Developing a Simulator for the Hatfield Superscalar Processor

Department of Computer Science
Technical Report No. 172

University of Hertfordshire

R. Collins
December 1993

Abstract

Researchers at the University of Hertfordshire's department of Computer Science are currently investigating the possibility of scheduling code at compile time for a new family of superscalar processors. This project has two main elements, the first being the specification of the architectural model for the superscalar processors and the development of a parametrised Superscalar Simulator program. The second element is an Instruction Scheduler program that will apply a global scheduling algorithm to the superscalar source assembly code. The overall objective of the project is increase and exploit the amount of instruction-level parallelism available by applying scheduling techniques at compile time, rather than by using the processor's hardware at run time. The Instruction Scheduler can take a global view of the whole source program and move code over large distances to expose the parallelism available. A conventional superscalar processor can only move code over small distances to enhance the available parallelism due to the limited 'window' of code that it examines at run time. By scheduling the code at compile time, the need for out-of-order instruction execution is eliminated and the processor's hardware can be greatly simplified leading to a 'minimal' superscalar.

This report deals specifically with the design and development of the Superscalar Simulator program and gives a detailed explanation of the main features of the architectural model adopted. The Hatfield Superscalar Processor (HSP) being developed can potentially execute many instructions in parallel and is designed to exploit the instruction-level parallelism that is present in general purpose computer programs. Features of the HSP's instruction set enable the parallelism detected by the Instruction Scheduler to be encoded into sequential source code so that the Superscalar Simulator can reconstruct the groups of instructions that can be safely executed in parallel. The sequential source code is compatible with all members of the HSP family and allows them to achieve high "speed-ups" over a normal RISC processor. A parametrised Superscalar Simulator is developed which enables a whole range of processor implementations to be studied and compared. The objective of the Superscalar Simulator is to run source code that has been passed through the Instruction Simulator and to identify the hardware features required that achieve near-optimal performance from a suite of general-purpose benchmark programs.

The report sets the HSP in the context of other recently studied approaches to exploiting instruction-level parallelism and discusses in detail the development of the Superscalar Simulator program. Initial results are given that show an encouraging degree of 'speed-up' over a conventional RISC type processor, but the full power of the Superscalar Simulator will not be realised until an Instruction Scheduler program has also been developed. The Simulator has many user-selectable features for studying a program's behaviour and can give a variety of statistical data to help analyse the results for a particular program run. Appendices are included that give the full instruction set specification for the Hatfield Superscalar Processor and a complete user's guide for operating the Superscalar Simulator program.

Contents

1 Background

1.1 VLIW Machines

One method of realising instruction-level parallelism is to schedule normal short RISC-like instructions into Very Long Instruction Words (VLIW) which are then executed, one per cycle, on a corresponding VLIW processor. Short instructions that can be executed in parallel with each other are placed in the same VLIW, with any unused space in the VLIW being padded out by NOP instructions. One advantage in this approach lies in the ability of an instruction scheduler to take a global view of the whole program at compile time so it can move code over great distances in order to realise extra parallelism. The scheduler must have detailed information about the target processor and its associated instruction latencies before it can correctly schedule the source code obtained from the compiler. By carrying out all the code motion at compile time, the processor hardware is greatly simplified and it can adopt an in-order policy for instruction issue. A HARP VLIW processor **[4,5]** has been developed at the University of Hertfordshire with the capability to execute four short instructions in parallel and much work has been done on developing suitable global scheduling algorithms that can exploit its potential for parallelism **[7,15]**.

The main disadvantage of the VLIW approach is that code is scheduled for one particular processor implementation only and the source code must be recompiled if a different target processor is to be used. Users of computers understandably show a great reluctance to change their compiled program code when they want to update their hardware. Another problem is the increased size of the static code due to the need for NOPs to pad out unused portions of VLIWs. Finally, some algorithm, however simple, for predicting conditional branch behaviour must be assumed at compile time that may prove incorrect at run time. There is a consequential loss in performance as the machine discards instructions that were fetched from the wrong stream to take account of the mis-prediction. Some research **[17]** suggests that high accuracy in branch prediction could be a limiting factor to the overall parallelism achievable by both VLIW and superscalar machines.

1.2 Superscalars

An alternative strategy adopted by processor manufacturers **[13,14]** is to let the processor hardware find the parallelism at run time. Such processors are known as Superscalars and they operate on the normal sequential source code. The name "Superscalar" is chosen to distinguish these machines from other computers with parallel capabilities, such as Vector Processors, which are designed to manipulate the large data structure arrays commonly found in scientific code. The superscalar processor is capable of fetching several instructions from memory each cycle and these help to form a "window" of available instructions. This "window" is then examined for suitable candidate instructions that can be safely executed in parallel with each

other. This process involves testing for data dependencies between candidate instructions and the use of techniques such as "scoreboarding" or dynamically renaming registers so that instructions can be dispatched out-of-order.

The superscalar processor may also have mechanisms that keep a "history" of recent conditional branch behaviour that allow it to adjust dynamically to changes in program control flow. This flexibility can lead to higher accuracy in making branch predictions than is possible at compile time, although this point is still the subject of debate in recent research papers **[18]**. The main advantage seen by the user of computers based on superscalar processors is the code compatibility across a range of processor implementations, as they can all operate with the same sequential source code. Users are not required to recompile their programs when they update the hardware.

One serious disadvantage of the superscalar machine is the limited size of the "window" from which parallel instruction groups are selected, as there is no possibility of moving code globally. This limited view only allows for localised code compaction and will probably place an upper bound on the achievable parallel performance that falls far short of the potential parallelism available. There is also a problem with the increased circuit complexity required to detect data dependencies at run time and to handle the consequences of out-of-order instruction issue and dynamic register renaming.

1.3 Scheduling Code for a Minimal Superscalar

This project attempts to combine the best features from both of the previous approaches by scheduling code globally at compile time and encoding the parallelism found in sequential code so that it is compatible with a range of processors. Features of the HSP architecture should ensure that good performance is obtained on processor models with limited ability, even when code has been scheduled for a high-powered machine. In the machine model adopted, the number of instructions fetched each cycle is no longer directly linked to the number of instructions executed, but it is influenced by the size of the "Instruction Buffer" into which instructions are initially placed.

The need for dynamic renaming of registers and out-of-order issue is eliminated, thus greatly simplifying much of the circuitry found in a normal superscalar processor, leading to the concept of a "Minimal Superscalar". The HSP instruction set allows all instructions to be defined as conditionally executed on the basis of one or more Boolean registers. Removal of conditionally executed code whilst still in the Instruction Buffer can significantly reduce the number of functional units that are required to execute individual instructions in a given machine. At compile time, an Instruction Scheduler makes global movements on the source program code in order to form "parallel groups" of instructions that can safely be executed in the same time frame. Such groups are then used to fill the conceptual branch and load delay slots

that are assumed to exist on the target family of superscalar processors. Instruction scheduling also aims to promote code as high as possible in the sequential order, subject only to true data dependencies. The Instruction Scheduler will also be able to employ techniques such as software pipelining to optimise the performance of loops in order to increase the overall level of parallelism available.

A new type of branch instruction, with an explicitly defined "branch-dependent" region, is introduced that enables the parallelism exposed by the Instruction Scheduler to be preserved when the code is converted back into a sequential form suitable for the superscalar processors. Although code may be scheduled with a particular processor model in mind, it will still be correctly executed by any processor from the same family, regardless of its particular functional capabilities and latencies. It is hoped that the advanced features of the Hatfield Superscalar Processor will enable good performance to be obtained over a range of different processor models, all running on the same scheduled source code.

2 Simulator Overview

2.1 Elements of the Simulator

The Superscalar Simulator has four distinct stages in the processing of instructions from the Instruction Cache. The Instruction Fetch (IF) stage involves reading a block of instructions from the Instruction Cache and placing them into the top end of the Instruction Buffer. Instructions from the bottom end of the Instruction Buffer are selected for Instruction Decode (ID) and have their source operands accessed. The instructions being issued are passed to appropriate Functional Units which use the source operands when performing the Instruction Execute (EX) stage of processing. Finally, the Functional Units compete for a Result Bus and Write Back (WB) the results to the appropriate destination. Each of the four processing stages can take one or more clock cycles to complete, depending on the latencies for the different classes of instructions and cache memory access times. Individual instructions may also be forced to incur extra "wait" cycles while they are in the Instruction Buffer queue or in a Functional Unit that has unsuccessfully bid for a Result Bus.

All of the Simulator's important parameters can be altered by the user, starting with the Fetch Width that defines the size of block accessed from the Instruction Cache. The length of the Fetch Unit pipeline is determined by the number of Fetch Cycles for an Instruction Cache access. The Instruction Buffer itself has a maximum capacity defined by Buffer Size and functions as a FIFO queue. The HSP architecture has the ability to process several instructions in parallel as determined by the number of "pipes" provided in the model. Each cycle, new instructions are selected from the bottom of the Instruction Buffer to form an Instruction Decode (ID) group of data-independent instructions, each of which is assigned a "pipe". The "pipes" serve as routing devices that send the instructions to the appropriate Functional Units for execution. Potentially, each "pipe" can deliver an instruction to any unit in a common pool of Functional Units, but the allocation of resources is prioritised to select pipes containing instructions from earlier in the program sequence. Once the Functional Units have completed their processing they compete for a Result Bus in order to Write Back the results to the Register Files, and to Forward results to other Functional Units. The Memory Load and Memory Store Functional Units are the only units that interface directly with the Data Cache.

A Block Diagram of the complete Hatfield Superscalar Processor is shown in Figure 1 and the important elements illustrated will be discussed in detail in the following Sections.

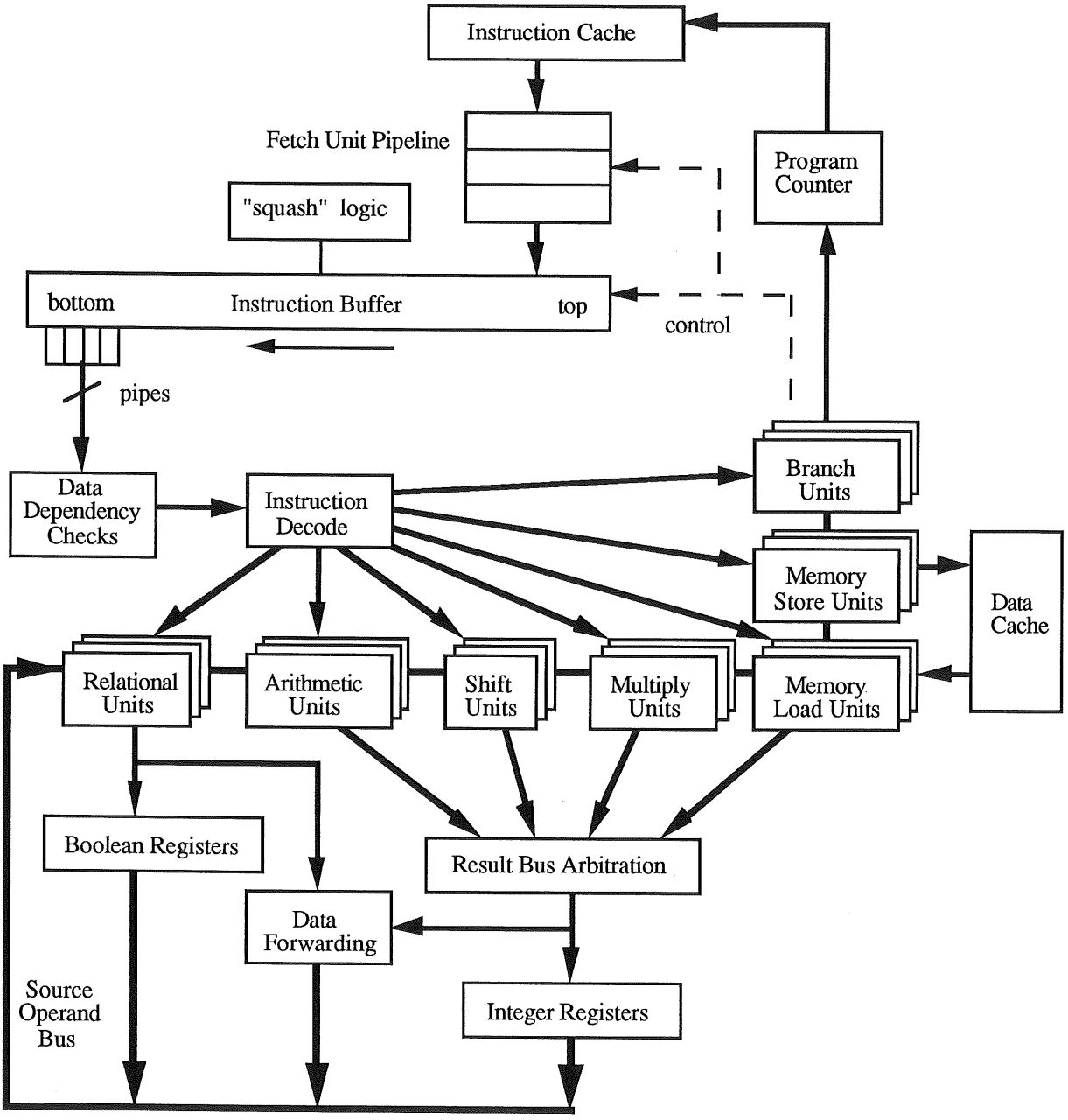

Figure 1 Block Diagram of the Hatfield Superscalar Processor

2.2 The Need for a Parametrised Model

It is of great importance to provide a highly parametrised model of the Superscalar Simulator that allows a wide range of values to be adopted for all of the machine's features. This flexibility allows the potential benefits to be assessed for the most ambitious machine models without being over-constrained by considering implementational problems at this early stage. The machine model can then be "fine-tuned" to remove redundant or little-used hardware features and to investigate possible tradeoffs of performance against the functionality provided. Some of the most important parameters defining the Simulator model are shown in Table 1.

Fetch Capacity and Latency
Instruction Buffer Size
Parallel Issue Capability
Instruction Latencies
Type and Number of Functional Units
Number of Result Busses
Number of Integer and Boolean Registers

Table 1 Main Simulator Parameters

The Superscalar Simulator has been given substantial potential resources in the hope that these can be exploited by the Instruction Scheduler to achieve a high degree of instruction parallelism. Some recent research [11,12] suggests that theoretically the amount of instruction-level parallelism available in non-scientific code is an order of magnitude above that realised to date by proposed architectures. It is the aim of the project to attempt to increase realisable parallelism towards these theoretical upper bounds, hence the powerful Superscalar Simulator model.

2.3 User Facilities

A menu-driven interface is provided to alter any of the machine's individual parameters and to select a particular machine model. The user has the ability to examine the contents of the Instruction and Data Caches as well as the Integer and Boolean Register Files. Facilities are provided to record the "trace" of a program and to "single-step" through a program's execution, with full details of the internal workings of all the functional units being displayed.

Many user-defined parameters are also available that enable or disable aspects of the Simulator behaviour so that different features of the machine can be assessed quantitatively by comparing the results for different program runs. These control signals are also used to specify the amount and type of information displayed or recorded whilst a simulation is being carried out on a source program. Emphasis is placed on statistical data to help to evaluate the usage of the various machine resources provided by a particular Simulator configuration and to obtain a profile of the program's run-time behaviour.

3 The Superscalar Instruction Buffer

3.1 Fetching Instructions from the Instruction Cache

There is a need to simulate the use of a variable number of cycles for accessing instructions from the Instruction Cache. Technological advances mean that the machine cycle-time for new processors is likely to fall faster than equivalent improvements in cache memory access time. A Program Counter register is used to supply the Instruction Cache address. The size of each new group of instructions fetched is a control parameter, as are the number of cycles required. A Fetch Unit Pipeline data structure is used to simulate the variable number of cycles it takes to fetch instructions from the Instruction Cache into the Instruction Buffer. A group of instructions from the bottom of this Fetch Pipeline are copied into the top of the Instruction Buffer only if there is sufficient room for the whole group.

3.2 Modelling Instruction Buffer Behaviour

The Instruction Buffer functions as a FIFO queue where, each cycle, new groups of instructions from the Instruction Cache are read into the top of the Buffer, if there is room for them. Instructions from the bottom of the Instruction Buffer are selected for in-order dispatch to the Functional Units. Successfully dispatched instructions are marked as "squashed" in the Instruction Buffer and contiguous squashed instructions are removed from the bottom of the Buffer at the end of each cycle. Instructions in the Buffer can be also be marked as "squashed" to ensure that they are never dispatched and this use is described in more detail in Section 9. The Instruction Buffer can be fully or partially "flushed" upwards from a given point if a branch is found to be taken. The Fetch Pipeline can also be flushed of unwanted instructions when there is a change in program control flow. Some possible implementations of the Instruction Buffer hardware have been considered [2], with particular attention being paid to the interconnection problems that arise when routing instructions from the Buffer to the Functional Units.

4 Instruction Decode and Parallel Instruction Issue

4.1 Selecting Candidate Instructions for Dispatch

The Simulator only dispatches instructions from the Instruction Buffer in strict program order. The first un-squashed instruction at the bottom of the Buffer receives priority and other un-squashed instructions can only be dispatched at the same time if there are no data dependencies between members of the group that is formed. Once a data dependency is detected, no further instructions from the Instruction Buffer can be dispatched in the current clock cycle.

4.2 Data Dependencies

The maximum number of instructions that can be dispatched concurrently is determined by a user specified parameter. Each new member of a group being formed for dispatch must check that none of the existing members can possibly alter one or more of the candidate instruction's source operands. If a data dependency is found no further instructions are added to the group, as out-of-order issue is not allowed. Members of this "Instruction Decode" group are then checked for source operand availability and then allowed to compete for Functional Units to execute the instructions. If a particular instruction from the Instruction Decode group is unable to obtain its source operands or find a suitable Functional Unit it will be "blocked" from proceeding, as will all instructions that follow it in the original sequential code.

4.3 Conditional Execution

Any instruction can specify a number of Boolean conditions that must be met if the instruction is to be executed. Boolean values used for the conditional execution of instructions are obtained directly from the Boolean Register File, or they can be forwarded from a Functional Unit that is completing a relational or logical calculation. If it is found that the conditions for execution of an instruction in a Functional Unit are not met, the unit is re-initialised and the instruction discarded. Any Functional Unit that is re-initialised in this way will not compete for a Result Bus and will be available to take a new instruction in the next machine cycle.

4.4 Resource Constraints

The number and type of Functional Units provided by the Simulator are specified by the user, with pipelined or non-pipelined versions being available for most types of instructions. The Functional Units form a common pool of resources that are available to any of the instructions in the Instruction Decode group on a first-come first-served basis. If an instruction is to be executed it competes with other instructions for an available Functional Unit of the correct type,

with priority being given to instructions that appear earlier in the program sequence. If an instruction is unable to secure a Functional Unit, it is "blocked" and no instructions that follow it can be dispatched.

4.5 Operand Availability

Once an instruction from the Instruction Decode group has successfully secured a Functional Unit, the instruction is passed to that unit along with the appropriate source operands. If source operand values are unavailable in the Main Register files this indicates that a new value is currently being computed by one of the Functional Units from an instruction that was issued earlier. A check is made to see if the new value is available from a Functional Unit that completed its work last cycle but has still to Write Back the result to the Main Registers. If any of the source operands are still unobtainable, the candidate instruction is "blocked" and no further instructions from its Instruction Decode group are dealt with this cycle. All instructions that are successfully dispatched to Functional Units are marked as "squashed" in the Instruction Buffer.

4.6 Removal of Instructions from the Instruction Buffer

At the end of each cycle the Instruction Buffer is examined from the bottom upwards and all instructions marked as "squashed" are removed, up to the first non-squashed instruction. The remaining instructions in the Buffer, squashed or un-squashed, are then counted and deducted from the maximum Buffer capacity in order to determine if there is sufficient room for the next group to be fetched in from the Instruction Cache. Instructions that are marked as "squashed" which are within the main body of "valid" instructions will not be considered for the next Instruction Decode group. Squashed instructions that are preceded by un-squashed instructions cannot be removed completely from the Buffer at this stage as they convey positional information that is needed to process Branch instructions correctly. The method used to convey the scope of the branch-dependent region relies on a Branch count that is associated with each Branch instruction. The Branch Count gives the position of the last branch-dependent instruction relative to the position of the Branch instruction itself, so it is essential to preserve the correct spatial representation when the instructions are in the Instruction Buffer. Any valid instructions that were unsuccessfully selected for dispatch this cycle will remain in the Buffer and will form part of the next Instruction Decode group.

5 Functional Units

5.1 Types of Functional Units

The HSP instruction set supported by the Superscalar Simulator, (Appendix A), includes most of the iHARP [1] instructions and has been split into different types, partly according to the nature of the destination specified by the instruction and latency considerations. By using this method of categorisation it is hoped to be able to simplify the hardware implementation of the processor by physically grouping Functional Units of a particular type around register files which they target as destinations. This approach should simplify the interconnection complexity that will exist in a processor with high parallel capabilities. There are seven types of Functional Units, as shown in Table 2.

Arithmetic
Relational (Boolean)
Multiply
Shift
Memory Load
Memory Store
Branch Units

Table 2 Functional Unit Types

At present, Memory Load and Memory Store capabilities are independent of each other but the machine model may be altered in the future to combine these into a single Memory Access capability. One possible Data Cache implementation could involve the use of two dual-ported caches, each holding the same data at all times. A Memory Store would then be required to write to both caches simultaneously, to ensure data coherency, and would use one port on each. Such an implementation could allow parallel access for 4 Memory Loads, or 2 Memory Stores, or 1 Memory Store and 2 Memory Loads. Until a clearer picture emerges of the likely demands on the Data Cache made by code that has been fully scheduled, implementation issues will be put to one side and the current orthogonal approach will be maintained.

5.2 A Common Pool of Resources

The maximum number of instructions that can be dispatched for execution in a cycle is specified by the user as the number of "pipes". Unlike the HARP approach [4], these pipes have no particular individual processing capabilities in their own right but can be thought of as "conduits" that route instructions, along with their source operands, to the appropriate type of

Functional Unit. By supplying a common pool of Functional Units that are available to any one of the individual pipes it is possible to economise on the number of Functional Units provided. The exact number and mixture of different types of Functional Unit that comprise the resource pool can be altered to closely match the likely needs of scheduled general purpose code, with a trade-off between the savings in chip complexity against the increased risk of a "blocked" instruction occurring. It is desirable that the number of Functional Units provided should be kept as low as possible to reduce the interconnection problem for the hardware and it is also important that each unit should spend as much time as possible executing useful code.

5.3 Pipelined and Non-Pipelined Implementations

All units are available in pipelined or non-pipelined forms, with individual latencies being specified by the user. Non-pipelined Functional Units can only work on one instruction at a time and are unavailable for new input for a number of cycles depending on the latency associated with each type of instruction. Pipelined units are always available for new input each cycle unless they are "stalled" due to a resource restraint at their output when they compete for Result Busses. One exception to this scheme is when Arithmetic Units are used to execute integer Divide instructions. As Divide instructions occur infrequently in general-purpose code they do not justify their own separate category of Functional Unit. Instead, the Arithmetic Units are forced to adopt the latency associated with the Divide instruction rather than the normal arithmetic latency. In the case of Pipelined Arithmetic units, the processing of a Divide means that the input can be blocked for several cycles, depending on the relative difference in instruction latencies. Branch Units are different to other Functional Units in that they can complete their work during the Instruction Decode (ID) phase and have a fixed latency of one.

5.4 Data Forwarding and Result Bus Arbitration

The last task of a Functional Unit is to Write Back the result to the appropriate Main Register file. If an instruction requires to access this new value as one of its source operands it is advantageous to provide some means of "forwarding" the new value between Functional Units, without having to wait for the Write Back to complete. However, the number of Result Busses for integer values is specified by the user and Functional Units must compete for their use. Only those units that secure a Result Bus can complete during the current cycle and forward their results on to other units. Functional Units that are unsuccessful in obtaining a Result Bus are forced to "stall" but will receive preferential treatment in the next arbitration round. Pipelined Functional Units that have their output stalled in this way will still be available to take a new instruction next cycle, unless all of their pipeline stages are already fully occupied. Sufficient Boolean Result Busses are provided to cope with the maximum possible demand, so there is no need to perform any kind of bus arbitration.

6 Register Files

6.1 A Pipelined View of Registers

In order that the Simulator can achieve the functionality required by the HSP model, it is convenient to model three banks of Integer and Boolean Registers which can be viewed as a "register pipeline". The three stages are represented by the "Next", "Forward" and "Main" register files. Functional Units that are completing their function during the current cycle will write their results to the "Next" register bank. At the end of the cycle all functional units that are still actively computing a result will mark the appropriate Main Registers as unavailable. The Main Register file is updated from the Forward Registers, which in turn take in the new values from the "Next" register file. This process is illustrated in Figure 2 where Functional Units write new results into the Next file but obtain source operands from the Forward or Main files.

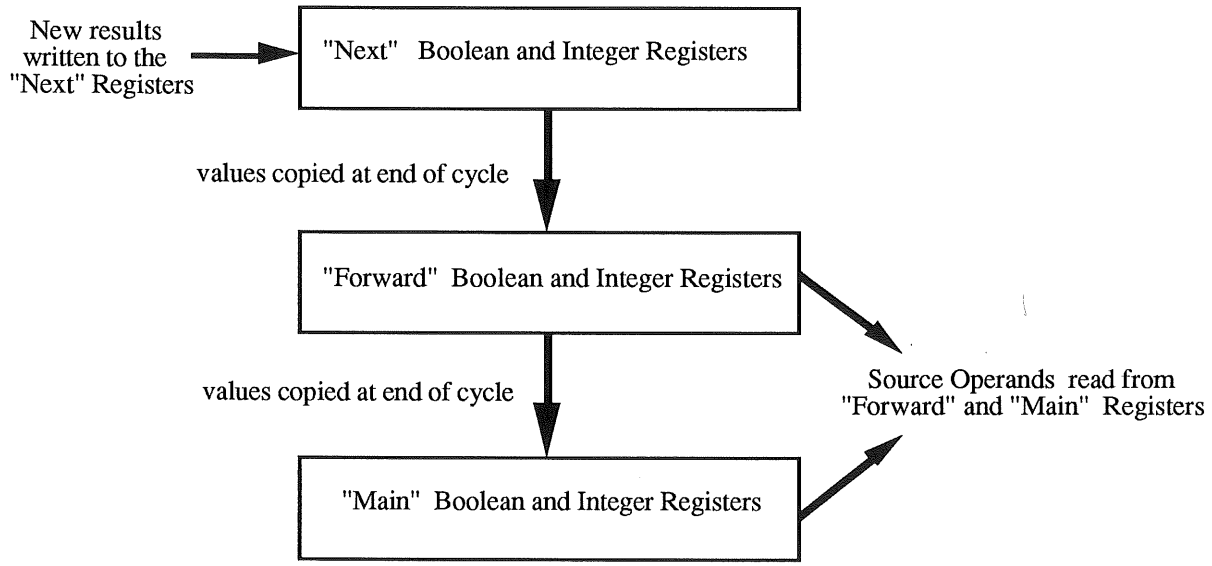

Figure 2 Register Files

6.2 Operand Availability

The source operands specified by an instruction are identified in the ID stage and made ready for the designated Functional Unit to access during the EX stage. It is essential for the processor model that access to the registers is performed prior to an instruction's execution if the machine cycle time is not to be unduly extended. However, in the Superscalar Simulator it is implementationally convenient to allow the Functional Units to perform these register accesses during the EX stage. Functional units requiring source operands will first look in the "Main"

registers and if these are marked as unavailable the "Forward" registers are examined to see if the required value was calculated last cycle. As new results from the Functional Units are always written first into the "Next" file, there is no possibility of undesirable side-effects in the Simulator when requesting source operands. Source operand values are taken from the Main or Forward registers and cannot be influenced by a new result being computed during the same cycle as these always target the Next register files.

6.3 Invalid Data

All the Register Files have a flag which can indicate that the data they contain is "invalid". This feature serves two purposes. Firstly it enables Speculative Execution of instructions to be supported and secondly it acts as a useful debugging tool when running diagnostic programs. The user may choose to mark all the Main registers as "invalid" at the start of a program run, and these will only become "valid" when a functional unit writes in a new value. If the program tries to access a source operand that is "invalid" an error message is given to indicate a possible programming error. Many real benchmark programs have procedures that start by automatically storing away in memory the contents of any registers that they will be using, regardless of their present contents. It is necessary to mark all registers, or possibly a subset identified by the compiler, as "valid" before starting to run a program of this nature or false error messages will occur.

6.4 Speculative Execution

It is possible that an instruction scheduler could move an instruction speculatively above the conditional branch that controls entry to the instruction's original location. Such an instruction may now be executed unconditionally, before it is known which way the conditional branch will go. If the promoted instruction causes an exception, such as arithmetic overflow, the program will be forced to terminate, even if this instruction was originally on the branch path that is not taken. A speculatively promoted Memory Load instruction could produce an invalid memory address that would also cause an exception when the Data Cache is accessed. A special marker is used in the HSP architecture to indicate that the promoted instruction is a Speculative Instruction and this inhibits any exception errors from occurring. The destination register of such an instruction is instead marked as "invalid" so that if a non-speculative instruction tries to access the register as a source operand, a trap is forced. Any instruction can be marked as Speculative except for Memory Store instructions as these permanently alter the machine state. One possible strategy that could support the speculative execution of Memory Store instructions is to introduce a Write Buffer for the new values rather than allowing the memory to be altered straight away. The HSP architecture does not encompass such a scheme at present.

6.5 Write-After-Write Hazards

Instructions may be held up due to anti-dependencies, or write-after-write (WAW) hazards. Anti-dependencies, also known as Write-After-Read (WAR) hazards, occur when the destination register of an instruction is the source register of an earlier instruction and these can be dealt with in the Instruction Scheduler by register re-naming. In WAW hazards an instruction issued earlier is still computing a result for a register that is also the destination for the instruction currently being issued. If the latter instruction is issued and completes before the earlier instruction has completed, there is a danger that the final value stored into the destination register will be incorrect. The Simulator offers two strategies to cope with the problem as it has not yet been decided which scheme should be encompassed in the HSP architecture. The first approach is to insist that instructions are held up until all previously issued instructions with the same destination have been completed. The second approach is to inhibit the Functional Unit that is handling the earlier instruction and to issue the second instruction as normal. Any Functional Unit that is inhibited will not compete for the Result Busses and will become available for new instructions in the next cycle. It is safe to inhibit instructions in this manner as there are no instructions waiting for the redundant result from the earlier instruction due to the fact that instructions are never issued out-of-order. Any Read instruction that is data dependent on the first Write will be forced to hold up until the new result is in the Forward Registers so a second Write issued in parallel with the Read cannot have adverse side-effects. A choice of Simulator behaviour for dealing with WAW hazards can be specified by the user.

6.6 Special Purpose Registers

Some special purpose registers are provided for the convenience of the compiler when it is dealing with "ORed" addressing **[16]** and procedure calls. Two pointers into memory are provided, Global Pointer (GP) and Stack Pointer (SP), that are used as base addresses for global data objects and stack frames respectively. There is a Status Register (SR) which can be used to hold the contents of the Boolean Register file so that this information can be easily stored away in memory on a procedure call. Lastly a second Stack Pointer (SP') is also provided as this can be used to reference an earlier stack frame. The SP' register is included to support code that was originally compiled for the HARP model from Modular 2 source code where scope rules for program variables need to be enforced. All four registers are arbitrarily mapped into Integer Registers reserved for this purpose, as specified in a global header file.

7 Modelling the Caches

7.1 The Instruction Cache

As the Superscalar Simulator is only concerned with modelling the functionality of the Superscalar Processor, there is no restriction on the data structures that can be used to represent the Instruction Cache. This freedom of choice means that a structure can be chosen that helps the Simulator run faster by holding instructions in a decoded form, even though this requires larger amounts of memory storage on the host computer. As a new program file is read into the Simulator, the individual instructions are decoded and the relevant fields of an Instruction Record are filled. The Instruction Cache consists of a contiguous linear array of pointers to instruction records to represent the source program code. Facilities in the Simulator allow the user to examine any part of the Instruction Cache and to display an instruction in the original assembler code or as an abstract instruction record. Individual locations in the Instruction Cache can be edited by the user, possibly to alter program constants between program simulation runs.

7.2 The Data Cache

The Data Cache consists of an array of pointers to data elements, each of which can hold a 32-bit value and an optional label to aid readability for the user. Procedures that access the Data Cache convert any memory addresses into an array index by dividing the address by the number of bytes in a word, which is currently defined by a global constant as four. In order to conserve memory on the host computer, the Data Cache is declared as an array of pointers according to a global constant that specifies the size of the array. Individual Data Elements are allocated storage space as and when they are accessed by the program being executed. This policy means that at the end of a program run the Data Cache will have only created elements for those locations used by the program. When the Data Cache is displayed to the user only elements that were used by the program are visible, which aids readability.

If a program source file specifies initialised data, the appropriate data elements are created and loaded with data. The Data Cache is byte-addressable and this is implemented by reading and writing complete words but only altering the specified byte within the word. Half-word accesses to memory are not supported by the HSP architecture. Functional Units that perform Memory Loads and Memory Stores have their own particular instruction latencies, which can be different from each other and from the latency associated with accesses to the Instruction Cache.

The Simulator keeps track of new data values on their way to Data Cache locations by modelling a Memory Store Pipeline. Any Memory Load instructions first check if the latest value for the required memory address is in the Memory Store Pipeline before looking in the Data Cache itself. This process ensures that the correct value is obtained and means that the machine does not have to wait for the Memory Store to complete before the new value can be accessed.

8 Branch Units

8.1 The Branch Delay Count

A branch instruction appearing in sequential code must convey to the Superscalar processor some sense of the amount of code that has been scheduled into its "branch delay" slot. A Branch Count can be associated with every branch instruction to indicate the scope of this "branch-dependent region". When the Simulator executes a branch instruction it first determines if the branch is to be taken. A taken branch with a Branch Count of zero causes all instructions following that branch to be flushed from the Instruction Buffer and from the Fetch Pipeline, as well as inhibiting the processing of later instructions from the same Instruction Dispatch group as the branch itself. The contents of the Program Counter are then altered to the new value specified as the branch target address. New instructions will be fetched from the branch target address during the next cycle.

If the value of Branch Count is greater than zero, checks must be made to ensure that the correct number of following instructions are allowed to be processed before dealing with any new instructions from the branch target stream. The Branch Count therefore specifies a "protected region" of instructions that follow the branch which must always be considered for execution, irrespective of how the branch instruction evaluates, and is therefore analogous to a branch delay slot in a conventional pipelined machine.

8.2 Flushing Instructions from the Buffer

When a branch is taken there may be a need to flush out instructions from the Instruction Buffer that have been pre-fetched from the sequential successor stream and that are no longer required. The Simulator removes these unwanted instructions from a point in the Instruction Buffer that depends on the position of the taken branch instruction and the value of its Branch Count. The newly freed Buffer locations increase the overall space that is available in the Instruction Buffer for it to accept a new group of instructions from the Fetch Unit towards the end of the same cycle.

8.3 Modelling the Fetch Unit Pipeline

Latencies associated with the fetching of instructions from the Instruction Cache are implemented by a pipeline of Fetch Registers. The user specifies how many instructions are to be fetched each cycle and this determines the width of the Fetch Unit. The length of the Fetch Pipeline is determined by the Fetch Latency. The top row of the Fetch Unit is filled with a new group of instructions from the Instruction Cache, from an address specified by the Program Counter. The Program Counter value is then adjusted to reflect the number of instructions

successfully read from memory during the current cycle. Instruction groups are then copied into each row of the Fetch Unit from the row above and instructions from the bottom of the Fetch Unit enter the Instruction Buffer, provided there is enough space for them.

If there is insufficient space in the Instruction Buffer to take a complete group of instructions from the Fetch Unit, no new instructions are put in the Instruction Buffer and the group remains in the bottom row of the Fetch Unit. If other stages of the Fetch pipeline are also occupied, a failure to load the Instruction Buffer will result in the Fetch Unit "stalling" and no new instructions will be fetched from the Instruction Cache during this cycle. Instruction positions in the Fetch Pipeline can be marked as "valid" and "squashed". When an instruction first enters the Fetch Unit from the Instruction Cache it is marked as "valid" to indicate that the position is occupied by a real instruction. When a branch is found to be taken, one or more whole rows in the Fetch Unit may be "flushed" by marking the relevant positions as "invalid". If some of the contents of a Fetch Unit row are to be preserved after a branch, due to a non-zero branch count, the whole row cannot be re-initialised so the redundant instructions are marked as "squashed" instead. Instructions that appear in the bottom of the Fetch Unit which are marked as "squashed" will not be considered when transferring a group of instructions to the Instruction Buffer. In order that the Simulator can model a Fetch latency of just one cycle, the length of the Fetch Pipeline is always one more than the Fetch latency. This means that normally the bottom row of the Fetch Unit Pipeline contains instructions that have also just been entered into the Instruction Buffer.

8.4 Processing Branch Instructions

Branch instructions differ from other types of instruction in that they perform the bulk of their work in the Instruction Decode stage rather than the Execution Stage. When a Branch instruction is considered for dispatch, an available Branch Unit is found and processing carried out during the same ID cycle. If the Branch evaluates as not taken, no further action is required so the Branch Unit is cleared and returned to the common pool of Functional Units to become available again in the next cycle.

If a branch is found to be taken there are several different cases that can arise which will require the HSP architecture to behave in different ways. When the Branch Count is zero, no instructions following the branch are to be executed and the Instruction Buffer can be flushed from the branch location upwards. Similarly, the Fetch Unit is completely flushed and all subsequent instructions from the Instruction Decode group are inhibited. The PC is then altered to the branch target address and the branch unit has finished its task.

When the Branch Count contains a non-zero value, the last branch-dependent instruction may be in the Instruction Buffer, or it may be on its way through the Fetch Unit pipeline, or it may not have been fetched from the Instruction Cache. A search is first made in the Instruction Buffer to

see if it contains the last branch-dependent instruction. In this case the dependency is said to have been "satisfied" and it is safe to flush the Instruction Buffer from beyond the last dependent instruction and to completely clear out the Fetch Unit. Again, instructions that follow the branch in the same Instruction Decode group must be inhibited if they come from beyond the last branch-dependent location in the Instruction Buffer. The new value for the Program Counter then takes effect and the Branch Unit finishes processing at the end of the cycle.

The second case to consider is when the last branch-dependent instruction cannot be found in the Instruction Buffer itself but is already somewhere in the Fetch Unit pipeline, on its way from the Instruction Cache. Instructions in the Fetch Unit are preserved up to the last branch-dependent instruction, with following locations being marked as "squashed" or "invalid". No instructions in the Instruction Buffer are flushed and all other members of the Instruction Decode group are allowed to continue as normal. The new value for the Program Counter then takes effect and the Branch Unit finishes processing at the end of the cycle.

The last case to consider when processing a branch instruction that is found to be taken is when the last branch-dependent instruction has not yet been fetched from the Instruction Cache. A count is made of the number of occupied positions in the Instruction Buffer that follow the branch instruction and this is added to the number of valid un-squashed instructions in the Fetch Unit pipeline. This combined total is deducted from the Branch Count specified in the branch instruction to yield a figure that represents the "short-fall" of dependent instructions still required to be fetched. The value of the "short-fall" is stored in the Branch Unit which will then continue to process the branch instruction in subsequent cycles. There may be several cycles before the last branch-dependent instruction is detected in the Fetch Unit and during each cycle the short-fall value is recalculated to take account of the actual number of valid instructions in the Instruction Buffer and the Fetch Unit Pipeline. Only then can the new value for the Program Counter take effect and the Branch Unit finish its processing.

8.5 Handling Multiple Branches

It is possible that branch instructions may be moved by an Instruction Scheduler so that they appear within the branch-dependent range of an earlier branch instruction. When an Instruction Decode group is being processed for dispatch to the Functional Units, there may be several branch instructions in the same group. As long as there are sufficient Branch Units, all the branch instructions can be dispatched, subject to the normal data dependency checks. Any number of branches that evaluate as not-taken can be processed in parallel, up to and beyond the first branch that is found to be taken.

Once a taken branch is found, its Branch Count may inhibit other instructions from the same Instruction Dispatch group from being executed. If a second surviving branch instruction from the same Instruction Decode group also evaluates as being taken, it is not allowed to take effect

during the current cycle even if its Branch Count can be satisfied. Only one branch instruction that evaluates as being taken can be allowed to alter the Program Counter in any one cycle. The second branch taken will be dealt with in subsequent cycles when its Branch Count will be re-appraised to determine if its branch-dependent region has been satisfied. This decision will depend on the modified state of the Instruction Buffer and Fetch Unit occupancy caused by the processing of the first branch's Branch Count. Processing of the first branch could cause instructions to be flushed from the Instruction Buffer that were within the scope of the second branch's dependent region.

It is also worth noting that this strategy implies that a branch instruction within the branch-dependent region of an earlier branch instruction, must itself have a branch-dependent region that extends at least as far as that of the first branch instruction. This relationship between "nested" branches will always be the case if the code has been scheduled correctly.

9 Processing in the Instruction Buffer

9.1 "Squashing" Instructions in the Instruction Buffer

In the normal course of events valid instructions enter the top of the Instruction Buffer from the Fetch Unit and move down to the bottom end of the Buffer from where they are selected as members of the next Instruction Decode group. Once an instruction has been successfully dispatched to a Functional Unit it is marked as "squashed" in the Instruction Buffer. Towards the end of the cycle, all contiguous squashed locations from the bottom end of the Instruction Buffer are removed, freeing up more space for incoming instructions from the Fetch Unit. When a branch is found to be taken, some or all of the locations in the Instruction Buffer may be marked "empty" as the unwanted instructions are flushed out.

Instructions in the Instruction Buffer may be also be marked as "squashed" in order to prevent them from ever being dispatched to Functional Units. While the latest Instruction Decode group is being formed and processed, all conditionally executed un-squashed instructions that are resident in the Instruction Buffer are examined to see if any of them can be safely eliminated. The bit-masks that encode the conditions to be met for the execution of each instruction are compared with the latest Boolean values, with new result values being forwarded from Relational Units that completed their calculation during the previous cycle. If one or more of the conditions necessary for execution is definitely not met the instruction in the Buffer can be safely "squashed" at the end of the cycle. As only un-squashed instructions are considered for inclusion in the Instruction Decode groups, this means of squashing instructions in the Instruction Buffer effectively alters the code that is dispatched to the Functional Units in the following cycle. One restriction to the squashing of conditionally executed code in the Instruction Buffer is that a test must be made to ensure that there is no un-squashed relational instruction ahead of the candidate to be squashed which could potentially alter the value of the Boolean upon which the decision to squash uniquely depends. The ability to squash instructions before they are considered for dispatch is put to good use in Section 9.3.

9.2 Scheduling Code for Delay Slots

One of the biggest problems in enhancing the performance of pipelined machines is to eliminate the branch penalty that is suffered whenever a branch is taken. This penalty is due to the need to flush out unwanted instructions from the sequential successor stream, leaving the processor nothing to do for one or more cycles while it waits for new instructions to arrive from the branch target stream. By using a delayed branch mechanism the processor is instructed to always execute instructions scheduled in the branch delay slots, irrespective of whether the branch is taken or not. This approach ensures that the processor is kept busy with useful instructions while new instructions are being fetched from the branch target address, but it relies on the Instruction Scheduler being able to fill the delay slots successfully with useful code.

Suitable candidate instructions can usually be found to fill the first branch delay slot but subsequent slots become progressively harder to fill [21]. The problem is magnified in a parallel machine where the number of instructions required to keep the machine busy is greatly increased. Traditional machines may be forced to suffer the branch penalty in terms of lost machine cycles each time the branch is take. The HARP project adopted a strategy of promoting conditionally executed code into the delay slots according to some branch-prediction algorithm. In such a scheme a penalty is suffered every time a branch is mis-predicted. A result of this observation is that much work is being carried out by other researchers on developing algorithms and hardware mechanism that attempt to achieve high branch prediction accuracy in high-performance superscalar machines [6,9,10,17].

9.3 Promoting Conditionally Executed Code

It is thought that the ability of the HSP architecture to dynamically alter the instruction sequence seen by the processing elements of the machine will be of crucial importance to the Superscalar's overall performance. If code is promoted into the conceptual branch delay slots from the branch target stream, rather than the sequential successor stream, it can be made to be conditionally executed on the same Boolean value that is controlling the branch instruction itself. Assuming that the branch control Boolean is calculated by a relational instruction immediately preceding the branch instruction, the new result will come through too late to effect the make-up of the Instruction Decode group containing the branch.

If the branch is not taken, the new Boolean value can be used to squash all of the Instruction Buffer code that was promoted into the branch-dependent region. In the following cycle the new Instruction Decode group can be selected from the sequential successor code that follows the branch-dependent region, if this has been pre-fetched into the Instruction Buffer. In general the fetch-rate exceeds the average dispatch-rate and the sequential successor code will have been pre-fetched into the Instruction Buffer by the time the branch instruction is being executed.

If the branch is taken, the promoted code from the branch target stream survives and provides the processor with useful instructions to be executed while the rest of the Instruction Buffer is flushed out and new instructions are being fetched from the branch target address. This mechanism should ensure that there is no branch penalty when the branch is taken, without slowing up the processing of the sequential successor code when the branch is not taken. It could also eliminate the need for complex branch prediction strategies for filling branch delay slots by adopting a simple scheme where code from the branch target is always selected for promotion. One exception to this rule may be procedure calls to library routines which are not in the scope of the Instruction Scheduler at compile time, thus making code promotion from the branch target impossible. Also, branch instructions that alter control flow at the end of a procedure call may have unfilled branch delay slots as the return address is determined dynamically at run time.

There still remains a need to use some scheduling algorithm for promoting code that is to be conditionally executed in parallel with the branch instruction itself when the new value of the controlling Boolean may not yet be known.

9.4 Code Performance over a Range of Processors

An important advantage that superscalar processors have over equivalent VLIW machines is their ability to be code-compatible with each other. The source program code need only be compiled once to be acceptable to a whole range of different superscalar processors, each processor having different capabilities and instruction latencies. A consequence of this code compatibility is that a customer using one type of superscalar processor can up-grade the hardware to a more powerful machine without the necessity to recompile all the programs that are being run on the current computer.

However, code that has been scheduled with a particular superscalar implementation in mind may not run very efficiently on a processor with different capabilities. This drop in performance could be particularly noticeable when code scheduled for a machine with, say eight pipelines, is run on a processor with only four pipes. There is a danger that by promoting sufficient code into the branch-dependent region such that eight pipes can be kept busy, the four-pipe model is forced to waste several cycles ploughing through conditional code that is destined to be discarded whenever the branch is not taken.

The ability to squash conditionally executed code that is resident in the Instruction Buffer enables a Superscalar processor with a limited number of pipes and Functional Units to still achieve a high performance when dealing with code scheduled for a more powerful machine. The overall effect is to achieve a graceful degradation in performance as the processor implementation model is reduced from that assumed when scheduling the code. It may even be found that by assuming unlimited hardware resources when first scheduling the code, good performance is still achieved on all but the most restrictive of processor models.

10 Assembler and Loader/Linker

10.1 Loading the Instruction Cache

The Simulator provides facilities that enable assembler source files to be selected by the user and read into the Instruction Cache so that they can be run on the machine model specified by the user. The original C code has first been passed through a "gnucc" compiler that produced the correct format of assembler mnemonic code, together with assembler directives and data allocation commands. The assembler code is relocatable, with branch targets and data references being expressed as labels rather than actual addresses in memory. All instructions are decoded to create Instruction Records which are placed in the Instruction Cache.

10.2 Initialised and Uninitialised Data

Certain assembler directives specify a data label and a requirement to reserve a number of bytes in the Data Cache as uninitialised data space. As each such directive is encountered an appropriate amount of memory is allocated, starting at a specified base address, and an entry made in a Data Cache Symbol Table for the data label and its memory address. Initialised data is dealt with in a similar fashion except that actual data is read into the memory locations that have been allocated in the Data Cache.

10.3 Jump Lists

The compiler may produce structures known as Jump Lists when it compiles "switch" statements in the C source program. These Jump Lists take the form of a "label" followed by a list of labels. Each Jump List must eventually be mapped into a region of data memory. The first label is entered in a symbol table and allocated a Data Cache address. The list of labels refer to instructions in the Instruction Cache and must be translated into the appropriate Instruction Cache addresses before being stored as data values in the Data Cache. These Jump Lists are encountered whilst reading in the assembler source file and contain labels that make forward references to code that has not yet been read in. It is not possible to fully translate all the Jump List labels whilst still reading in the source program so temporary data structures are used to hold the information until all the symbol tables have been created.

10.4 Symbol Tables

Symbol tables provide a means of recording the actual addresses allocated to all labelled instructions and data structures. These tables are then used to "bind" the code that has been read into the Instruction and Data Caches by replacing references to labels with real address values.

After the Instruction Cache has been loaded, it is scanned through by the Simulator and an Instruction Symbol Table is created. The Instruction Symbol Table can then be used to process the abstract Jump List structures, which in turn enter the correct address values in the corresponding jump tables formed in the Data Cache. The temporary Jump List structures are then no longer required and are deallocated.

The Instruction Symbol Table is combined with the Data Symbol Table to form a Working Symbol Table. The Simulator also contains a Library Symbol Table that is designed to deal with standard procedure calls made by a source program for which the code has not been supplied. Typically these will be calls requiring Input/Output such as "printf" or "scanf" and will be handled in the Simulator by making system calls to the host computer on which the Simulator is being run. The Library Symbol Table contains reserved Data Cache addresses, a sort of auto-vector address, which can be picked up by the Simulator at run time in order to implement the relevant procedures. Any instruction that attempts to perform a memory access to such a reserved Data Cache address will cause the Simulator to make the appropriate system call. The compiler must have details of the actual method used by the Simulator for passing parameters to such procedures in order that the code to access the correct register values is produced.

Lastly, the Library Symbol Table is appended to the Working Symbol Table which is then used to bind all the data and instruction labels referred to by instructions contained in the Instruction Cache.

11 User Interface

11.1 Defining the Machine Model

11.1.1 Specifying Latencies

The Simulator offers the user the opportunity to define the parameters that govern many details of the machine model's behaviour. Before starting a program run, the user can display the current parameter settings that are in force and then edit these via the menu options as required. The individual latencies associated with the different types of Functional Units can all be altered, normally in the range one to four, except for Branch Units which always have an associated latency of one. Also Multiply and Divide instructions have extended latency ranges of one to six and one to 32 respectively. Similarly, the number of cycles it takes to access instructions from the Instruction Cache is determined by the Fetch latency, which can be set to a value between one and four. If the Fetch latency is set to one it represents a machine with one branch delay slot while a setting of four would imply a branch delay of four cycles.

A Memory Load latency of one can be selected to model a machine where the "ORed" mechanism is used for Data Cache address calculation. In such a model the value accessed from memory in one cycle is made available via forwarding in the next cycle and there is no Load delay slot. The impact of performing a conventional address calculation by adding two component parts can be modelled by increasing the latency for Memory Load by one cycle and this introduces a Load delay of one cycle.

It will be the job of the Instruction Scheduler to find ways of filling these branch and load delay slots with useful code in order to prevent the branch penalty from seriously limiting machine performance. The Instruction Scheduler will assume a machine model with the longest possible branch and load delays, as well as the maximum envisaged amount of machine resources. The flexibility provided in defining latencies also enables machines to be modelled where processor cycle time is much less than cache access time and allows for non-integer ratios between these two quantities, such as a fetch latency of three cycles with an arithmetic latency of two cycles.

11.1.2 Setting Buffer Parameters

The Instruction Buffer plays an important part in the Superscalar machine and may prove essential in achieving good performance from highly scheduled code, particularly for machines with several branch delay slots. The user can alter several parameters that will define the model to be used for the Instruction Buffer. The maximum size of the Buffer can be set to any value in the range one to 256. The number of new instructions fetched each cycle from the Instruction Cache is determined by Fetch Width, which can be in the range one to 32. The Fetch Width selected must not be larger than the Maximum Buffer Size otherwise no instructions will ever be

successfully fetched into the Instruction Buffer.

A flag signal can be set by the user to disable conditionally executed instructions from being squashed in the Instruction Buffer, so that comparisons can be made between program runs that, hopefully, will show the power of this important machine feature. Another signal can be set to hold up the issuing of instructions, if a Write-After-Write dependency is detected, until the earlier instruction with the same destination has finished being processed.

11.1.3 Altering Branch Behaviour

The Simulator model provides for three alternative implementations of Branch instructions and their action is closely coupled with the models used for the Instruction Buffer and the Fetch Pipeline. The most powerful model adopted is that described in earlier sections where a branch instruction is always issued to a Branch Unit, if one is available. The branch instruction remains in the Branch Unit until its last branch-dependent instruction is detected in the Instruction Buffer or the Fetch Pipeline, at which time the Program Counter is altered and unwanted instructions are removed.

A second model for Branch behaviour is possible where a taken Branch instruction will be blocked in the Instruction Buffer until the last branch-dependent instruction is in the Instruction Buffer or on its way via the Fetch Pipeline. If a Branch instruction is held up in the Instruction Buffer, no instructions following the Branch will be issued as the HSP architecture only dispatches instructions on an in-order basis. This restriction means that the processor could "lock-up" if the Branch Count value is greater than the combined capacities of the Instruction Buffer and Fetch Unit Pipeline.

Lastly, the most restrictive model for Branches can be chosen where all taken Branch instructions are required to hold up in the Instruction Buffer until the last branch-dependent instruction has been fetched into the Instruction Buffer. These alternative models for Branch instruction implementations are provided to enable an assessment to be made of the benefits to be gained by using the most powerful model where there is minimal hold-up in dispatching instructions.

11.1.4 Functional Capability

As discussed in earlier sections, the Simulator has a number of "pipes" that are used as conduits to route instructions in parallel to the appropriate Functional Units. A user can set the number of pipes available to the machine, up to a maximum of 32. The common pool of Functional Units is then accessed by instructions from the pipes on a first-come first-served basis to ensure in-order dispatch of instructions. The number and type of Functional Units comprising the

common pool can be individually specified by the user via menu options, or a pre-defined set of parameters can be selected for a frequently used model. An ability to vary the mix of Functional Units, and to select pipelined or non-pipelined versions for most types of Functional Units, allows a number of interesting Simulator models to be tested and results compared for different runs of a particular program.

11.2 Tracing Facilities

The Simulator offers several methods for following the progress of a program as it is executed on the machine model. When the menu option to run a program is selected, a number of prompts are given to the user that will determine what information will be shown on the screen during the run. If the "trace" option is activated the user can specify the Instruction Cache address from which the trace is to start, or give the number of clock cycles that are to be executed before tracing begins.

Once the tracing commences the contents of the "pipes" are shown each cycle, along with the contents of the Instruction Buffer and the Fetch Unit. In addition, various procedures that are activated in the Simulator program will give the user information relating to the issuing of instructions, data dependency tests and results of the latest computations taking place in the Functional Units. A parameter can be set by the user that calls for a "full report" when in trace mode and this gives additional information for each machine cycle. In particular, the current state of all the Functional Units can be seen showing all instructions in their different pipeline stages together with the new result value and the final destination register or memory address. The full report facility was extremely useful in debugging the Simulator program itself and will be a valuable tool in tracing a scheduled program that is not performing as expected, possibly due to a scheduling error.

As well as explicitly requesting the trace mode, the user can ask the Simulator to stop after an error is encountered and then enter the trace mode for the remainder of the program run. This facility is useful when debugging code that runs successfully for a large number of cycles before an error occurs.

An option exists that will record the results of a trace in a specified file so that a hard-copy listing can be produced to help analyse the Simulator run. This recorded trace contains a cycle-by-cycle account of which instructions were successfully issued to Functional Units, together with a snapshot of the current contents of the Main Integer and Boolean Register files.

11.3 Statistical Data

During a program run on the Simulator a large amount of statistical data is being recorded to give a profile of the program and the way in which the machine's resources have been utilised. The user can display or print out this information via the menu options. The analysis includes a breakdown of the instructions dynamically executed according to the instruction type. Branch instructions are categorised into conditional or unconditional branches, with figures showing the percentage of conditional branches that were actually taken.

The total number of machine cycles used is displayed as well as a count of the total number of instructions that completed execution. The number of instructions dispatched to Functional Units is recorded separately and may not tally with the total for those that complete execution. This discrepancy is due to instructions failing to meet the conditions required for execution whilst in a Functional Unit or being squashed because of WAW dependencies.

The abstract Instruction Cache used by the Simulator serves as a useful data structure for recording the dynamic execution of each of the instructions in the program by simply adding a counter field that is incremented by the Functional Units when they complete execution. The modified contents of the Instruction Cache can then be displayed or printed out to give a profile of the program's dynamic behaviour.

The usage of each pipe is shown as a percentage of all instructions executed so that the user can see the likely effect of increasing or decreasing the number of pipes in a subsequent program run. Similarly, figures are given for each of the Functional Units to show the percentage of instructions of the specified type that were executed in the individual units provided. In addition, various counters are used to record the number of times that instructions of different types were unable to be issued due to a resource being unavailable. These figures give an indication of the likely loss in performance due to an over-restrained machine model being specified.

Careful analysis of the statistical data can result in machine parameters being altered to obtain maximum utilisation from a minimal set of simulated hardware functions when running the benchmark programs, without undue loss in performance. It is hoped that this "fine-tuning" of the Simulator model will result in a machine model being developed that is far simpler than the maximum allowed for in the range of Simulator parameters. Such a Minimal Superscalar could then be studied in more detail with regard to circuit complexity and signal timing arising from actual hardware implementations [2].

The Simulator program makes use of the concept of object orientation by separating out some of the data structures into separate modules, along with all the procedures required to manipulate them. The main Simulator program has only an abstract view of structures such as the Instruction Buffer so that the actual implementation can be changed in the future, to take account

of circuit implementational considerations, without undue knock-on effects throughout the rest of the program.

11.4 User Control Switches

The Simulator provides control signals that can be set by the user before a program is run and their status can be seen on the display option that shows all the program parameters. These controls include specifying a maximum permissible cycle count, in case a program goes into an endless loop, as well as an interval at which the user is given a brief status line of information showing which instruction is currently being executed. If a small clock interval is chosen it may result in many status lines being sent to the screen which will cause the display to scroll rapidly. Activating the "status report pause" control signal forces the program to pause when the screen display is full, giving the user time to examine the status lines before continuing with the simulation. The "WAW hold", "record trace" and "squash enable" signals discussed earlier can also be activated by the user, each one of which may radically alter the Simulator's behaviour. Branch instructions can be forced to hold up until all branch-dependent instructions are fetched into the Instruction Buffer by activating the "hold branches" signal.

12 Results to Date

All the Stanford Integer Benchmark programs have been successfully run on the simulator, in their original compiled form and after some limited hand-scheduling. The results obtained are most encouraging, with speed-ups in excess of 2.5 being obtained for one program even with this very rudimentary scheduling. Unfortunately, many of the advanced features offered by the HSP architecture cannot be used effectively until a suitable Instruction Scheduler has been developed that implements the full scheduling algorithm, but it is hoped that speed-ups obtained so far can be substantially improved.

The user interface with the Simulator works well and the various methods for displaying and analysing results from program runs help to profile the important characteristic of program behaviour. This information should prove useful in the developing and testing of scheduling techniques that are to be included in an overall general scheduling algorithm.

12.1 Simulator Performance on Different Host Computers

The Simulator program itself was developed in the integrated environment provided by the Borland C++ application software, running on a 386 PC. Two versions of the Simulator were written, one taking into account the limited memory available on the PC while the other was designed to be run on a Unix system. This approach worked well, with all of the program development and limited runs of benchmarks taking place on the PC, and the resulting program being easily ported onto the main-frames at the University.

The "Sol" computer at the University offered a performance slightly worse than that achieved on the 33MHz 386 PC, but the "Bacon" computer, running with the latest "sparc" processor, achieved a factor of six speed-up over the PC. These figures assume that the Simulator is the only major process on the host computer. Time-sharing with other users of the system can drastically increase the total execution time for a simulation run. Typical run times for the Stanford Integer benchmark programs are in the range of 5 to 20 minutes and are heavily dependent on the particular parameters chosen to specify the Superscalar Simulator model.

12.2 The One-Pipe Model and Measuring "Speed-ups"

Many of the published research papers dealing with instruction level parallelism express their results in terms of the "speed-up" obtained over a conventional RISC machine but the base model used for such comparisons is not always clearly defined **[12,18]**. If performance is being assessed in this way it is important that the base model used is a realistic one in terms of the effort being made to fill branch and load delay slots with useful code. Any modern optimising compiler can be assumed to try to fill such delay slots with unconditional code, but

when this is exhausted what should be done about the remaining empty slots? Some form of branch prediction algorithm could be used to promote conditionally executed code into these regions, but this may or may not actually help the program at run time.

For this reason it is intended to express speed-up ratios against two different base models. Comparisons will be made with code running on a base machine capable of fetching and issuing no more than one instruction per cycle. The first base model used is one where there are no branch or load delays at all and so represents a machine with a perfect branch prediction mechanism, although instructions may still be held up if they are data dependent on Divide and Multiple instructions which have long latencies. The second model is one where an optimising compiler fills delay slots with unconditional code from within the same basic block, or from branch target or sequential successor code. Conditional code is then used to fill empty delay slots by always promoting code from the branch target, based on the assumption that conditional branches are usually taken. The second base model must take a fixed view as to the number of branch and load delay slots, so there could be a whole family of Base 2 models to cover all possible machine implementations.

This first model takes a harsh view in terms of speed-up ratios but provides an undisputable base line against which to take measurements. This feature may become important if outstanding results are achieved and there is a need to justify the figures claimed. It also acts as a measure of how well the optimising compiler did its job in creating the second base model. Figures for a machine with no branch or load delays can be calculated from the statistical data provided by the normal Simulator model running with the appropriate parameter settings. However, an additional feature has been added to the Simulator that allows it to behave as if there are no delays when running a one-pipe model, so that the base model cycle count is obtained directly. This feature is activated via a user-set control signal on the Simulator.

12.3 The Stanford Benchmark Programs

The Stanford Integer Benchmark programs are a suite of eight programs that require very little initialised data and are thought to exhibit similar behaviour to general-purpose computer programs, although some are quite recursive in nature. The programs exist in two forms, one being the original length program and the other being a "cut-down" version that completes execution in a much shorter time. The long versions of the programs typically involve the execution of 100,000 - 300,000 instructions. This suite of programs have proved very useful in the development and testing of the Superscalar Simulator and will also be used for the development of the Instruction Scheduler program that will implement a powerful code scheduling algorithm. Some preliminary results are shown in Table 3 for the eight benchmark programs run on a machine model that has one branch-delay slot and no load-delay. "Base1" is a single-pipe machine with no branch or load delays. The "Parallel" results are obtained by running the compiled code on a Superscalar model with unlimited resources, so as not to

artificially limit the potential size of the parallel groups. Table 3 shows the instruction counts for each of the benchmark programs followed by the machine cycle counts for the one-pipe "Base1" model with no branch penalty and for the "Parallel" simulator model.

		Machine Cycle Count	
		Base 1	Parallel
Program	Instr. Count	(no delays)	(with delays)
bubble	246,002	249,406	190,139
matrix	258,611	423,915	380,578
perm	338,293	338,297	175,949
puzzle	35,395	35,399	34,369
queens	220,327	220,331	167,652
sort	75,333	83,837	76,148
tower	272,132	252,136	152,749
treesort	144,905	153,409	124,239

Table 3 Benchmark Program Cycle Counts

The results shown in Table 3 can be converted into "speed-up" ratios as illustrated in Table 4. Bracketed figures in Table 4 are theoretical estimates of the speed-ups possible if there are no wasted machine cycles due to unfilled branch or load delays, although there may still be lost cycles due to data dependencies. In other words, complete concurrent groups of useful instructions are assumed to have been promoted into all delay slots.

Program	Speed-up over Base1 Parallel	Scheduled
bubble	1.31*	(1.57)
matrix	1.11*	(1.18)
perm	1.92	(2.56)
puzzle	1.03	(1.23)
queens	1.31	(1.53)
sort	1.10*	(1.27)
tower	1.65	(2.03)
treesort	1.23*	(1.53)
arithmetic means	1.33	(1.61)
geometric means	1.28	(1.52)

Table 4 Speed-ups over Base Models

* Note that some of these speed-up figures are low due to the long latencies of MULT and DIV instructions which force data-dependent instructions to hold up. At present, no attempts have been made in the compiler to schedule code into these "data-dependent delay" slots which could reduce the cycle count for Base1 and Base2 models. Scheduling code into these slots will also reduce the cycle count for the Parallel model but not necessarily to the same degree, as much of the code in these slots may be dispatched in parallel, still leaving the processor with nothing to do at times.

As was mentioned earlier, only the "perm" program has been hand-coded to produce the Base2 and Scheduled versions of the source code. It was decided to "in-line" all the non-recursive procedure calls in the original compiled source code for the "perm" program before moving on to apply the scheduling rules for generating the Base1, Base2 and Scheduled versions. "Base2" represents an optimising compiler version that is hand-scheduled to fill the branch-delay slots with unconditionally executed code wherever possible before promoting conditional code from the branch target stream. The "Scheduled" versions of the programs are hand-scheduled to fill the branch-delay slots with parallel groups of instructions. The figures presented in Table 5 show the cycle counts and the speed-up values for code running on the "full-spec" Simulator, as measured against the two Base models. All four versions take advantage of procedure in-lining.

Machine Cycle Count			
Base1	Base2	Parallel	Scheduled
297,967	301,588	133,413	115,450

Base1 speed-up		Base2 speed-up	
Parallel	Scheduled	Parallel	Scheduled
2.23	2.58	2.26	2.61

Table 5 Measured Performance Values for the "perm" Program

Table 5 shows that a speed-up of more than 2.6 is achieved for the Scheduled code when measured against the optimised compiler model, Base2. This encouraging result is obtained merely by filling the branch delay slots with concurrent groups of useful code, without applying any other elements of the overall scheduling algorithm. Comparing the "perm" cycle count of 115,450 for the Scheduled code from Table 5 with the original unoptimised Base1 cycle count of 338,297 from Table 3 gives the over-optimistic speed-up value of 2.93 and illustrates the importance of using realistic base models. Results for the other seven benchmark programs will be given in a forthcoming report **[19]** that details more precisely the scheduling techniques applied and the issues that they raise regarding performance enhancement.

13 Future Developments

13.1 Extending the Model - Floating Point Capabilities

Although Floating Point instructions have been defined in the HSP Instruction Set (Appendix A), the Simulator does not have the capability to execute them at present. When suitable floating point benchmark programs become available this capability will be added to the Simulator program by defining a Floating Point Register File, together with new types of Functional Unit that will handle the floating point instructions.

13.2 Coping with I/O

At present the Superscalar Simulator is unable to deal with source programs that request data input or output, such as manipulating files or interfacing with a display screen. It is intended to implement these features by having a built-in set of library functions, such as "printf" and "scanf", which have associated reserved Data Cache addresses. A call to one of these library procedures will cause a Memory Load or Memory Store instruction to access one of these reserved Data Cache addresses, analogous to the action of an auto-vectoring mechanism in traditional computers. An access to a reserved Data Cache address is detected by the Simulator and activates a procedure that makes system calls to the host computer in order to achieve the desired result. Parameters for these system procedure calls are passed via specific registers in the Simulator, the identities of which must be made known to the compiler so that registers are allocated correctly.

Currently, the "printf" function has been implemented to a limited extent in that it can handle a control string that includes up to six formatted parameters, as long as these parameters are themselves not of the type "string" or "float". It may prove necessary to alter the current implementation model for the Data Cache to one that more closely resembles the actual memory structure of the host computer so that pointers to data structures can be passed to system procedure calls.

13.3 The Next Step - A Scheduler Program

The Superscalar Simulator provides a number of novel and complex features designed to support the needs of highly scheduled source code. The potential power of some of these features, such as the ability to squash conditional code in the Instruction Buffer, cannot be tested by running the current versions of the benchmark programs, with the limited hand-scheduling that they employ. Work is currently under way to develop useful components of a general scheduling algorithm that will draw on scheduling techniques put forward in recent research papers **[9,20]**. By extending the scope of some of these techniques to take account of

explicitly declared branch delay slots and applying them to the scheduling of sequential code for the HSP Simulator, it is hoped to achieve substantial increases in values for speed-ups seen to date, perhaps as high as four over the Base1 model.

The Instruction Scheduler when written will adopt the same parametrised approach as the Superscalar Simulator program did, in that all the major features of the scheduling algorithm will be switchable. This flexibility will allow different scheduling strategies to be compared and contrasted with the hope that an optimal mix of scheduling features can be found which gives good performance over a range of HSP models.

References

1) Steven, G.
"iHARP Instruction Set Specification"
Computer Science Technical Report No.124,
School of Information Sciences, University of Hertfordshire, June 1991

2) Collins, R.
"Towards a Minimal Superscalar Implementation"
Computer Science Technical Note UHCS-93-N1,
School of Information Sciences, University of Hertfordshire, February 1993

3) Katevenis, M. G. H.
"Reduced Instruction Set Computer Architecture for VLSI"
The MIT Press, 1985

4) Steven, G. B. and Gray, S. M.
"Specification of a Machine Model for the HARP Architecture & Instruction Set - Version 3",
Computer Science Technical Report No.117,
School of Information Sciences, University of Hertfordshire, January 1991

5) Steven, G. B., Adams, R. G., Findlay, P. A. and Trainis, S. A.
"iHARP : A Multiple Instruction Issue Processor"
Computer Science Technical Report No.125,
School of Information Sciences, University of Hertfordshire, November 1991

6) Hwu, W. W. et al
"IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors"
Proceedings of the 18th Annual International Conference on Computer Architecture,
Toronto, May 1991.

7) Gray, S. M.
"Code Generation for Long Instruction Word Architectures",
PhD Thesis, University of Hertfordshire, December 1991

8) Johnson, M.
"Superscalar Microprocessor Design"
Prentice-Hall, Englewood Cliffs, New Jersey, 1991

9) Moon, S. M. and Ebcioglu, K.
"An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors"
Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, December 1992, pp 55-71

10) Butler, M. and Patt, Y.
"An Investigation of the Performance of Various Dynamic Scheduling Techniques"
Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, December 1992, pp 1-9

11) Lam, M. and Wilson, R. P.
"Limits of Control Flow on Parallelism"
Proceedings of the 19th Annual Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp 46-57

12) Wall, D. W.
"Limits of Instruction-Level Parallelism"
Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April 1991, ACM, pp 176-188

13) Alpert, D. and Avnon, D.
"Architecture of the Pentium Microprocessor"
IEEE Micro, June 1993, pp 11-21

14) McLellan, E.
"The Alpha AXP Architecture and 21064 Processor"
IEEE Micro, June 1993, pp 36-47

15) Wang, L.
"Instruction Scheduling for a Family of Multiple-Instruction-Issue Architectures"
PhD Thesis, University of Hertfordshire, December 1993

16) Steven, G. B.
"A Novel Effective Address Calculation Mechanism for RISC Microprocessors"
SIGARCH, September 1988, pp 150-156

17) Yeh, T. Y. and Patt, Y. N.
"Alternative Implementations of Two-Level Adaptive Branch Prediction"
Proceedings of the 19th Annual Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp 124-134

18) Hwu, W. W., Conte, T. M., and Chang, P. P.
"Comparing Software and Hardware Schemes for Reducing the Cost of Branches",
Proceedings of the 16th International Symposium on Computer Architecture,
April 1989, pp 224-231

19) Collins, R.
"Scheduling Code for Delayed Branch Instructions in the HSP Architecture"
Computer Science Technical Note,
School of Information Sciences, University of Hertfordshire, February 1994

20) Lam, M. S.
"Software Pipelining: An Effective Scheduling Technique for VLIW Machines"
Proceedings of the SIGPLAN 88 Conference of Programming Language Design
and Implementation, Georgia, USA, June 1988, pp 318-328

21) Hennessy, J. and Gross, T.
"Postpass Code Optimization of Pipeline Constraints"
ACM Transactions on Programming Languages and Systems,
Vol. 5, N0 3, July 1983, pp 422-448

Appendix A
The Hatfield Superscalar Processor Instruction Set

The instruction set for the Hatfield Superscalar Processor (HSP) is based on the HARP instruction set [1] and retains the HARP formats wherever possible. However, the instruction set is not a superset of iHARP as some iHARP instructions are inappropriate and have been dropped. A significant number of new instructions have also been added that can implement combined operations involving three source operands, and others to add floating point capabilities. All instructions can have multiple Boolean guards and can be marked for speculative execution, except for Memory Stores.

The instruction set is likely to evolve as the project progresses and new instructions may be added in the future. The current HSP instruction set is as follows:

1 Arithmetic Unit Instructions

ADD Ri,Rj,Rk	Ri := Rj + Rk
ADD Ri,Rj,#Imm	Ri := Rj + #Imm
ADDV Ri,Rj,Rk	Ri := Rj + Rk ; trap on signed arithmetic overflow
ADDV Ri,Rj,#Imm	Ri := Rj + #Imm; trap on signed arithmetic overflow
SUB Ri,Rj,Rk	Ri := Rj - Rk
SUB Ri,Rj,#Imm	Ri := Rj - #Imm
SUBV Ri,Rj,Rk	Ri := Rj - Rk ; trap on signed arithmetic overflow
SUBV Ri,Rj,#Imm	Ri := Rj - #Imm; trap on signed arithmetic overflow
ADDC Ri,Rj,Rk	Ri := Rj + Rk + Carry; Carry flag set by result
ADDC Ri,Rj,#Imm	Ri := Rj + #Imm + Carry; Carry flag set by result
SUBC Ri,Rj,Rk	Ri := Rj - Rk + Carry; Carry flag set by result
SUBC Ri,Rj,#Imm	Ri := Rj - #Imm + Carry; Carry flag set by result
DIV Ri,Rj,Rk	Ri := Rj DIV Rk ; 32bits X 32 bits --> 32 bits
DIV Ri,Rj,#Imm	Ri := Rj DIV Imm ; 32bits X 32 bits --> 32 bits
DIVV Ri,Rj,Rk	Ri := Rj DIV Rk ; 32bits X 32 bits --> 32 bits; trap on overflow
DIVV Ri,Rj,#Imm	Ri := Rj DIV Imm ; 32bits X 32 bits --> 32 bits; trap on overflow
MOD Ri,Rj,Rk	Ri := Rj MOD Rk ; non pipelined; 32bits X 32 bits --> 32 bits
MOD Ri,Rj,#Imm	Ri := Rj MOD Imm ; non pipelined; 32bits X 32 bits --> 32 bits
MOV Ri, Bj	If Bj = 1 Ri = 1 else Ri = 0
MOV Ri,SR	

2 Shift Unit Instructions

ASL Ri,Rj,Rk	Ri := Rj << (Rk AND 31)
ASL Ri,Rj,#Imm	Ri := Rj << #Imm where (0 <= Imm < 32)
ASLV Ri,Rj,Rk	Ri := Rj << (Rk AND 31); trap on overflow
ASLV Ri,Rj,#Imm	Ri := Rj << #Imm where (0 <= Imm < 32); trap on overflow
ASR Ri,Rj,Rk	Ri := Rj >> (Rk AND 31)
ASR Ri,Rj,#Imm	Ri := Rj >> #Imm where (0 <= Imm < 32)
AND Ri,Rj,Rk	Ri := Rj AND Rk
AND Ri,Rj,#Imm	Ri := Rj AND #Imm
OR Ri,Rj,Rk	Ri := Rj OR Rk
OR Ri,Rj,#Imm	Ri := Rj OR #Imm
EOR Ri,Rj,Rk	Ri := Rj EOR Rk
EOR Ri,Rj,#Imm	Ri := Rj EOR #Imm
EXT Ri,Rj	Ri := Rj (byte sign extended); for GNUCC
BIC Ri,Rj,Rk	Ri := Rj AND ~(Rk)
BIC Ri,Rj,#Imm	Ri := Rj AND ~(#Imm)

3 Multiply Unit Instructions

MULT Ri,Rj,Rk	Ri := Rj MULT Rk; 32bits X 32 bits --> 32 bits
MULT Ri,Rj,#Imm	Ri := Rj MULT #Imm ; 32bits X 32 bits --> 32 bits
MULTV Ri,Rj,Rk	Ri := Rj MULT Rk; 32bits X 32 bits --> 32 bits; trap on signed overflow
MULTV Ri,Rj,#Imm	Ri := Rj MULT #Imm ; 32bits X 32 bits --> 32 bits; trap on signed overflow

4 Relational Unit Instructions

Signed:

GTS Bi,Rj,Rk	Bi := (Rj > Rk)
GTS Bi,Rj,#Imm	Bi := (Rj > #Imm)
GES Bi,Rj,Rk	Bi := (Rj >= Rk)
GES Bi,Rj,#Imm	Bi := (Rj >= #Imm)
LTS Bi,Rj,Rk	Bi := (Rj < Rk)
LTS Bi,Rj, #Imm	Bi := (Rj < #Imm)
LES Bi,Rj,Rk	Bi := (Rj =< Rk)
LES Bi,Rj, #Imm	Bi := (Rj =< #Imm)

Unsigned:

GTU Bi,Rj,Rk	Bi := (Rj > Rk)
GTU Bi,Rj,#Imm	Bi := (Rj > #Imm)
GEU Bi,Rj,Rk	Bi := (Rj >= Rk)
GEU Bi,Rj,#Imm	Bi := (Rj >= #Imm)
LTU Bi,Rj,Rk	Bi := (Rj < Rk)
LTU Bi,Rj, #Imm	Bi := (Rj < #Imm)
LEU Bi,Rj,Rk	Bi := (Rj =< Rk)
LEU Bi,Rj, #Imm	Bi := (Rj =< #Imm)

Signed and Unsigned

EQ Bi,Rj,Rk	Bi := (Rj = Rk)
EQ Bi,Rj, #Imm	Bi := (Rj = #Imm)
NE Bi,Rj,Rk	Bi := (Rj <> Rk)
NE Bi,Rj, #Imm	Bi := (Rj <> #Imm)

Boolean Instructions

AND Bi,Bj,Bk	Bi := Bj AND Bk
OR Bi,Bj,Bk	Bi := Bj OR Bk

EQ Bi,Bj,Bk	Bi := Bj = Bk
NE Bi,Bj,Bk	Bi := Bj <> Bk
GT Bi,Bj,Bk	Bi := Bj > Bk
LT Bi,Bj,Bk	Bi := Bj < Bk
LE Bi,Bj,Bk	Bi := Bj <= Bk
GE Bi,Bj,Bk	Bi := Bj >= Bk
MOV Bi, Rj	Bi := Rj ; Bi := l.s. bit of Rj; To define registers in GNU it must be possible to transfer data between registers
MOV SR,Ri MOV SR,#Imm	SR contains Boolean registers

5 Memory Reference

LD Bi,offset(Rj)	Bi := Mem[offset + Rj]; Bi set to lsb of memory byte
LD Bi,(Rj,Rk)	Bi := Mem[Rj + Rk]; Bi set to lsb of memory byte
LDB Ri,offset(Rj)	Ri := Mem[offset + Rj]; load sign extended byte
LDB Ri,(Rj,Rk)	Ri := Mem[Rj + Rk]; load sign extended byte
LD Ri,offset(Rj)	Ri := Mem[offset + Rj]; load word
LD Ri,(Rj,Rk)	Ri := Mem[Rj + Rk] ; load word
LDD Ri,offset(Rj)	Ri:Ri+1 := Mem[offset + Rj]; load double words
LDD Ri,(Rj,Rk)	Ri:Ri+1 := Mem[Rj + Rk]
LDQ Ri,offset(Rj)	Ri:Ri+1:Ri+2:Ri+3 := Mem[offset + Rj]; load quad words
LDQ Ri,(Rj,Rk)	Ri:Ri+1:Ri+2:Ri+3 := Mem[Rj + Rk]
LD SR,offset(Rj)	SR := Mem[offset + Rj]; load word
LD SR,(Rj,Rk)	SR := Mem[Rj + Rk]; load word
ST offset(Rj),Bi	Mem[offset + Rj] := Bi; least significant bit of memory byte set to Bi; other bits cleared
ST (Rj,Rk),Bi	Mem[Rj + Rk] := Bi least significant bit of memory byte set to Bi; other bits cleared
STB offset(Rj),Ri	Mem[offset + Rj] := Ri ; store byte only
STB (Rj,Rk),Ri	Mem[Rj + Rk] := Ri; store byte only

ST offset(Rj),Ri	Mem[offset + Rj] := Ri
ST (Rj,Rk),Ri,	Mem[Rj + Rk] := Ri
STD offset(Rj),Ri	Mem[offset + Rj] := Ri:Ri+1; store double words
STD (Rj,Rk),Ri	Mem[Rj + Rk] := Ri:Ri+1
STQ offset(Rj),Ri	Mem[offset + Rj] := Ri:Ri+1:Ri+2:Ri+3; store quad words
STQ (Rj,Rk),Ri	Mem[Rj + Rk] := Ri:Ri+1:Ri+2:Ri+3
ST offset(Rj),SR	Mem[offset + Rj] := SR; 32 bits stored
ST (Rj,Rk),SR	Mem[Rj + Rk] := SR; 32 bits stored

6 Branch Instructions

BT Bi,label (#delay-count)	
BF Bi,label (#delay-count)	
BSR Ri,label (#delay-count)	Save return address in Ri
MOV PC,Ri (#delay-count)	
TRAP #n,Bi (#delay-count)	Normal delayed branch used to enter opsy routines; Delay count = 0 for debugging and faults
TRAP #n,(#delay count)	

7 Special Purpose Instructions

EI	Enable interrupts
DI	Disable interrupts

8 Floating-point Relational Unit

GTSSF Bi,Fj,Fk	Bi := (Fj > Fk)
GTSDF Bi,Fj,Fk	Bi := (Fj > Fk)
GESSF Bi,Fj,Fk	Bi := (Fj >= Fk)
GESDF Bi,Fj,Fk	Bi := (Fj >= Fk)
LTSSF Bi,Fj,Fk	Bi := (Fj < Fk)

LTSDF Bi,Fj, Fk	Bi := (Fj < Fk)
LESSF Bi,Fj,Fk	Bi := (Fj =< Fk)
LESDF Bi,Fj, Fk	Bi := (Fj =< Fk)
EQSF Bi,Fj,Fk	Bi := (Fj = Fk)
EQDF Bi,Fj, Fk	Bi := (Fj = Fk)
NESF Bi,Fj,Fk	Bi := (Fj <> Fk)
NEDF Bi,Fj, Fk	Bi := (Fj <> Fk)

9 Floating-point Add Unit

ADDSF Fi,Fj,Fk	Fi := Fj + Fk
ADDDF Fi,Fj,Fk	Fi := Fj + Fk
SUBSF Fi,Fj,Fk	Fi := Fj - Fk
SUBDF Fi,Fj,Fk	Fi := Fj - Fk
NEGSF Fi,Fj	Fi := - Fj
NEGDF Fi,Fj	Fi := - Fj
DIVSF Fi,Fj,Fk	Fi := Fj DIV Fk
DIVDF Fi,Fj,Fk	Fi := Fj DIV Fk
MODSF Fi,Fj,Fk	Fi := Fj MOD Fk
MODDF Fi,Fj,Fk	Fi := Fj MOD Fk

MOVSF Fi,Fj
MOVDF Fi,Fj
EXTDF Fi,Fj
TRUNCSF Fi,Fj
MOV Ri,Fj
MOVSF Fi,Rj

10 Floating-point Multiply Unit

MULTSF Fi,Fj,Fk	Fi := Fj MULT Fk
MULTDF Fi,Fj,Fk	Fi := Fj MULT Fk

11 Floating-point Memory Reference

LDSF Fi,offset(Rj)	Ri := Mem[offset + Rj]; load word
LDSF Fi,(Rj,Rk)	Ri := Mem[Rj + Rk] ; load word
LDDF Fi,offset(Rj)	Fi:Fi+1 := Mem[offset + Rj]; load double words
LDDF Fi,(Rj,Rk)	Fi:Fi+1 := Mem[Rj + Rk]
LDQF Fi,offset(Rj)	Fi:Fi+1:Fi+2:Fi+3 := Mem[offset + Rj]; load quad words
LDQF Fi,(Rj,Rk)	Fi:Fi+1:Fi+2:Fi+3 := Mem[Rj + Rk]
STSF offset(Rj),Fi	Mem[offset + Rj] := Fi
STSF (Rj,Rk),Fi,	Mem[Rj + Rk] := Fi
STDF offset(Rj),Ri	Mem[offset + Rj] := Fi:Fi+1; store double words
STDF (Rj,Rk),Ri	Mem[Rj + Rk] := Fi:Fi+1
STQF offset(Rj),Ri	Mem[offset + Rj] := Fi:Fi+1:Fi+2:Fi+3; store quad words
STQF (Rj,Rk),Ri	Mem[Rj + Rk] := Fi:Fi+1:Fi+2:Fi+3

Appendix B
A User's Manual for the Superscalar Simulator

This manual is written for a novice user and is presented in the form of an interactive tutorial where the user can carry out a typical simulation run by following the directions given. An index for the many figures contained in this document is included at the end of the manual to serve as quick reference guide for the more experienced user.

The Superscalar Simulator program implements the HSP architecture and is driven via a series of screen menus that are presented to the user, each detailing about seven option choices that can be selected. Once a particular option is taken, further menus are presented to allow the user to provide detailed information as to the exact action required. Once an action has been completed, the user can stay on the same menu or return to the previous menu and hence back to the top level "Main Menu". This simple menu structure, coupled with single key entries for most options, allows the Simulator model to be easily and rapidly tailored to the user's requirements.

The first step is to activate the Simulator program by typing "sim" which will result in a few lines of user information appearing on the screen giving details of the amount of host computer memory used to build the Instruction Cache. This information is of use when running the Simulator program on a PC where available memory is at a premium. Press the "enter" key several times until the first page of the two-page Main Menu appears on the screen as shown in Figure 1.

```
                    SIMULATOR MAIN MENU (version 1.0)

        "R"         REGISTER CONTENTS DISPLAY

        "V"         VIEW FILES

        "U"         FUNCTIONAL UNITS STATUS

        "S"         SELECT INPUT FILES

        "D"         DATA CACHE CONTENTS

        "I"         INSTRUCTION CACHE CONTENTS

        "F"         FORMAT INSTRUCTION CACHE

        "O"         ONE INSTRUCTION RECORD

        "A"         ACTIVATE THE PROGRAM

        "2"         MAIN MENU PAGE 2

PRESS "X" TO EXIT THE PROGRAM          SELECTION = ?
```

Figure 1 Main Menu page 1

There are a number of menu screens used by various options in the Simulator and it is useful to see the overall structure and interdependence of these menus, as illustrated in Figure 2.

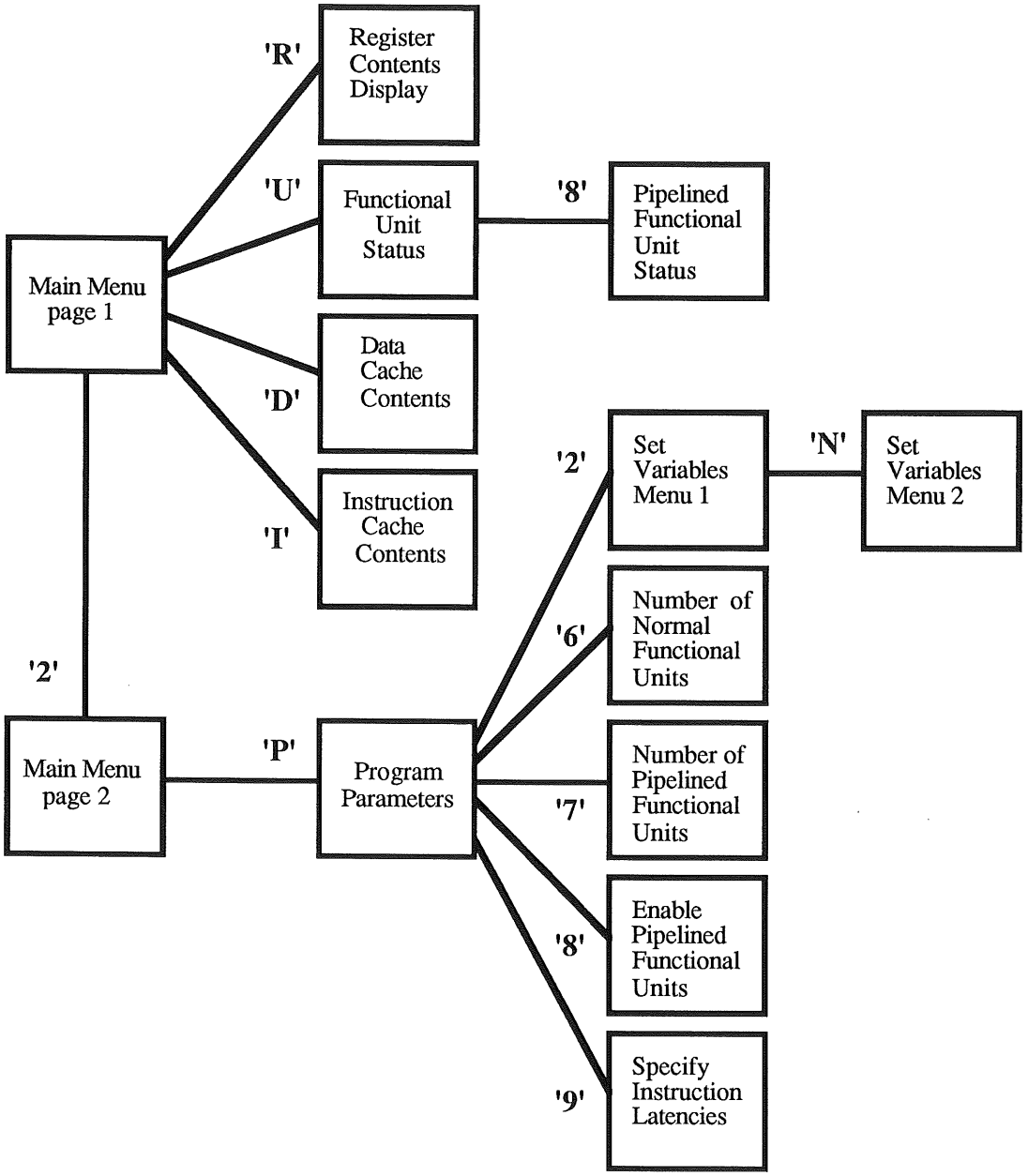

Figure 2 Menu Structure for the Simulator

Each of these screen menus contains up to nine options that will alter the Simulator model or allow the user to get information about the internal workings of the Simulator's data structures, as well as being able to activate various procedures and processes.

Continue by pressing "2" to examine the options available on page 2 of the Main Menu, as shown in Figure 3, then press "M" to return to page 1 once more.

```
                    SIMULATOR MAIN MENU PAGE 2

        "L"         LIST INSTRUCTION SET

        "F"         FETCH-UNIT STATUS

        "S"         STATE OF MACHINE

        "P"         PROGRAM PARAMETERS

        "U"         USAGE OF FUNCTIONAL UNITS

        "B"         DISPLAY INSTRUCTION BUFFER

        "I"         STORE ICACHE USAGE IN SPECIFIED FILE

        "W"         WRITE FILE TO PRINTER

PRESS "M" TO EXIT BACK TO THE MAIN MENU        SELECTION = ?
```

Figure 3 Main Menu page 2

Operation of the Superscalar Simulator is best explained by following a worked example which involves the use of most of the facilities provided for user interaction. The example chosen is to run a simulation of the Stanford Integer Benchmark program "perm". The reader should follow the instructions given in the next sections and check the screen displays seen against the data shown in the figures in this document.

The first thing to do on the Simulator is to select an input source file for the program to be simulated by pressing "S" at the Main Menu page 1. The user is then prompted to enter the filename of the source program, which is assumed to always have an extension of ".ins". If the source file is not in the current directory of the host computer the full path must be specified and the filename should always be entered without the ".ins" extension. Type in the filename "perm" then press "enter" and the appropriate source file will be read into the Instruction Cache. If any errors are detected whilst reading in the individual instructions, an appropriate error message will appear on the screen to help identify the cause of the fault. If the source program is read in correctly, the "Format Instruction Cache" option is automatically called so that the symbol tables are created. These symbol tables are then used to "fix-up" the program code in the Instruction Cache and to allocate the data structures and jump lists in the Data Cache. During this process, information concerning the contents of the symbol tables appears on the screen and the user is prompted to press "enter" several times before the Main Menu is displayed again. Typical Symbol table information for the example program "perm" is shown in Figure 4.

```
WHAT IS THE NAME FOR THE INPUT PROGRAM FILE ?
FILENAME = instr/perm1

                    CONTENTS OF THE SYMBOL TABLE
NAME            ADDRESS     NAME            ADDRESS     NAME            ADDRESS

_dummy          I 3         _Swap           I 6         _Initialize     I 13
L6              I 16        _Permute        I 25        L12             I 46
L8              I 60        _Perm           I 67        L18             I 72
L19             I 83        _main           I 87        printf_str      L 4100

program length is 93

THE INSTRUCTION CACHE HAS BEEN FORMATTED
  (LABELS AND MEMORY ADDRESSES COPIED)

icache table deleted
working table deleted

     Press 'Enter' to continue.......
```

Figure 4 Symbol Table for the "perm" Program

Every label used in the source program file corresponds to an address in the Instruction or Data Caches, as indicated by a prefix of "I" or "D". Any library structures supplied by the Simulator are shown with a prefix of "L" and are allocated addresses in Data Cache. Now that the Instruction Cache has been loaded and formatted, press "I" to display the contents of the Instruction Cache. A menu will appear giving different display options as shown in Figure 5.

```
          DISPLAY INSTRUCTION CACHE OPTION

          ICACHE VALID ADDRESS RANGE IS 0 TO 92

     '1' -         DISPLAY FROM ICACHE BASE ADDRESS

     '2' -         SPECIFY START ADDRESS FOR DISPLAY

     'M' -         RETURN TO MAIN MENU

      OPTION SELECTED = ?
```

Figure 5 Instruction Cache Display Options

Select option "1" to display all the Instruction Cache which will appear in assembler mnemonic form on the screen, one page at a time. Each line starts with the Instruction Cache word address followed by a dynamic usage count for each instruction which is initially set to zero. The first page of Instruction Cache contents for the "perm" program is shown in Figure 6. Exit back to the Main Menu at any time by pressing "Q" to quit the display option.

```
                    *** CONTENTS OF INSTRUCTION CACHE ***

  0     0                                          MOV  GP,#4096
  1     0                                          MOV  SP,#4096
  2     0                                          BSR  RA,_main (0){87}
  3     0 _dummy                                   SUB  SP,SP,#128
  4     0                                          ADD  SP,SP,#128
  5     0                                          MOV  PC,RA(0)
  6     0 _Swap                                    SUB  SP,SP,#128
  7     0                                          LD   R7,(R0,R5)
  8     0                                          LD   R8,(R0,R6)
  9     0                                          ST   (R0,R5),R8
 10     0                                          ST   (R0,R6),R7
 11     0                                          ADD  SP,SP,#128
 12     0                                          MOV  PC,RA(0)
 13     0 _Initialize                              SUB  SP,SP,#128
 14     0                                          MOV  R7,#1
 15     0                                          ADD  R8,GP,#12
 16     0 L6                                       ASL  R5,R7,#2
 17     0                                          ADD  R5,R8,R5
 18     0                                          ADD  R6,R7,#-1
 19     0                                          ST   (R0,R5),R6

more to come.... press 'C' to continue or 'Q' to quit
```

Figure 6 Instruction Cache Contents

Now the parameters must be selected for the Superscalar Simulator model by using an option on page 2 of the Main Menu. Press "2" for page 2 then "P" for the "Program Parameters" option. Select option "1" to display the current default settings for the machine model and the screen will give the display illustrated in Figure 7.

```
                CURRENT STATE OF SIMULATOR PROGRAM PARAMETERS

ICACHE_SIZE = 1000     DCACHE_SIZE = 10000  DCACHE_BASE_ADDRESS = 0
INTEGER_FILE_LENGTH = 33  BOOLEAN_FILE_LENGTH = 16
SR = R32     GP = R31     SP = R30     SP' = R29     RA = R28

max_clocks = 1000000          clock_interval = 100        squash_enable = 1
hold_branches = 0             max_buffer_length = 256 (256)   waw_hold  = 0
num_pipe_units = 32 (32)      enable_library = 0   status _report_pause = 0
fetch_unit_width = 64 (64)    fetch_cycles = 2 (4)      div_latency  = 8 (8)
num_result_busses = 24        debug = 0               zero_branch_delay = 0
record_trace = 0    trace_start = -1     trace_end = -1     reg_option = 0

Arith Units  = 16 (16) Arith Pipe Units = 16 (16) latency = 1 (4) Pipe = 1
Load Units   = 16 (16) Load Pipe Units  = 16 (16) latency = 1 (4) Pipe = 1
Store Units  = 16 (16) Store Pipe Units = 16 (16) latency = 1 (4) Pipe = 1
Bool Units   = 16 (16) Bool Pipe Units  = 16 (16) latency = 1 (4) Pipe = 1
Shift Units  = 16 (16) Shift Pipe Units = 16 (16) latency = 1 (4) Pipe = 1
Branch Units = 16 (16)                            LATENCY = 1     Pipe = 0
Mult Units   = 16 (16) Mult Pipe Units  = 16 (16) latency = 6 (6) Pipe = 1
```

Figure 7 Default Parameter Settings

Figure 7 contains all the important machine model parameters, most of which can be altered by the user. Names given in upper case are for fixed parameters that are defined in a global header file while all lower case parameters can be specified interactively by the user, up to a maximum value shown in brackets. Instruction Cache and Data Cache sizes are displayed as fixed constants, along with the base address used for the Data Cache. The number of Integer and Boolean Registers available are given, as well as the mappings for the four special purpose registers.

The variable "max clocks" determines the maximum permissible number of machine clock cycles allowed before a trap is forced to terminate a program. The "clock interval" parameter specifies the number of clock cycles between "status reports" which are one-line snap-shots of the instruction currently in pipe 1 of the Instruction Decode group. A Boolean switch "status report pause" provides a means to pause when the status lines fill the display screen and prompts the user to press "enter" to continue with the simulation.

A switch called "squash enable" can be deactivated to prevent the squashing of conditionally executed instructions while they are in the Instruction Buffer so that the performance of this important architectural feature can be assessed. Another Boolean switch "hold branches" imposes a restrictive model on the Simulator where no Branch instructions can be issued until their last branch-dependent instruction has been fetched into the Instruction Buffer. The signal "waw hold" prevents an instruction from being issued until all Functional Units that target the same destination register have finished processing, rather than "killing" these Functional Units.

Instruction Buffer capacity and the number of "pipes" available for instruction issue are determined by "max buffer length" and "num pipe units" respectively, and these can take a value between one and the maximum indicated by their corresponding bracketed values. The Boolean switch "enable library" allows the Simulator to perform host system calls when certain reserved Data Cache locations are accessed. These system calls can perform specific input/output operations such as "printf" and "scanf", if these are called for in the source program.

The size of the block of instructions that are fetched from the Instruction Cache is determined by "fetch unit width" and can be set to any value in the range one to 64. However, the "fetch unit width" must never be set to a value greater than the "max buffer size" or the Simulator will never find enough room in the Instruction Buffer in order to load in a block of instructions. The parameter "fetch cycles" can be set to a value in the range one to four and represents the minimum number of cycles required to access the Instruction Cache and load new instructions into the Instruction Buffer, equivalent to the IF stage.

The latency for Divide instructions is set by "div latency" in the range one to eight and this is used to force an Arithmetic Functional Unit to extend its normal latency when it processes a Divide instruction. The number of Result Buses available can be specified by "num result buses" up to a maximum of 24, with low values causing a performance loss when Functional

Units compete for access to these busses. The Boolean switches, "debug" and "zero branch delay", are special-purpose facilities to aid program development and to help generate Base 1 model statistics respectively.

There are four parameters that are useful for recording a full or partial trace of a program's execution. The Boolean switch "record trace" activates this facility and "reg option" determines what register information is included. The recorded trace consists of the cycle count and the instructions successfully issued during that cycle. The contents of the Integer and Boolean Registers can also be recorded, depending on the setting of "reg option". The parameter "reg option" can take values of "1", "2" or "3" which will record no values, new values only or all active values respectively, with reference to the contents of the register files. If "trace start" or "trace end" have positive values they will determine the Instruction Cache start and end addresses for the trace being recorded.

Finally, parameters can be set that determine the number of each type of Functional Units that will comprise the Simulator's common pool of processing resources, and the corresponding "Pipe" switches specify if pipelined or non-pipelined versions are to be used. The latencies for each type of Functional Unit can be altered in the range one to four, except for Multiple which has a range of one to six and Divide which has a range of one to eight. Branch Units are special as they complete much of their work in the ID stage rather than the EX stage and are always present as a non-pipelined implementation with a fixed latency of one.

The large number of user specified parameters can allow for a vast range of Simulator models to be investigated and certain commonly used model specifications can be selected from the "Program Parameters" menu shown in Figure 8.

```
            OPTION 'P' -   PROGRAM PARAMETERS OPTION
THIS SCREEN ALLOWS THE PROGRAM PARAMETERS TO BE EXAMINED AND ALTERED

          '1' -   DISPLAY ALL PROGRAM PARAMETERS

          '2' -   SET VARIABLES

          '3' -   LOAD MINIMAL 1-PIPE MODEL

          '4' -   LOAD WORKING MODEL

          '5' -   LOAD MAXIMUM MODEL

          '6' -   SPECIFY NUMBER OF NORMAL FUNCTIONAL UNITS REQUIRED

          '7' -   SPECIFY NUMBER OF PIPELINED FUNCTIONAL UNITS REQUIRED

          '8' -   ENABLE PIPE UNITS

          '9' -   SPECIFY INSTRUCTION LATENCIES
```

Figure 8 Program Parameters Menu

Options "3", "4" and "5" force the Simulator to adopt a set of predefined parameter settings so that the user can switch quickly between models without altering the parameters individually. These models allow for a restrictive one-pipe model as well as the "full power" default model that we have just seen. Option "4" selects a Working Model which sets the parameters to the values shown in Figure 9.

```
                CURRENT STATE OF SIMULATOR PROGRAM PARAMETERS

ICACHE_SIZE = 1000     DCACHE_SIZE = 10000  DCACHE_BASE_ADDRESS = 0
INTEGER_FILE_LENGTH = 33  BOOLEAN_FILE_LENGTH = 16
SR = R32      GP = R31      SP = R30      SP' = R29      RA = R28

max_clocks = 1000000          clock_interval = 100          squash_enable = 1
hold_branches = 0             max_buffer_length = 32 (256)    waw_hold  = 0
num_pipe_units = 8 (32)       enable_library = 0    status_report_pause = 0
fetch_unit_width = 16 (64)    fetch_cycles = 2 (4)      div_latency  = 8 (8)
num_result_busses = 24        debug = 0                zero_branch_delay = 0
record_trace = 0    trace_start = -1     trace_end = -1     reg_option = 0

Arith Units  = 8  (16) Arith Pipe Units = 8  (16) latency = 1 (4) Pipe = 1
Load Units   = 2  (16) Load Pipe Units  = 2  (16) latency = 1 (4) Pipe = 1
Store Units  = 2  (16) Store Pipe Units = 2  (16) latency = 1 (4) Pipe = 1
Bool Units   = 4  (16) Bool Pipe Units  = 4  (16) latency = 1 (4) Pipe = 1
Shift Units  = 2  (16) Shift Pipe Units = 2  (16) latency = 1 (4) Pipe = 1
Branch Units = 2  (16)                           LATENCY = 1     Pipe = 0
Mult Units   = 2  (16) Mult Pipe Units  = 2  (16) latency = 6 (6) Pipe = 1
```

Figure 9 Working Model Parameter Settings

Note that this model currently gives an Instruction Buffer of 32, parallel issue capacity of 8 and a fetch block size of 16 with a fetch latency of 2. Pipelined versions of the Functional Units are enabled as is normal branch behaviour and the ability to squash instructions in the Instruction Buffer. There is no particular significance in the exact values chosen for this model and it can be altered by re-compiling the Simulator program to whatever mix of parameter values is felt to be most appropriate and useful as a working model.

If a user wishes to deviate from one of the pre-programmed machine models, this can be achieved via one of the several menus provided for this purpose. Continue the Simulation run by returning to the "Program Parameters" menu and then select option "2" to "Set Variables" which gives the menu shown in Figure 10. This option allows the user to set or deactivate various Boolean control switches and to specify some parameter values.

```
THIS SCREEN ALLOWS SOME OF THE PROGRAM PARAMETERS TO BE ALTERED
        '1' -    SET "STATUS REPORT PAUSE"
        '2' -    SET MAXIMUM CYCLE COUNT ALLOWED FOR A RUN
        '3' -    SET NUMBER OF CLOCK CYCLES BETWEEN STATUS REPORTS
        '4' -    SPECIFY NUMBER OF FETCH CYCLES
        '5' -    SET "HOLD BRANCHES"
        '6' -    SPECIFY INSTRUCTION BUFFER LENGTH
        '7' -    SPECIFY FETCH UNIT WIDTH
        '8' -    SET "FULL TRACE"
        '9' -    SET "SQUASH ENABLE"
        'N' -    NEXT MENU OF VARIABLES
   PRESS "M" TO RETURN TO PREVIOUS MENU    SELECTION = ?
```

Figure 10 Select Variables Menu page 1

The "Full Trace" control switch refers to the quantity of information displayed if an interactive program trace is being performed, and will cause details of all the Functional Units to be shown during each clock cycle if the switch is activated. Press "N" to view additional parameters that can be set on page 2 of the Set Variables menu as shown in Figure 11.

```
THIS SCREEN ALLOWS SOME OF THE PROGRAM PARAMETERS TO BE ALTERED
        '1' -    SET "RECORD TRACE"
        '2' -    SET "WAW HOLD"
        '3' -    SPECIFY NUMBER OF RESULT BUSSES
        '4' -    SET "DEBUG"
        '5' -    SET "ZERO BRANCH DELAY"
        '6' -    SET "ENABLE LIBRARY"

   PRESS "M" TO RETURN TO PREVIOUS MENU    SELECTION = ?
```

Figure 11 Select Variable Menu page 2

Option "1" on page 2 of the Set Variables menu enables the user to activate the "record trace" facility and prompts the user for start and end addresses for the trace as well as selecting which register information is to be included, as shown in Figure 12.

```
PRESS '1' TO ACTIVATE THE SIGNAL OR '0' TO DE-ACTIVATE

       SELECTION = 1

Enter start address for trace recording = 45

Enter end address for trace recording   = 67

Enter option setting for recording register contents

          '1' -    NO REGISTERS CONTENTS RECORDED

          '2' -    ACTIVE FORWARD REGISTER VALUES ONLY

          '3' -    LATEST VALUES FOR ALL ACTIVE REGISTERS

           SELECTION = ?
```

Figure 12 Record Trace Options

Return to the "Program Parameters" menu and select options "6", "7" and "8" in turn which will give the menus shown in Figures 13, 14 and 15 respectively. These options allow the user to determine how many pipelined and non-pipelined Functional Units will be available to the Simulator, and which implementation to choose for each class of instruction.

```
THIS SCREEN ALLOWS THE NUMBER OF NORMAL FUNCTIONAL UNITS TO BE SPECIFIED

          '1' -    SET NUMBER OF ARITHMETIC UNITS

          '2' -    SET NUMBER OF MEMORY LOAD UNITS

          '3' -    SET NUMBER OF MEMORY STORE UNITS

          '4' -    SET NUMBER OF BOOLEAN UNITS

          '5' -    SET NUMBER OF SHIFT UNITS

          '6' -    SET NUMBER OF BRANCH UNITS

          '7' -    SET NUMBER OF MULT UNITS

PRESS "M" TO RETURN TO PREVIOUS MENU     SELECTION = ?
```

Figure 13 Number of Non-Pipelined Functional Units

```
THIS SCREEN ALLOWS THE NUMBER OF PIPELINED FUNCTIONAL UNITS TO BE SPECIFIED

        '1' -    SET NUMBER OF PIPELINED ARITHMETIC UNITS
        '2' -    SET NUMBER OF PIPELINED MEMORY LOAD UNITS
        '3' -    SET NUMBER OF PIPELINED MEMORY STORE UNITS
        '4' -    SET NUMBER OF PIPELINED BOOLEAN UNITS
        '5' -    SET NUMBER OF PIPELINED SHIFT UNITS
        '7' -    SET NUMBER OF PIPELINED MULT UNITS
        '8' -    SET NUMBER OF PIPE UNITS
```

Figure 14 Number of Pipelined Functional Units

```
THIS SCREEN ALLOWS SOME OF THE PROGRAM PARAMETERS TO BE ALTERED

        '1' -    SET "USE PIPELINED ALU UNIT" SIGNAL
        '2' -    SET "USE PIPELINED MEMORY LOAD UNIT" SIGNAL
        '3' -    SET "USE PIPELINED MEMORY STORE UNIT" SIGNAL
        '4' -    SET "USE PIPELINED BOOLEAN UNIT" SIGNAL
        '5' -    SET "USE PIPELINED SHIFT UNIT" SIGNAL
        '7' -    SET "USE PIPELINED MULT UNIT" SIGNAL
```

Figure 15 Enable Pipelined or Non-Pipelined Functional Units

Individual instruction latencies can be specified by selecting Program Parameters option "9" which gives the menu shown in Figure 16.

```
THIS SCREEN ALLOWS THE LATENCY OF EACH INSTRUCTION TYPE TO BE SPECIFIED

        '1' -    SET ARITHMETIC LATENCY
        '2' -    SET MEMORY LOAD LATENCY
        '3' -    SET MEMORY STORE LATENCY
        '4' -    SET BOOLEAN LATENCY
        '5' -    SET SHIFT LATENCY
        '6' -    SET DIVIDE LATENCY
        '7' -    SET MULT LATENCY
```

Figure 16 Instruction Latencies Menu

Once the Simulator parameters have been set the user should select the option to display the parameter settings again to confirm that the model is correct. The user is now in a position to run the program from the Instruction Cache by returning to the Main Menu and selecting option "A" to activate the program. A series of interactive prompts will appear on the screen that determine how the program run is to be carried out, a typical sequence being shown in Figure 17. Note that answers to "yes/no" prompts can be given in upper or lower case.

```
                    ACTIVATE PROGRAM OPTION

THIS ALLOWS YOU TO SPECIFY THE START ADDRESS OF THE PROGRAM

DO YOU WANT TO USE AN ICACHE LABEL AS A START ADDRESS?  (Y/N)  n

START ADDRESS MUST BE BETWEEN 0 AND 93

          START ADDRESS = 0

DO YOU WANT TO DE-ALLOCATE MEMORY FROM THE ENTIRE DCACHE ? (Y/N)   y

DO YOU WANT TO RE-INITIALISE ALL REGISTER FILES ? (Y/N)   y

DO YOU WANT TO SET MAIN REGISTER FILES "ACTIVE" ? (Y/N)   y

DO YOU WANT TO SET MAIN REGISTER FILES "VALID" ? (Y/N)   y

DO YOU WANT TO TRACE THE PROGRAM'S EXECUTION ? (Y/N)   n

DO YOU WANT TO REPORT ERRORS ? (Y/N)   y

DO YOU WANT TO STOP ON ERRORS ? (Y/N)   y

DO YOU WANT TO TRACE AFTER ERROR STOP ? (Y/N)   y
```

Figure 17 Activate Program Option

The first prompt allows the user to specify an address label from which the program is to start running, or allows a specific Instruction Cache word address to be selected as a starting point. In the latter case the user is given the range of valid Instruction Cache addresses. A choice is given as to whether the Data Cache is to be reinitialised or the present contents are to be maintained and use of this option may depend on whether the source program file specified any initialised data or not. The next three options allow the user to reset all the Integer and Boolean Registers and to mark the Main Register files as "active" and "valid". These last two actions ensure that error messages are not produced as a result of the source program accessing the contents of a register that has not first been written to by the program. Such an access may occur on a procedure entry when registers to be used locally are first stored away in memory. However, these features can be useful when running diagnostic programs that should never read values from an uninitialised register.

The user is then given the option to dynamically trace the program's execution, which we will look at in more detail in a later section. Error options allow error messages to be displayed to

the user as they are detected and this can be made to cause a halt in program execution and to enter the dynamic trace mode thereafter, if required. Once all the prompts have been answered the program will start to run, with status report lines appearing at set intervals to show the user that useful progress is being made through the program, as illustrated in Figure 18. As the full status report is very long only the beginning and end lines are shown.

```
Real Time Start =  Mon Nov.  8 12:30:45 1993
clock_cycles = 100    PC = 89    Pipe Unit[0] =          ST     from 28
clock_cycles = 200    PC = 25    Pipe Unit[0] =          ADD    from 52
clock_cycles = 300    PC = 89    Pipe Unit[0] =          ST     from 30
clock_cycles = 400    PC = 89    Pipe Unit[0] =          ST     from 30
clock_cycles = 500    PC = 41    Pipe Unit[0] =          xxx    from 0
clock_cycles = 600    PC = 89    Pipe Unit[0] =          ADD    from 33
clock_cycles = 700    PC = 89    Pipe Unit[0] =          ST     from 34
clock_cycles = 800    PC = 73    Pipe Unit[0] = _Permute SUB    from 25
. . . . . .
. . . . . .
clock_cycles = 254200 PC = 73    Pipe Unit[0] = _Permute SUB    from 25
clock_cycles = 254300 PC = 89    Pipe Unit[0] =          ST     from 34
clock_cycles = 254400 PC = 124   Pipe Unit[0] =          LD     from 64
clock_cycles = 254500 PC = 89    Pipe Unit[0] =          xxx    from 0

Real Time Start =  Mon Nov.  8 12:30:45 1993
Real Time Stop  =  Mon Nov.  8 12:44:38 1993

CPU time for this run is 808650986 microseconds which is equivalent to
0 hours, 13 minutes, 28 seconds.
(If the run time for the program is more than 36 minutes the time displayed
 may not be accurate due to a limitation of the Unix implementation)

Total clock cycles used = 254538
```

Figure 18 Status Report for "perm" Program Run

At the end of the program run the real start and stop times are displayed, along with the amount of CPU time consumed. Finally, the total number of machine clock cycles used is shown. The user will probably want to confirm that the program has been correctly executed by examining the contents of the Integer and Boolean Registers which can be accessed by selecting option "R" from the Main Menu. A further menu of register display options is then presented, as shown in Figure 19.

```
                OPTION 'R' -   REGISTER FILE DISPLAY
THIS SCREEN ALLOWS THE SIX REGISTER FILES TO BE EXAMINED

     '1' -   DISPLAY CONTENTS OF MAIN INTEGER REGISTER FILE

     '2' -   DISPLAY CONTENTS OF NEW INTEGER REGISTER FILE

     '3' -   DISPLAY CONTENTS OF FORWARD INTEGER REGISTER FILE

     '4' -   DISPLAY CONTENTS OF MAIN BOOLEAN REGISTER FILE

     '5' -   DISPLAY CONTENTS OF NEW BOOLEAN REGISTER FILE

     '6' -   DISPLAY CONTENTS OF FORWARD BOOLEAN REGISTER FILE

     '8' -   DISPLAY CONTENTS OF MAIN INTEGER REGISTER FILE (IN HEX)

PRESS "M" TO RETURN TO PREVIOUS MENU    SELECTION = ?
```

Figure 19 Register Contents Option

The contents of the main Integer Registers provide the information required to verify the execution of the example benchmark program "perm", as shown in Figure 20.

```
          CONTENTS OF THE MAIN INTEGER REGISTER FILE
   (REGISTERS THAT CONTAIN INVALID DATA ARE MARKED WITH AN 'X')

R0  = 0                  R1  = 0                  R2  = 0
R3  = 0                  R4  = 0                  R5  = 8660
R6  = 4112               R7  = 0                  R8  = 6
R9  = 0                  R10 = 0                  R11 = 0
R12 = 0                  R13 = 0                  R14 = 0
R15 = 0                  R16 = 0                  R17 = 0
R18 = 0                  R19 = 0                  R20 = 0
R21 = 0                  R22 = 0                  R23 = 0
R24 = 0                  R25 = 0                  R26 = 0
R27 = 0                  R28 = 3                  R29 = 0
R30 = 4096               R31 = 4096               R32 = 0

SPECIAL PURPOSE REGISTERS MAPPED INTO INTEGER REGISTERS AS FOLLOWS:

  SR = R32     GP = R31     SP = R30     SP' = R29     RA = R28
```

Figure 20 Integer Register Contents for the "perm" Program

The Simulator can produce a very detailed breakdown of the simulation run just completed and this is accessed by selecting option "U" from page 2 of the Main Menu. The report shows how the specified machine resources have been utilised, including an indication of the number of cycles lost due to a resource restraint. This information is followed by a total count of clock cycles and instructions executed during the program run. The report concludes with a detailed analysis of each family of Functional Units showing the distribution of instructions amongst the available resources. The full usage report for the "perm" program is shown in Figure 21.

```
USAGE OF FUNCTIONAL UNITS (PIPES = 8) (MAX USED = 8) (MAX NEEDED = 4)
NUMBER OF CYCLES WHEN PIPED ALU (8) REQUEST DENIED = 0
NUMBER OF CYCLES WHEN PIPED MEMORY LOAD  (2) REQUEST DENIED = 17320
NUMBER OF CYCLES WHEN PIPED MEMORY STORE (2) REQUEST DENIED = 17321
NUMBER OF CYCLES WHEN PIPED BOOLEAN (4) REQUEST DENIED = 0
NUMBER OF CYCLES WHEN PIPED SHIFT (2) REQUEST DENIED = 0
NUMBER OF CYCLES WHEN BRANCH (2) REQUEST DENIED = 0
NUMBER OF CYCLES WHEN PIPED MULT (2) REQUEST DENIED = 0
NUMBER OF TIMES THAT BUFFER COULD NOT SUPPLY  A VALID INSTRUCTION = 87896
TOTAL CYCLES USED = 254538   TOTAL INSTRUCTIONS DISPATCHED = 338293

INDIVIDUAL TYPES OF INSTRUCTION DISPATCHED TO FUNCTIONAL UNITS:
ARITH  = 113242 (33%)  of which  DIVIDE = 0
MEM_LD =  72120 (21%)  MEM_ST =  72127 (21%)
BOOL   =  17328 ( 5%)  SHIFT  =   8666 ( 3%)  BRANCH =  54809 (16%)
MULT   =      0 ( 0%)  NOPS   =      0 ( 0%)
TOTAL INSTRUCTIONS FULLY EXECUTED = 338293    NOPS = 0
INDIVIDUAL TYPES OF INSTRUCTION FULLY EXECUTED BY FUNCTIONAL UNITS:
ARITH  = 113242 (33%)  of which DIVIDE = 0
MEM_LD =  72120 (21%)  MEM_ST = 72127 (21%)
BOOL   =  17328  (5%)  SHIFT  = 8666   (3%) BRANCH  =  54809 (16%)
MULT   =      0  (0%)  NOPS   =    0   (0%) BRN.TKN =  43947 (80% OF BRNS)
BSR    =  18741 (34%)  MOV    = 18740 (34%)  COND. BRANCHES =  17328 (32%)

           PIPELINED ARITHMETIC FUNCTIONAL UNIT USAGE
[0 ] = 99534 (88%) [1 ] = 13707 (12%) [2 ] = 1  ( 0%) [3 ] = 0  ( 0%)
[4 ] = 0     ( 0%) [5 ] = 0     ( 0%) [6 ] = 0  ( 0%) [7 ] = 0  ( 0%)
           PIPELINED MEMORY LOAD FUNCTIONAL UNIT USAGE
[0 ] = 44721 (62%) [1 ] = 27399 (38%)
           PIPELINED MEMORY STORE FUNCTIONAL UNIT USAGE
[0 ] = 44728 (62%) [1 ] = 27399 (38%)
           PIPELINED BOOLEAN FUNCTIONAL UNIT USAGE
[0 ] = 17328 (100%) [1 ] = 0  ( 0%) [2 ] = 0  ( 0%) [3 ] = 0  ( 0%)
           PIPELINED SHIFT FUNCTIONAL UNIT USAGE
[0 ] = 8666  (100%) [1 ] = 0  ( 0%)
           BRANCH FUNCTIONAL UNIT USAGE
[0 ] = 54808 (100%) [1 ] = 1  ( 0%)
           PIPELINED MULTIPLY FUNCTIONAL UNIT USAGE
[0 ] = 0     ( 0%) [1 ] = 0  ( 0%)

PIPE UNIT USAGE AT ID STAGE (TOTAL NUMBER OF PIPE-CYCLES USED = 540857)
[0] = 166642 (31%) [1] = 131985 (24%) [2] = 90871 (17%) [3] = 61262 (11%)
[4] = 37485  ( 7%) [5] = 28824  ( 5%) [6] = 18744 ( 3%) [7] = 5044  ( 1%)
PIPE UNIT USAGE FOR DISPATCH (TOTAL NUMBER OF PIPE-CYCLES USED = 338302)
[0] = 166642 (49%) [1] = 116858 (35%) [2] = 44724 (13%) [3] = 10078 (3%)
[4] =     0 ( 0%)  [5] =     0 ( 0%)  [6] =     0 ( 0%) [7] =     0 ( 0%)
RESULT BUS USAGE FOR THE 24 BUSSES AVAILABLE
[0 ] = 116862 (60%) [1 ] = 63466 (33%) [2 ] = 13700 ( 7%) [3 ] =    0 ( 0%)
[4 ] =      0 ( 0%) [5 ] =     0 ( 0%) [6 ] =     0 ( 0%) [7 ] =    0 ( 0%)
[8 ] =      0 ( 0%) [9 ] =     0 ( 0%) [10] =     0 ( 0%) [11] =    0 ( 0%)
[12] =      0 ( 0%) [13] =     0 ( 0%) [14] =     0 ( 0%) [15] =    0 ( 0%)
[16] =      0 ( 0%) [17] =     0 ( 0%) [18] =     0 ( 0%) [19] =    0 ( 0%)
[20] =      0 ( 0%) [21] =     0 ( 0%) [22] =     0 ( 0%) [23] =    0 ( 0%)
NUMBER OF TIMES RESULT BUS REQUEST WAS DENIED = 0
NUMBER OF INSTRUCTIONS SQUASHED IN BUFFER PRIOR TO ID STAGE = 0
NUMBER OF DATA CACHE WORDS USED BY PROGRAM = 46
```

Figure 21 Usage Report for the "perm" Program

The information contained in the Usage Report can help optimise the selection of suitable parameter values for the Simulator model to obtain the best performance from a limited set of machine resources. For some programs it may be necessary to examine the contents of the Data Cache, which can be achieved by selecting option "D" from the Main Menu. The menu for the Data Cache option is shown in Figure 22.

```
        DISPLAY DATA CACHE OPTION

DCACHE VALID ADDRESS RANGE IS 0 TO 40000

'1' -          DISPLAY FROM DCACHE BASE ADDRESS

'2' -          SPECIFY START ADDRESS FOR DISPLAY

'3' -          DISPLAY AMOUNT OF MEMORY ALLOCATED FOR DCACHE

'4' -          DISPLAY AMOUNT OF CORE LEFT

'M' -          RETURN TO MAIN MENU
```

Figure 22 Data Cache Option

The Data Cache can be displayed, one screen-full at a time, from the Data Cache base address or from any chosen address within the range given. The first screen page showing the contents of the Data Cache after running the "perm" program is illustrated in Figure 23.

```
                    *** CONTENTS OF DATA CACHE ***

NUMBER OF DATA CACHE WORDS USED BY PROGRAM = 46
(NO DATA WILL BE DISPLAYED IF NO MEMORY IS ALLOCATED FOR A DCACHE LOCATION)

 736    2944  novar        54          (HEX= 00000036) ASCII = ""
 738    2952  novar        4112        (HEX= 00001010) ASCII = ""
 739    2956  novar        1           (HEX= 00000001) ASCII = ""
 740    2960  novar        4116        (HEX= 00001014) ASCII = ""
 741    2964  novar        2           (HEX= 00000002) ASCII = ""
 768    3072  novar        54          (HEX= 00000036) ASCII = ""
 770    3080  novar        4112        (HEX= 00001010) ASCII = ""
 771    3084  novar        1           (HEX= 00000001) ASCII = ""
 772    3088  novar        4120        (HEX= 00001018) ASCII = ""
 773    3092  novar        3           (HEX= 00000003) ASCII = ""
 800    3200  novar        54          (HEX= 00000036) ASCII = ""
 802    3208  novar        4112        (HEX= 00001010) ASCII = ""
 803    3212  novar        1           (HEX= 00000001) ASCII = ""
 804    3216  novar        4124        (HEX= 0000101c) ASCII = ""
 805    3220  novar        4           (HEX= 00000004) ASCII = ""
 832    3328  novar        54          (HEX= 00000036) ASCII = ""
 834    3336  novar        4112        (HEX= 00001010) ASCII = ""

more to come.... press 'C' to continue or 'Q' to quit
```

Figure 23 Data Cache Contents for the "perm" Program

The Data Cache is represented by an array of data records, each record holding a data label and a 32-bit value. The Data Cache display shows the array line numbers followed by actual memory addresses then the data label, if any. The data value is expressed as an integer value, in decimal and hexadecimal, and is also interpreted as a group of four ascii characters to aid readability of stored text strings.

While the program is being run, counters associated with each of the source program instruction lines are incremented every time that an instruction is successfully issued and this information is stored in special fields of the data structure representing the Instruction Cache. The Instruction Cache can be viewed by selecting option "I" from the Main Menu, the data for the program "perm" being shown in Figure 24.

```
                    *** CONTENTS OF INSTRUCTION CACHE ***

 0     1                                    MOV  GP,#4096
 1     1                                    MOV  SP,#4096
 2     1                                    BSR  RA,_main (0){87}
 3     0 _dummy                             SUB  SP,SP,#128
 4     0                                    ADD  SP,SP,#128
 5     0                                    MOV  PC,RA(0)
 6 10078 _Swap                              SUB  SP,SP,#128
 7 10078                                    LD   R7,(R0,R5)
 8 10078                                    LD   R8,(R0,R6)
 9 10078                                    ST   (R0,R5),R8
10 10078                                    ST   (R0,R6),R7
11 10078                                    ADD  SP,SP,#128
12 10078                                    MOV  PC,RA(0)
13     1 _Initialize                        SUB  SP,SP,#128
14     1                                    MOV  R7,#1
15     1                                    ADD  R8,GP,#12
16     7 L6                                 ASL  R5,R7,#2
17     7                                    ADD  R5,R8,R5
18     7                                    ADD  R6,R7,#-1
19     7                                    ST   (R0,R5),R6
```

Figure 24 Instruction Cache after a Run of the "perm" Program

Analysis of the dynamic execution counts for different instructions can help give a profile of the program behaviour so that scheduling attention can be focussed on optimising portions of code that account for the bulk of the execution time. This point is well illustrated in Figure 24 where the procedure "_Swap" is heavily used, when compared with the surrounding code.

The user may wish to edit one or more instructions while they are in the Instruction Cache to modify a program before carrying out another simulation run. This can be achieved by selecting option "O" from the Main Menu which prompts the user to enter the Instruction Cache word address of the candidate instruction, as illustrated in Figure 25.

```
                    ONE INSTRUCTION RECORD OPTION

    THIS SCREEN ALLOWS YOU TO EDIT ONE INSTRUCTION IN THE iCACHE

    ENTER THE iCACHE ADDRESS FOR THE INSTRUCTION RECORD

                        SELECTED ADDRESS = 3

SEARCH ADDRESS = 3,  iCACHE ADDRESS = 3,  LABEL = "_dummy"

Selected instruction is:

_dummy                                          SUB  SP,SP,#128

Memory Address = 3  Label = "_dummy"

                                Boolean Mask
B0  B1  B2  B3  B4  B5  B6  B7  B8  B9 B10 B11 B12 B13 B14 B15
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
op code =  "SUB"
Instruction Type = "ALU"
Destination Register = "SP"
Second Destination Register = "XX"
Source Register 1 Label = " "
Source Register 1 = "SP"  Source Register 2 = "128"
Source Register 3 = "XX"  Carry Register    = "XX"
Operation for third operand = ' '
ASL value = 0    ASR value = 0
Branch Boolean Register  = "XX" Branch Integer Register = "XX"
Branch Target Address    =  0   Branch Target Label     = " "
Branch Dependent Count   =  0
Squash Bit = 0      Speculative = 0      Usage Counter = 0

DO YOU WISH TO EDIT THIS INSTRUCTION RECORD?  (Y/N)
```

Figure 25 Edit Instruction Option

Once the user has specified the address of the instruction to be edited, the instruction is shown in normal assembler form followed by the internal data structure representation used by the Simulator. If the user wishes to edit the current instruction, a prompt is given to type in the new version using normal assembler mnemonics. This results in the instruction being echoed back to the user after verification, along with the modified contents of the internal data structure representation, as shown in Figure 26.

```
CURRENT RECORD AT iCACHE ADDRESS 3 IS

_dummy                                          SUB  SP,SP,#128

TYPE IN THE NEW INSTRUCTION

_dummy                                SUB SP,SP,#256


new text entered is :  _dummy                                  SUB SP,SP,#256

THE NEW INSTRUCTION RECORD IS

_dummy                                          SUB  SP,SP,#256

Memory Address = 0  Label = "_dummy"

                              Boolean Mask
B0  B1  B2  B3  B4  B5  B6  B7  B8  B9 B10 B11 B12 B13 B14 B15
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
op code =  "SUB"
Instruction Type = "ALU"
Destination Register = "SP"
Second Destination Register = "XX"
Source Register 1 Label = " "
Source Register 1 = "SP"  Source Register 2 = "256"
Source Register 3 = "XX"  Carry Register    = "XX"
Operation for third operand = ' '
ASL value = 0    ASR value = 0
Branch Boolean Register  = "XX" Branch Integer Register = "XX"
Branch Target Address    =  0   Branch Target Label     = " "
Branch Dependent Count   =  0
Squash Bit = 0      Speculative = 0      Usage Counter = 0

DO YOU WISH TO STORE THIS NEW INSTRUCTION?  (Y/N)Y

EDIT NEXT ADDRESS?  (Y/N)   N

     PRESS 'C' TO CONTINUE IN THIS OPTION OR 'Q' TO QUIT ....
```

Figure 26 Edited Instruction

If this new version of the instruction is acceptable to the user it can be committed to the Instruction Cache and an opportunity is given to edit the next sequential instruction. If changes are made to the Instruction Cache in this way it is advisable to re-format the Instruction Cache by selecting option "F" from the Main Menu.

It may be useful for the user to be reminded of the HSP instruction set when editing the Instruction Cache so an option is available, via "L" on Main Menu page 2, to display the mnemonic assembler forms of all the instructions supported by the Simulator. The resulting data is shown in Figure 27.

```
                    *** INSTRUCTION SET ***

ADD     ADDV    SUB     SUBV    ADDC    SUBC    DIV     DIVV    MOD     GTS
GES     LTS     LES     GTU     GEU     LTU     LEU     EQ      NE      ASL
ASLV    ASR     LSR     AND     OR      EOR     EXT     BIC     MULT    MULTV
LD      LDB     ST      STB     AND     OR      EQ      NE      GT      LT
LE      GE      MOV     BT      BF      BSR     BRA     EI      DI      TRAP
BRK     BEQ     BNE     BGTS    BLTS    BLES    BGES

                **** END OF INSTRUCTION SET ****
```

Figure 27 Instruction Set for the Simulator

As can be seen in Figure 27, only a limited sub-set of the HSP instructions are supported by the Simulator at present. In particular, no attempt has been made to deal with floating point instructions although structures to handle these will be added in the future.

Another facility offered by the Simulator is the ability to read a specified file from the host computer, which can be useful if the original source program file contains comments which are stripped out when the program is loaded into the Instruction Cache. Any file can be viewed by selecting the "V" option from the Main Menu then entering the path and filename required.

The Simulator's dynamic tracing facility is best explained by following an example, which again uses the "perm" source program. Use the Select Files option "S" from the Main Menu to reload the "perm" source code into the Instruction Cache as before, then choose option "A" to activate the program. This time we will answer "yes" to the prompt that asks if a trace is required and a trace start address of 32 is specified, as shown in the recorded interaction of Figure 28.

```
                    ACTIVATE PROGRAM OPTION

THIS ALLOWS YOU TO SPECIFY THE START ADDRESS OF THE PROGRAM

DO YOU WANT TO USE AN ICACHE LABEL AS A START ADDRESS?  (Y/N)  n

START ADDRESS MUST BE BETWEEN 0 AND 93          START ADDRESS = 0

DO YOU WANT TO DE-ALLOCATE MEMORY FROM THE ENTIRE DCACHE ? (Y/N)   y

DO YOU WANT TO RE-INITIALISE ALL REGISTER FILES ? (Y/N)   y

DO YOU WANT TO SET MAIN REGISTER FILES "ACTIVE" ? (Y/N)   y

DO YOU WANT TO SET MAIN REGISTER FILES "VALID" ? (Y/N)   y

DO YOU WANT TO TRACE THE PROGRAM'S EXECUTION ? (Y/N)   y

DO YOU WANT TO SPECIFY TRACE START ICACHE ADDRESS ? (Y/N)   y

ENTER THE ICACHE START ADDRESS FOR TRACE
(SELECTED INSTRUCTION WILL BE IN EX STAGE)

START ADDRESS = 32

DO YOU WANT TO REPORT ERRORS ?   (Y/N)   y

DO YOU WANT TO STOP ON ERRORS ?  (Y/N)   y

DO YOU WANT TO TRACE AFTER ERROR STOP ?  (Y/N)   y
```

Figure 28 Activate Option with Trace Selected

The Simulator will go into trace mode as soon as the specified instruction from Instruction Cache address 32 is detected at the ID stage of processing. The Simulator will then display the contents of the dispatch "pipes", the Fetch Unit and the Instruction Buffer at the end of clock cycle 69. For each clock cycle thereafter, the trace option causes the following sequence of information to be displayed. The prospective Instruction Decode group is shown, subject only to data dependency checks between its members, followed by messages from the various machine procedures that issue instructions and process the Functional Units. At the end of the cycle the modified Instruction Decode group is shown, along with the latest contents of the Fetch Unit Pipeline and the Instruction Buffer. The trace sequence for clock cycle 70 of the "perm" program is shown in Figure 29.

```
data dep. between pipe_unit[5](0) R5 and pipe_unit[4] dest1 = R5
data dependency found between ADD instruction in Pipe 5
and LD instruction in Pipe 4
common register is "R5"

Pipes in use at Allocate Pipe Unit stage = 5

                CONTENTS OF THE 8 (32) PIPE UNITS

pipe_unit[7 ] slot= 0  instr=      xxx 0   occ= 0 disp= 0 inht= 0 b_tkn= 0
pipe_unit[6 ] slot= 0  instr=      xxx 0   occ= 0 disp= 0 inht= 0 b_tkn= 0
pipe_unit[5 ] slot= 5  instr=      ADD 33  occ= 1 disp= 0 inht= 0 b_tkn= 0
pipe_unit[4 ] slot= 4  instr=       LD 32  occ= 1 disp= 1 inht= 0 b_tkn= 0
pipe_unit[3 ] slot= 3  instr=      MOV 31  occ= 1 disp= 1 inht= 0 b_tkn= 0
pipe_unit[2 ] slot= 2  instr=       ST 30  occ= 1 disp= 1 inht= 0 b_tkn= 0
pipe_unit[1 ] slot= 1  instr=       ST 29  occ= 1 disp= 1 inht= 0 b_tkn= 0
pipe_unit[0 ] slot= 0  instr=       ST 28  occ= 1 disp= 1 inht= 0 b_tkn= 0

updated prohibit mask = "00XX000000000000" in SquashInBuffer1
Conditional Execution procedure just completed.  "execute" = 1

the effective address is 3724, valid1 = 1, valid2 =1
Instruction from Pipe Unit[0] successfully dispatched to Functional Unit.
Buffer Element 0 has been squashed in SquashBufferElement
Conditional Execution procedure just completed.  "execute" = 1

the effective address is 3728, valid1 = 1, valid2 =1
Instruction from Pipe Unit[1] successfully dispatched to Functional Unit.
Buffer Element 1 has been squashed in SquashBufferElement
Conditional Execution procedure just completed.  "execute" = 1
warning!  unable to process instruction (op code = ST) in Pipe 2 due to
lack of available functional unit
Unable to dispatch instruction from Pipe Unit[2].  Pipe Unit inhibited.
Pipe Unit[3] is marked as Inhibited due to active "stop_dispatch"
Pipe Unit[4] is marked as Inhibited due to active "stop_dispatch"
Pipes that successfully dispatched their instructions = 2
0 instructions squashed in the buffer by SquashInBuffer2
unable to load 16 new instructions into buffer with only 5 vacant positions
Buffer is full in FetchToBuffer.  Memory Fetch pipeline will stall unless
there are empty rows that can be over-written
the priority list for branch_units is empty
unable to put new instructions in buffer

end of cycle 70

Unit contents at end of clock cycle 70 are......

                CONTENTS OF THE 8 (32) PIPE UNITS

pipe_unit[7 ] slot= 0  instr=      xxx 0   occ= 0 disp= 0 inht= 0 b_tkn= 0
pipe_unit[6 ] slot= 0  instr=      xxx 0   occ= 0 disp= 0 inht= 0 b_tkn= 0
pipe_unit[5 ] slot= 5  instr=      ADD 33  occ= 1 disp= 0 inht= 0 b_tkn= 0
pipe_unit[4 ] slot= 4  instr=       LD 32  occ= 1 disp= 1 inht= 1 b_tkn= 0
pipe_unit[3 ] slot= 3  instr=      MOV 31  occ= 1 disp= 1 inht= 1 b_tkn= 0
pipe_unit[2 ] slot= 2  instr=       ST 30  occ= 1 disp= 1 inht= 1 b_tkn= 0
pipe_unit[1 ] slot= 1  instr=       ST 29  occ= 1 disp= 1 inht= 0 b_tkn= 0
pipe_unit[0 ] slot= 0  instr=       ST 28  occ= 1 disp= 1 inht= 0 b_tkn= 0
```

```
                    CONTENTS OF THE FETCH UNIT ARE
INSTRUCTIONS ARE FETCHED FROM PC ADDRESS IN ICACHE INTO ELEMENTS[1][0 - 15]
INSTRUCTIONS ARE COPIED TO THE INSTRUCTION BUFFER FROM  ELEMENTS[0][0 - 15]

F[1][15] (1)                                          ST   0(SP),RA
F[1][14] (1) _main                                    SUB  SP,SP,#128
F[1][13] (1)                                          MOV  PC,RA(0)
F[1][12] (1)                                          ADD  SP,SP,#128
F[1][11] (1)                                          LD   RA,0(SP)
F[1][10] (1) L19                                      LD   R17,8(SP)
F[1][9 ] (1)                                          BSR  RA,_dummy (0){3}
F[1][8 ] (1)                                          MOV  R5,#0
F[1][7 ] (1)                                          BT   B6,L19(0){83}
F[1][6 ] (1)                                          EQ   B6,R5,#8660
F[1][5 ] (1)                                          LD   R5,8(GP)
F[1][4 ] (1)                                          BT   B5,L18(0){72}
F[1][3 ] (1)                                          LES  B5,R17,#1
F[1][2 ] (1)                                          ADD  R17,R17,#1
F[1][1 ] (1)                                          BSR  RA,_Permute (0){25}
F[1][0 ] (1)                                          MOV  R5,#7

F[0][15] (1) L18                                    BSR  RA,_Initialize (0){13}
F[0][14] (1)                                          MOV  R17,#1
F[0][13] (1)                                          ST   8(GP),R0
F[0][12] (1)                                          ST   8(SP),R17
F[0][11] (1)                                          ST   0(SP),RA
F[0][10] (1) _Perm                                    SUB  SP,SP,#128
F[0][9 ] (1)                                          MOV  PC,RA(0)
F[0][8 ] (1)                                          ADD  SP,SP,#128
F[0][7 ] (1)                                          LD   RA,0(SP)
F[0][6 ] (1)                                          LD   R20,20(SP)
F[0][5 ] (1)                                          LD   R19,16(SP)
F[0][4 ] (1)                                          LD   R18,12(SP)
F[0][3 ] (1) L8                                       LD   R17,8(SP)
F[0][2 ] (1)                                          BT   B4,L12(0){46}
F[0][1 ] (1)                                          GTS  B4,R18,#0
F[0][0 ] (1)                                          ADD  R18,R18,#-1

                    CONTENTS OF THE INSTRUCTION BUFFER
END POSITION (EMPTY) = 27  CURRENT POSITION = 0
CURRENT BUFFER LENGTH = 27  GREATEST BUFFER LENGTH = 31
CURRENT BUFFER OCCUPANCY = 27  GREATEST BUFFER OCCUPANCY = 31
NUMBER OF TIMES BUFFER COULD NOT SUPPLY PIPES A VALID INSTRUCTION = 24

E  O S M  A
L  C Q K  D                          INSTRUCTION
E  C H D  R

26 1 0 0 56                                           BSR  RA,_Swap (0){6}
25 1 0 0 55                                           MOV  R6,R17
24 1 0 0 54                                           MOV  R5,R19
23 1 0 0 53                                           BSR  RA,_Permute (0){25}
22 1 0 0 52                                           ADD  R5,R20,#-1
21 1 0 0 51                                           BSR  RA,_Swap (0){6}
20 1 0 0 50                                           MOV  R6,R17
19 1 0 0 49                                           MOV  R5,R19
18 1 0 0 48                                           ADD  R17,R5,R7
17 1 0 0 47                                           ADD  R7,GP,#12
16 1 0 0 46 L12                                       ASL  R5,R18,#2
15 1 0 0 45                                           ADD  R19,R5,R7
14 1 0 0 44                                           ADD  R7,GP,#12
13 1 0 0 43                                           ASL  R5,R20,#2
12 1 0 0 42                                           BT   B3,L8(0){60}
11 1 0 0 41                                           LES  B3,R18,#0
```

```
10 1 0 0 40
9  1 0 0 39
8  1 0 0 38
7  1 0 0 37
6  1 0 0 36
5  1 0 0 35
4  1 0 0 34
3  1 0 0 33
2  1 0 0 32
1  1 0 0 31
0  1 0 0 30
```

```
MOV  R18,R17
BSR  RA,_Permute (0){25}
MOV  R5,R17
ADD  R17,R20,#-1
BT   B2,L8(0){60}
EQ   B2,R20,#1
ST   8(GP),R5
ADD  R5,R5,#1
LD   R5,8(GP)
MOV  R20,R5
ST   20(SP),R20
```

Figure 29 Dynamic Trace for One Cycle of the "perm" Program Run

The Simulator performed this program run using the Working Model selected earlier in the Program Parameters option and Figure 29 reports information for the 8 "pipes" units, the two-cycle Fetch Unit Pipeline with 16 elements per stage and the 32-word Instruction Buffer. These dynamic trace facilities enable the working of the Simulator to be followed cycle-by-cycle to investigate possible errors in scheduled code but they still may not be detailed enough to find the cause of a problem. For this reason, an extended trace report can be generated by selecting the "Full Report" option via the Set Variable menus in the Program Parameters section. The Full Report option causes the trace report to show additional data during each clock cycle on the current state of each Functional Unit, which results in several more screens of data being displayed.

Once the trace mode has been exited, the user may wish to examine the current state of a particular group of Functional Units. To view a unit select the "U" option from the Main Menu page 1 which leads to the menus for non-pipelined and pipelined versions of Functional Units, as shown in Figures 30 and 31.

```
        DISPLAY STATUS OF FUNCTIONAL UNITS OPTION

    '1' -   DISPLAY STATUS OF ARITHMETIC UNITS
    '2' -   DISPLAY STATUS OF MEMORY STORE UNITS
    '3' -   DISPLAY STATUS OF MEMORY LOAD UNITS
    '4' -   DISPLAY STATUS OF BOOLEAN UNITS
    '5' -   DISPLAY STATUS OF SHIFT UNITS
    '6' -   DISPLAY STATUS OF BRANCH UNITS
    '7' -   DISPLAY STATUS OF MULT UNITS
    '8' -   DISPLAY STATUS OF PIPELINED FUNCTIONAL UNITS
PRESS "M" TO RETURN TO PREVIOUS MENU    SELECTION = ?
```

Figure 30 Menu for Displaying Non-Pipelined Functional Units

```
        DISPLAY STATUS OF PIPELINED FUNCTIONAL UNITS

    '1' -   DISPLAY STATUS OF PIPELINED ARITHMETIC UNITS
    '2' -   DISPLAY STATUS OF PIPELINED MEMORY STORE UNITS
    '3' -   DISPLAY STATUS OF PIPELINED MEMORY LOAD UNITS
    '4' -   DISPLAY STATUS OF PIPELINED BOOLEAN UNITS
    '5' -   DISPLAY STATUS OF PIPELINED SHIFT UNITS
    '7' -   DISPLAY STATUS OF PIPELINED MULT UNITS
    '8' -   DISPLAY STATUS OF PIPE UNITS
PRESS "M" TO RETURN TO PREVIOUS MENU    SELECTION = ?
```

Figure 31 Menu for Displaying Pipelined Functional Units

A typical display for the pipelined Memory Store Functional Units is shown in Figure 32, representing part of the machine state at the end of clock cycle 70 of the "perm" program simulation run.

```
STATUS OF THE 2 PIPELINED MEMORY STORE UNITS (NORMAL LATENCY = 1)

                    PIPELINED MEMORY STORE UNIT (0)

STAGE NUM.   AVAIL. INHIBIT  BUSY_CYCLES  DEST.   VALID  VALUE      (HEX)

     1        0       0         1        3724      1      0          0
     0        1       0         0        3712      1      75         4b

                    PIPELINED MEMORY STORE UNIT (1)

STAGE NUM.   AVAIL. INHIBIT  BUSY_CYCLES  DEST.   VALID  VALUE      (HEX)

     1        0       0         1        3728      1      0          0
     0        1       0         0        3720      1      1          1

    Press 'Enter' to continue.......
```

Figure 32 Contents of the Memory Store Functional Units

Other options are available that allow the contents of the "pipes" and the Instruction Buffer to be examined by the user. Facilities exist that enable statistical data relating to a particular program run to be stored in a file of the user's choice. Access to this option is achieved by pressing "W" from Main Menu page 2 which causes the menu in Figure 33 to appear on the screen. The user can enter a filename, or just press "enter" to designate the file name "usage".

```
Enter the name of the file to be written to
or just press 'Enter' to select the default file 'usage'

     File name =

File name = "usage"

 THIS SCREEN ALLOWS SIMULATOR DATA TO BE STORED IN THE SPECIFIED FILE

         '1' -   APPEND PARAMETERS

         '2' -   APPEND INTEGER REGISTERS

         '3' -   APPEND STATISTICAL DATA

         '4' -   APPEND ICACHE LISTING

         '5' -   APPEND DCACHE LISTING

         '8' -   GENERATE SHORT REPORT (NO CACHES)

         '9' -   GENERATE FULL REPORT  (NO DCACHE)

    PRESS "M" TO RETURN TO PREVIOUS MENU    SELECTION = ?
```

Figure 33 Report Generation Option

The menu options allow elements of statistical data to be appended to the selected file, or to choose one of two predefined report formats that are commonly used. Once the usage file has been completed, it is useful to be able to print out a hard copy version on the laser printer and this is achieved by selecting option "P" from Main Menu page 2. The user is then prompted for a filename, or to accept the default file name "usage", as shown in Figure 34. This print option makes use of the "mpage" document formatting facility offered by the host unix environment.

```
Enter the name of the file to be printed out
or just press 'Enter' to select the default file 'usage'

      File name =

The contents of the "usage"
will be printed out

command issued to system = "/usr/5bin/pr -o5 usage | mpage -Plw -A -W90 -1"

[mpage: 4 pages, printer lw]
```

Figure 34 File Print Option

It is hoped that the information contained in this user manual is sufficient to enable a novice user to become familiar with the operation of the Superscalar Simulator program, but if any questions arise please contact the author for further explanation.

Although the range of options offered by the Simulator may seem daunting at first, they have been added to the program one by one in response to actual needs when debugging the hand-scheduled source program code. It is hoped that these options will prove very useful during the development of the Instruction Scheduler program that is to take place next.

Index of Figures