. Flynn & H. Mulder, *"And Now a Case for More Complex Instruction Sets"*, IEEE Computer, Vol. 20, No. 9, pp. 71-83, September 1987.

[49] *"Modula 3: User Manual"*, Digital Equipment Corporation, Palo Alto, California.

[50] A. Mullarney & J. West, *"Modsim: A Language for Distributed Simulation"*, Distributed Simulation pp 155-157, 1988.

[51] J.A. Nestor, *"Visual Register-Transfer Description of VLSI Microarchitectures"*, IEEE Transactions on Very Large Integrated (VLSI) Systems, VOL. 1, No. 1, March 1993.

[52] Nigel Topham & Douglas Rogers, *"Implementing a Practical Context Flow Machine"*, Internal Report, Computer Science Department, University of Edinburgh, 1989.

[53] C. A. Petri, *"Kommunikation mit Automaten"*, Schriften des Institut fuer Instrumentelle Mathematik, Bonn.

[54] B. J. Pierce, *"Type-Theoretic Foundation for Object Oriented Programming"*, Lecture Notes for LFCS Short Cources, Computer Science, Edinburgh University, May 1992.

[55] R. J. Pooley, *"An Introduction to Programming in Simula"*, Blackwell Scientific Publications, 1987.

[56] R.J. Pooley and M.W. Brown, *"Improved Methods for Performance Engineering"* January 1988, Department of Computer Science, University of Edinburgh, January 1988.

[57] R.J. Pooley and M. W. Brown, *"A Diagramming Paradigm for the Hierarchical Process Oriented Discrete Event Driven Simulation"*, Internal Report, Department of Computer Science, University of Edinburgh, January 1988.

[58] R.J. Pooley, *"An Experimental Tool for an Integrated Modelling Support Enviroment, its Role and Design"*, Internal Report, Department of Computer Science, University of Edinburgh, September 1988.

[59] C.V. Ramamoorthy, *"Pipeline Architectures"* Computing Surveys, Vol. 9, No 1, March 1977.

[60] J. Reddi, *"A Conceptual Framework for Computer Architecture"*, Computing Surveys, Vol 7, 1976.

[61] A.R. Robertson & R. N. Ibbett, *"Simulation of the MC88000 Microprocessor System on a Transputer Network"*, Lecture Notes in Computer Science, ED-MCC2, Springer-Verlag, April 1991.

[62] A.R. Robertson & R. N. Ibbett, *"A Hierarchical Architectural Simulation Environment"*, UKSS, A EUROSIM Conference, September 1993.

[63] A.R. Robertson & R. N. Ibbett, *"MINITRACK: Fast Simulation of Computer Architecures"*, Proceedings: 27th Hawaii International Conference on System Sciences, January 3-5, 1994.

[64] F. B. Schneider *"Synchronisation in Distributed Programs"*, ACM Transaction, pp 179-195, April 1982, pp 179-195.

[65] K. Sheehan & M. Esslinger, *"The SES/sim Modelling Language,"*, The Society for Computer Simulation, San Diego, CA, July 1989.

[66] A. J. Smith, *"Cache Memories"* Computing Surveys, Vol. 14, No. 3, September 1982.

[67] A. J. Smith, *"Line (Block) Size Choice for CPU Caches"*, IEEE Transactions on Computers, C-36-9, pp 1063-1075, Sept (1987).

[68] P. Steinkiste, *"The Impact of Code Density on Instruction Cache Performance"*, The 16th Annual Int. Symposium on Computer Architecture, 1989.

[69] H. S. Stone, *"Introduction to Computer Architecture"* SRA Computer Science Series, 1975.

[70] B. Stroustrup, *"The C++ Programming Language"*, Addison-Wesley, 1986.

[71] K. D. Tocher, *"Some Techniques for Model Building"*, Proc. IBM Scientific Computing Symposium on Simulation Models and Gaming, New York, 119-155.

[72] B. Tuck, *"Users Turn to Graphics for High Level System Specification"*, Computer Design, March 1993.

[73] B. Unger, D. Jefferson, *"Distributed Simulation"*, Simulation Series, Vol. 19, No. 3, July 1988..

[74] C. Uppal, *"The Integrated Modelling Support Environment Project"* R2.2 - 3 Version 4, 23 January 1991.

[75] B. Wilkerson & L. Wiener & R. Wirfs-Brock, *"Designing Object-Oriented Software"*, Prentice Hall, 1990.

[76] T. Williams, *"Performance Analysis Spots Hardware/Software Bottlenecks"*, Computer Design, October 1992.

[77] P. Wisskirchen, *"Object Oriented Graphics"*, Springer-Verlag, 1990.

[78] W. Wolf, *"Object Oriented Programming for CAD"*, IEEE Design and Test of Computers, March 1991.

[79] B. P. Zeigler, *"Theory of Modelling and Simulation"*, Wiley, New York.