TECHNICAL REPORT

COMPUTER SCIENCE

An introduction to the Hatfield Superscalar Scheduler

Technical Report No 316

Fleur L Steven

Spring 1998

Abstract

This document presents a comprehensive overview of the Hatfield Superscalar Scheduler (HSS). It concentrates on the concepts involved rather than on the detailed coding because the scheduler is in a state of evolution and is constantly under review. The main features of the scheduler, its associated data structures, and the parameters that define and control the scheduler are fully described. A complete list of the parameters is also found in the appendix. This document is intended to be a guide for anyone with an interest in scheduling, or for anyone who wants to know how to use HSS.

Contents

List of Figures

1. Introduction

The Hatfield Superscalar Scheduler (HSS) is being developed as part of the wider Hatfield Superscalar Architecture (HSA) project currently being undertaken at the University of Hertfordshire (Steven et al, 1997). The long term objective of the project is to achieve an order of magnitude performance increase compared with a classic RISC processor while avoiding the often associated increase in code size. A suite of software has been written to support this work including HSS, an HSA gnu 'C' compiler, an HSA simulator (Collins, 1993, Collins, 1995), and a Trace Driven Simulator (Potter, 1996). The significant features of HSA are guarded instruction execution (Steven & Collins, 1996) and a delayed branch mechanism (Collins & Steven, 1994, Egan, 1997). Other features include in-order instruction issue and out-of-order completion, and support for speculative execution. Unlike many other superscalar projects the architecture is minimal in terms of the hardware help provided to issue instructions in parallel at run time and instead relies heavily on an instruction scheduler to reorder the original sequential code into instruction groups which can be executed in parallel. Thus the instruction scheduler is the key for achieving the high performance desired.

2. The Hatfield Superscalar Architecture

The Hatfield Superscalar Architecture (HSA) was developed as a vehicle for instruction scheduling research. It has been described as a minimal superscalar architecture (Steven & Collins, 1996) since it embodies a hybrid technology which combines the best features of VLIW and superscalar concepts.

HSA is a load and store architecture with a RISC instruction set (Appendix A) derived from its predecessor HARP (Steven, Steven & Wang, 1995). Separate integer and Boolean register files are provided. The latter are used both to store branch conditions and to implement guarded instruction execution. A simple four stage pipeline is used:

IF	Instruction Fetch
ID	Instruction Decode
EX	Execute
WB	Write Back

During the IF stage, multiple instructions are fetched into an Instruction Buffer from the instruction cache. Instructions are then issued in order from the Instruction Buffer to functional units during the ID stage. Conditional branches are also resolved in the ID stage.

While more complete details of the architecture can be found elsewhere (Steven et al, 1997), two further features are directly relevant to this paper. First, all HSA instructions, including branches, are conditionally executed. The following divide instruction, for example, will only be executed if the value in Boolean register seven evaluates to false (or zero) at run time:

```
FB7 DIV R1, R6, R13
```

Second, the processor attempts to avoid issuing instructions from the Instruction Buffer if the associated guard condition has already failed. Instructions are therefore marked as "squashed" in the instruction buffer if they have remained unissued in the buffer for a full cycle and the associated Boolean condition evaluates to false. To avoid increased pressure on the processor cycle time, the IF stage evaluates squashing conditions in parallel with its primary instruction issue function.

To achieve high performance, HSS reorders the HSA assembly language code to form groups of instructions that can be issued to functional units in parallel at run time. These instruction groups are then presented to the processor as traditional sequential code. The ID stage has the task of reconstructing the original instruction groups, before issuing them in parallel for execution. This task involves checking that each instruction being issued does not use the result of an instruction being issued ahead of it in the same group. Since each

pair of instructions being issued must be compared for dependencies, the complexity of this dependency checking increases in proportion to the square of the issue rate and places increasing pressure on the ID stage cycle time.

In the Instruction Decode stage, the processor simply rebuilds instruction groups that have already been assembled at compile time. We are therefore now investigating the idea of marking the end of each instruction group within the instruction stream. This involves using only a single bit in each instruction to flag the end of each parallel group. Issuing instructions then simply involves scanning through the instruction buffer looking for the first end of group flag. No dependence checks are required between instructions. Instruction issue is subject only to functional unit availability and operands being available from previously issued instructions. The instruction group flags therefore effectively encapsulate information about compile-time instruction groups, yet do not sacrifice compatibility over a range of processor designs.

3. Data Dependencies

Scheduling instructions for parallel execution can be viewed as a process in which each instruction is successively moved or percolated (Nicolau, 1985) up through the code structure in an attempt to ensure that it is executed at the earliest possible opportunity. This code motion is ultimately stopped by data dependencies between pairs of instructions.

Three classes of data dependencies can be identified: Read after write (RAW), write after read (WAR) and write after write (WAW). However, only RAW dependencies represent true data dependencies and therefore ultimately limit the performance of MII processors. In contrast, WAR and WAW data dependencies can both be removed by using register renaming.

In the instruction sequence below instruction I2 has a WAR or anti-dependence on I1.

```
I1      ADD R5, R6, R7          /*R5 := R6 + R7*/
I2      ADD R6, R8, #256        /*R6 := R8 + 256*/
```

This dependence can be removed by returning the result of I2 to an unallocated register, in this case R20. This renaming allows I2 to be percolated ahead of I1 in the instruction schedule:

```
I2      ADD R20, R8, #256       /*R20 := R8 + 256*/
I1      ADD R5, R6, R7          /*R5 := R6 + R7*/
 ..
        MOV R6, R20             /*R6 := R20*/
```

The move instruction is required to restore the new result to R6. Note this extra instruction need not introduce further data dependencies. Subsequent instructions using R6 can equally well use R20.

Register renaming can also be used to remove spurious data dependencies which arise when code is moved between basic blocks. Consider the following example:

```
NE B6, R1, R2       /* B6 := (R1 <> R2)*/
BT B6, Label        /* if B6 is true goto Label*/

Label:
LD R6, 8(SP)        /*R6 := contents of (SP + 8)*/
```

The LD instruction could be moved ahead of the branch instruction and executed speculatively[1] giving the following code:

```
NE B6, R1, R2       /*B6 := (R1 <> R2)*/
```

[1] An instruction is executed speculatively if it is executed before it is known whether the path originally containing the instruction will actually be taken.

```
    LD R6, 8(SP)        /*R6 := contents of (SP + 8)*/
    BT B6, Label        /*if B6 is true goto Label*/

Label:
```

Unfortunately if R6 is live on the alternative path it will be incorrectly updated whenever the branch fails. Register renaming can be used to avoid this problem:

```
    NE B6, R1, R6       /*B6 := (R1 <> R6)*/
    LD R20, 8(SP)       /* R6 replaced by R20*/
    BT B6, Label        /*if B6 is true goto Label*/

Label:
    MOV R6, R20         /*R6 := R20*/
```

As before, a MOV instruction is required to copy the contents of R20 into R6 if the branch is taken.

An alternative solution is to use guarded instruction execution. On HSA any of the Boolean registers which are used to record the results of relational instructions can also be used as Boolean guards. In the above example B6 can therefore be used to guard the execution of the LD instruction:

```
      NE B6, R1, R2     /*B6 := (R1 <> R2)*/
T B6  LD R6, 8(SP)      /*if B6 is true execute LD*/
      BT B6, Label      /*if B6 is true goto Label*/

Label:
```

Now the Boolean guard ensures that the LD will only be executed if the branch is taken.

Any further code motion will move the LD instruction beyond the scope of the Boolean guard. Now only register renaming can be used:

```
LD R20, 8(SP)      /*R6 replaced by R20*/
NE B6, R1, R6      /*B6 := (R1 <> R6)*/
BT B6, Label       /*if B6 is true goto Label*/

Label:
MOV R6, R20        /*R6 := R20*/
```

The above code illustrates a further problem introduced by the speculative execution of instructions. Suppose the load instruction in the previous example generates an invalid memory reference. If the path originally containing the load instruction is not actually followed, the instruction will generate a spurious exception which will incorrectly terminate the program.

To solve this problem, all non-branch instructions must exist in two forms. In the normal form, any exception generated by an instruction is immediately taken. In the second speculative form an exception will simply generate a polluted value in its result register. For example, consider the code below:

```
BT B6, Label       /*if B6 is true goto Label*/
LD R6, 8(SP)       /*R6 := contents of (SP + 8)*/
SUB R8, R6, #1     /*R8 := R6 - 1*/
NE B3, R8, #0      /*B3 := (R8 <> 0)*/

Label:
```

Now assume that both the load and subtract instructions are scheduled speculatively ahead of the branch instruction:

```
LD! R6, 8(SP)       ; speculative load
SUB! R8, R6, #1     ; speculative subtract
BT B6, Label
NE B3, R8, #0

Label:
```

If the load instruction generates an exception, R6 will be marked as polluted. Since the subtract instruction is also marked as speculative, it will in turn mark R8 as polluted when it finds R6 is polluted. An exception will only be taken when the non-speculative relational instruction attempts to use the polluted value held in R8. Note this is the earliest point in the code where we can be certain that the speculative load should have been executed.

To support speculative execution an extra bit must be added to all processor registers, including the Boolean registers, to flag polluted values. This additional hardware support allows loads and other instructions, such as adds which generate an exception on overflow, to be executed speculatively. However, in the absence of additional hardware support, store instructions can still not be executed speculatively. In HSA, stores can therefore only be safely percolated into a preceding basic block if they can be guarded.

4. Guards

One of HSA's major features is an ability to execute all instructions conditionally. The instructions are conditionally executed by placing a Boolean guard in front of the instruction. The condition tested is denoted by T (true) or F (false). For example:

```
T B1 ADD R6, R7, R8
```

will only execute if the value in B1, at run time, is true. Likewise:

```
F B6 MULT R5, R4, #4
```

will only execute if the value in B6 is false. The instructions can also be guarded by more than one Boolean guard. The maximum number of guards

which can be appended to a non-branch or non-move instruction is controlled by a parameter "MAX_GUARDS" in the file **hsp_const.h**. Branch instructions have their own parameter "MAX_BR_GUARDS" to determine the number of guards available to them. Likewise MOV instructions have their own parameter "MAX_MOV_GUARDS" which also determine the number of guards available to them. Thus several guards up to this predefined number can control instructions as follows:

```
T B2 F B5 F B7 BT B1, Loop
```

Here the BT (branch if true) instruction will only execute if B2 is true, B5 is false, and B7 is false. It will then only branch to Loop if B1 is true.

By setting the appropriate parameters to zero, conditional execution can also be switched off. In these circumstances the scheduler will make no attempt to attach new guards to instructions.

Guarded instructions have three major advantages. Firstly, renaming is avoided when an instruction is percolated into a preceding basic block. If we consider the fragment of code below:

```
GT B2, R2, R3
BT B2, Label
MULT R6, R8, R9
```

The MULT instruction can always be moved in parallel with the branch instruction as long as the guard condition T B2 is attached:

```
GT B2, R2, R3
BT B2, Label;  T B2 MULT R6, R8, R9
```

If guarded execution is switched off R6 must be renamed if the contents of R6 is live on entry to the basic block starting at Label. Therefore, conditional execution reduces the number of registers required by the code and removes the need for the extra MOV instruction introduced by the renaming process.

Secondly, conditional execution can result in the deletion of short basic blocks. For example, typical code for an "if then else" statement is shown below:

```
                NEQ B6, R14, R15        /*B6 = (R14 <> R15)*/
                BF B6, else_code        /*Branch if B6 = false*/
then_code:      ADD R1, R2, R3
                BRA continue            /*Unconditional branch*/
else_code:      SUB R1, R2, R3
continue:
```

After scheduling the following code may be produced:

```
                NEQ B6, R14, R15
                T B6 ADD R1, R2, R3;  F B6 SUB R1, R2, R3
```

Thus, three basic blocks have been reduced to one, thereby reducing code size and reducing the number of branch execution units required.

Thirdly, branch instructions themselves can be moved into a branch delay slot of an earlier branch:

```
                BT B1, Label
                .
                .
                BT B2, Label2
```

becomes:

```
                BT B1, Label;  F B1 BT B2, Label2
```

There are however, two main disadvantages of conditional execution. Firstly, extra encoding space is required in each instruction word to specify the condition. Secondly, guards significantly increase the complexity of the instruction scheduler. Instruction scheduling ultimately reduces to a massive case analysis problem and our experience suggests that using Boolean guards introduces a significant number of additional hard-to-handle cases.

5. Comparison with Related Work

This section attempts to put the HSS scheduling algorithm into context and compares it with other work being carried out in the USA.

Trace Scheduling (Fisher, 1981), which is perhaps the best known instruction scheduling technique, was developed by Fisher at Yale in conjunction with a VLIW architecture. Central to this technique is the concept of a trace which is a path through a sequence of basic blocks that is frequently executed. Traces are selected and scheduled in order of their frequency of execution. The selected trace is scheduled as if it were a single basic block. Code is then inserted at path exits and entrances to preserve the semantics of the off-trace state. The process is repeated until all the code has been scheduled. However, code from successive traces is never overlapped by the scheduler and therefore successive iterations of a loop will never be overlapped to achieve software pipelining. This problem can be addressed by unrolling loop bodies to provide longer traces and then allowing the unrolled loop bodies to be scheduled for parallel execution. However, loop unrolling is an excellent mechanism for achieving code explosion.

At Hertfordshire, we feel that code compatibility and code expansion problems make Fisher's VLIW architecture unsuitable for general-purpose computation. Fortunately, the instruction scheduling techniques pioneered by Fisher can be equally well applied to superscalar processors. Only slight changes are required to preserve the instruction-level semantics and to ensure that the resultant code can also be executed in the traditional sequential style.

Current scheduling developments try to achieve software pipelining and build on either Modulo Scheduling Techniques (Rau and Fisher, 1993, Rau, 1994) or on the Enhanced Percolation Scheduling algorithm (Ebcioglu, 1994) developed by Kemal Ebcioglu's group at IBM. Software pipelining is a method of overlapping operations from different loop iterations, without initially unrolling the loop, thereby attempting to produce a minimum number of cycles between successive loop iterations.

Modulo Scheduling attempts to construct a schedule based on a fixed initiation interval (II). The II is the delay between the beginning of successive iterations

of a loop. Modulo Scheduling computes a lower bound on the II. The minimum II may be determined by loop carried dependencies. Loop carried dependencies are dependencies between instructions in different iterations of a loop. In Fig 5.1 a circular chain of dependencies ensures that II must be at least four. In general an instruction in loop iteration 'n' may depend on an earlier instruction in loop iteration 'n-i', or by available resources.

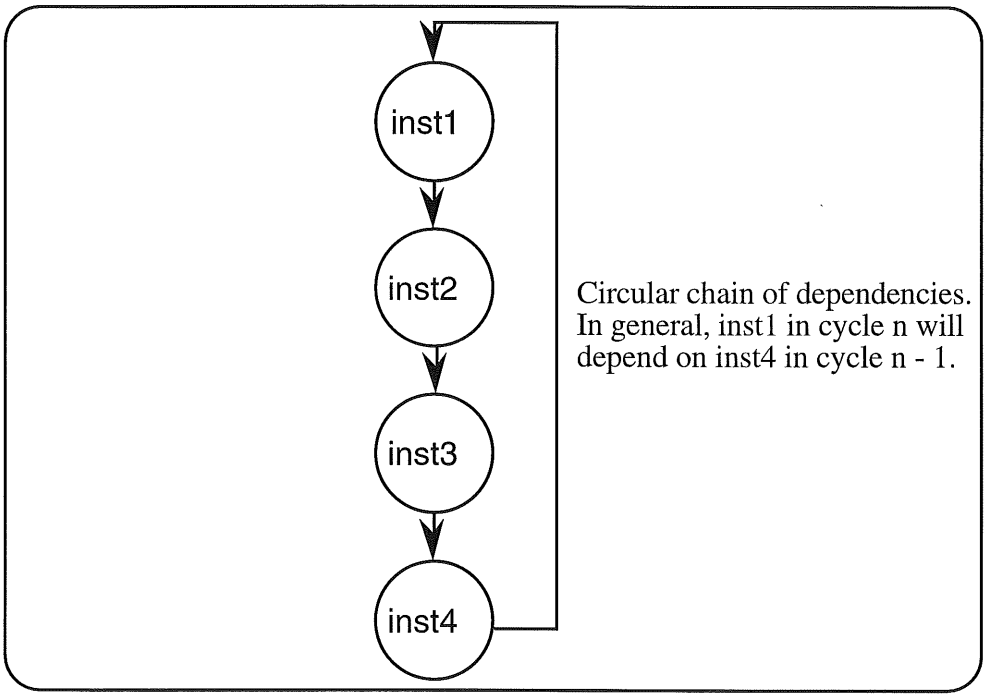

Fig 5.1 Loop Carried Dependencies

The II may also be limited by the number of instructions that can be issued in each cycle. For example, if there are five arithmetic instructions in the loop and only two ALU operations can be issued in each cycle, the II must be at least three. The loop schedule is then obtained by placing instructions in a window of appropriate size for the minimum II. A reservation table is used to record resources currently in use by a particular instruction. Instructions can be "unscheduled" because of resource conflicts and then rescheduled later in the window as part of a backtracking scheduling algorithm. When an instruction is "unscheduled" the resource information pertaining to that instruction is removed from the window. If a schedule cannot be obtained for a given II, the II is incremented and scheduling is attempted again. This

process is repeated until a satisfactory schedule is obtained. A loop prelude and postlude are added after the schedule. This basic algorithm only works for single basic blocks. The main challenge in using Modulo Scheduling is to extend the technique to loop structures of arbitrary complexity.

One technique that has been proposed is to use guarded execution to convert multiple basic blocks into a single basic block before scheduling. This procedure is called if-conversion (Warter et al, 1993). If guarded execution is not provided the process must then be reversed using reverse if-conversion (loc. cit). Unfortunately, the resultant scheduled code is far from optimum.

In contrast to Modulo Scheduling, Enhanced Percolation Scheduling keeps the body of the loop intact throughout the scheduling process. During scheduling an instruction group, termed the fence, is selected and searches are then made through the code for instructions which will coexist within the selected group. This continues until the group is full or until no further instructions can be found. All successor groups of the fence are then selected as the new fences and searches are again made for instructions to fill the groups. Instructions in already filled fences are now allowed to migrate upwards past the join point of the loop, both around the backedge and into a loop prelude. To facilitate this code motion, two copies of each filled fence are created, one on the path entering the loop and one at the end of the loop (Fig 5.2).

The copy of the filled fence at the end of the loop represents operations from the next iteration of the loop. The whole process is repeated until all the instructions in the loop have been moved into fences. A major challenge with this technique is to avoid the excessive code expansion caused by the aggressive fence duplication.

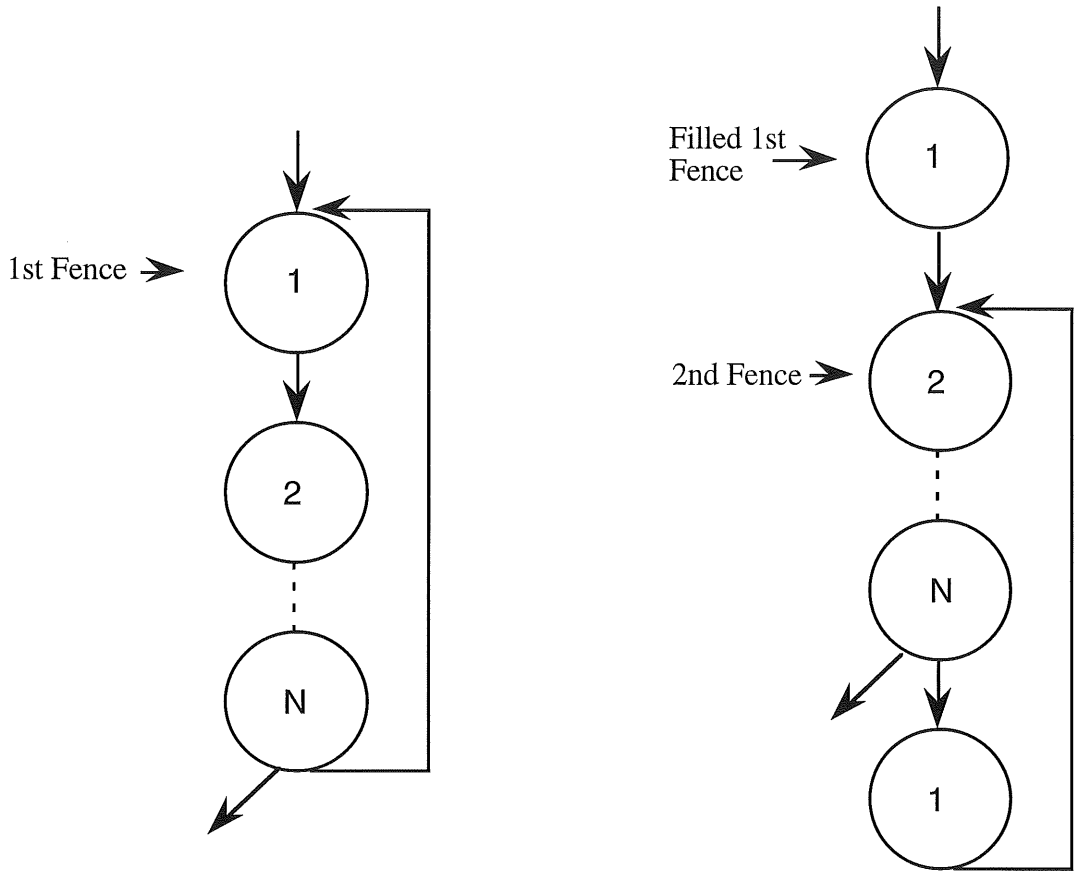

Fig 5.2 Ebcioglu's Enhanced Percolation Scheduling

HSS builds on the IBM work and can therefore schedule loops of any complexity. However, our scheduler differs in two important respects. Firstly, each individual instruction is scheduled in turn and percolated up through the code structure as far as possible. In contrast, the IBM algorithm assembles each parallel instruction group in turn by repeatedly searching forward through the code for candidate instructions. Our approach avoids repeatedly picking up candidate instructions only to find that they can not be moved far enough to be included in the group currently being assembled.

Secondly, code motion across loop back edges is restricted to avoid excessive code expansion. As in the IBM algorithm, code is systematically moved across loop back edges to overlap successive loop iterations and achieve software pipelining. However, code is only moved across a loop back edge if it can be demonstrated that the instruction being moved is part of a critical path dependence chain that determines the iteration time of at least one path

through the loop.

6. Introduction to HSS

HSS has been developed to improve performance through static instruction scheduling. The long term aim is to achieve an order of magnitude performance increase over conventional RISC processors.

HSS must gather information about a particular benchmark and place the benchmark's assembly instructions in appropriate data structures before any instruction scheduling can take place. The scheduler must therefore first read in the instructions, form one-word parallel instruction groups (LIW), detect basic blocks, find branch targets, detect procedures, detect loops[2], and compute register live ranges. In addition there is an option for inlining functions which can be invoked after the live range analysis has been completed. If the inlining option is invoked, the loop detection and live range information must then be updated before scheduling can commence.

The program is scheduled a procedure at a time. Because programs spend a great deal of time in loops and provide much potential parallelism, innermost loops are scheduled first, followed by outer loops and finally by straight line code.

Once scheduling has been completed, the final scheduled code is output by the functions "PrintParInstructionRecord" and "PrintSeqInstructionRecord" found in the file **hsp_sched.c**. "PrintSeqInstructionRecord" outputs the scheduled code in a form suitable for the HSP simulator. The resulting scheduled code is then used by the HSP simulator to produce statistics about the scheduled code. The final schedule will comprise many instructions in parallel and the resulting code will execute more quickly than the unscheduled code.

HSS is available on Sun Unix workstations and PCs under Linix, and is very

[2] Nested loops which share the same header are merged before scheduling.

straightforward to use. It takes as input the HSA assembly code which has been produced by the HSA Gnucc 'C' Compiler and outputs four other files with various extensions. These files must be specified by the user as follows:

% hsp	<file.s>	<file.u>	<file.ps>	<file.ins>	<file.stat>
	input	output	output	output	output

If the correct number of files is not specified HSS will output an error message conveying the correct usage and will then exit.

The output files all serve different purposes as follows:

The lines of assembly code in **file.u** are numbered to make it easier to see which instructions have been moved up during scheduling. In **file.ps** the scheduled code is output as parallel instruction groups for viewing by the user. In **file.ins** the same code is output with only one instruction per line in a format suitable for the HSA Simulator. Finally, **file.stat** contains statistics about the code gathered during instruction scheduling such as the number of instructions contained in the program both before and after scheduling.

HSS is configured by changing parameters in the file **hsp_const.h**. There are two types of configuration parameter. The first type selects the target machine model. For example, the number of arithmetic units available is determined by assigning a value to the parameter "ARITH_UNIT". The second type selects the various scheduling options available. For example, the decision whether to percolate code into a basic block ending in a BSR is determined by assigning the value YES or NO to the parameter "PERCOLATEINTOBSR". Full details and an explanation of these parameters can be found in Appendix B.

7. The HSS Scheduling Mechanism

This section is concerned with the scheduling process and examines both local

and global percolation. Firstly, it presents an overview of the scheduling process. Secondly, it goes on to describe two major processes involved in instruction percolation, that is, checking to determine whether instructions can coexist within instructions groups and if so whether they can pass these instructions to move into the next instruction group. Thirdly, the percolation of a single instruction both at the local level and at the global level is outlined. Finally, an example is presented showing the addition of guards to a percolating instruction from a sequential successor basic block and a branch target basic block.

7.1 Overview of HSS

The HSP GNU CC Compiler was generated using GNU CC (Stallman, 1989). The benchmarks compiled with this compiler can then be presented to HSS for scheduling.

Fig 7.1 shows a structural overview of the HSS scheduler. For simplicity, only certain of the modules are shown in the diagram. The code presented for scheduling can be optionally inlined (indicated by the shaded inlining box). Inlining is fully described in Section 15.

The HSS Backedge Algorithm is at the highest level of this system and incorporates both local and global percolation. Local percolation takes place first and then calls global percolation to percolate an instruction into another basic block. Within percolation, dependencies must be checked and this involves both a check to see whether an instruction can coexist with another instruction, that is, be executed in the same cycle as another instruction and also whether an instruction can pass another instruction, that is, be executed before another instruction. CheckCoexist also checks for any instruction merging (Section 12) that is possible. This is one technique for removing the problems associated with true data dependencies. During CheckPass, the memory disambiguation function (Section 14) is invoked in order to ascertain whether load and store instructions access the same memory locations.

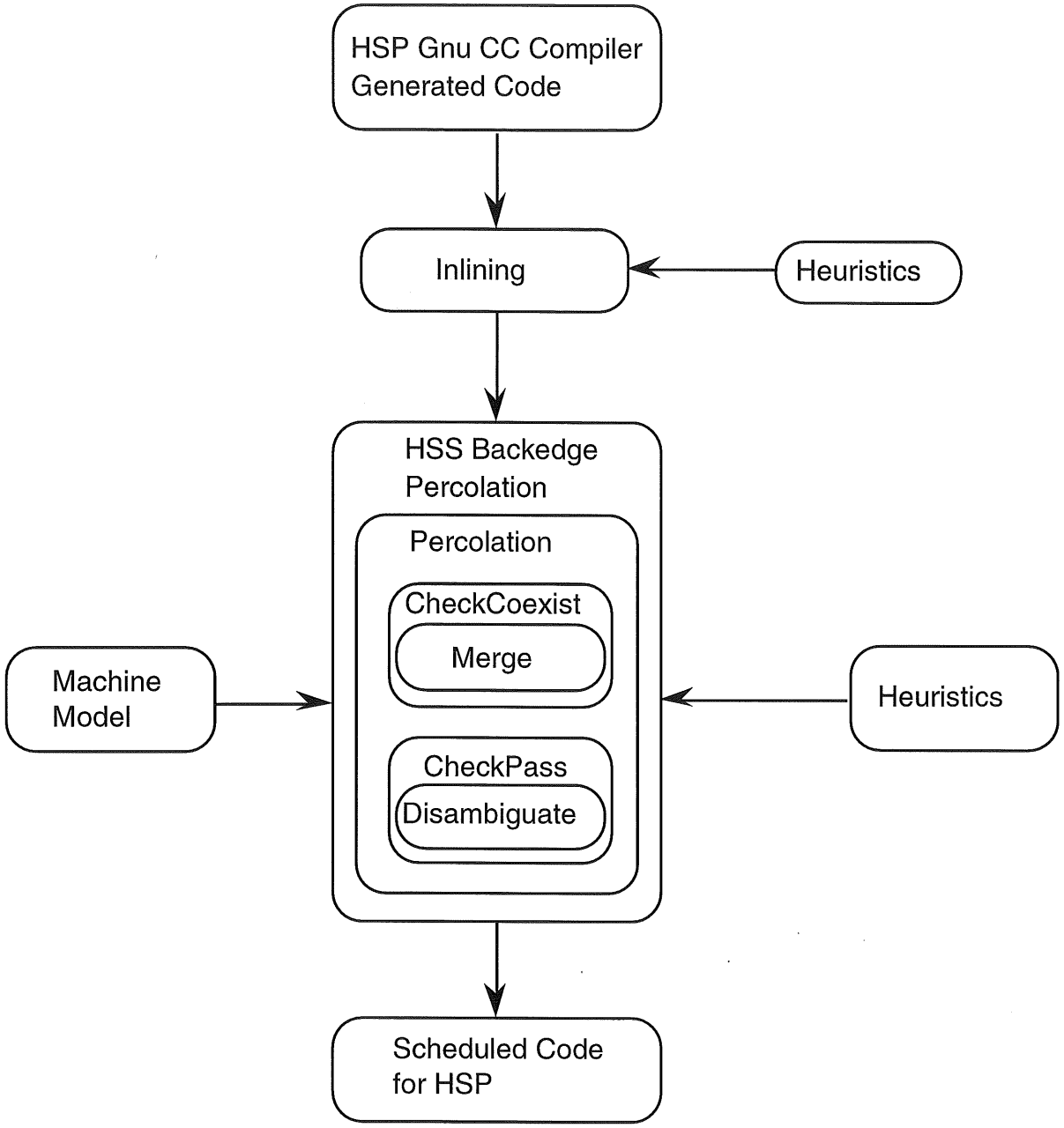

Fig 7.1 Structure of HSS

7.2 Coexistence and Passing Within Instruction Groups

During the scheduling process, whether a percolating instruction can coexist with or pass other instructions already within an instruction group is checked and determined by code contained in files **hsp_checkcoexist.c** and **hsp_checkpass.c**

7.2.1 Checking for Instruction Coexistence

File **hsp_checkcoexist.c** contains a number of functions which compare pairs of instructions for coexistence. Each of the functions deals with a particular pair of instructions according to their types. For example function "CoexistTypes1_1" compares two instructions both of which must have a value between one and four inclusive in their type fields. These values indicate that each instruction is one of four types: arithmetic, logical, multiplication or relational. In contrast to this, function "CoexistTypes6_5" compares two instructions, the first of which must be of type 6 which indicates a store instruction, and the second or percolating instruction must be of type 5 which indicates a load instruction. There are 16 such functions.

A check for coexistence is invoked during the percolating process. A percolating instruction can coexist within an instruction group if there are no true data dependencies between it and the instructions already in the group, and if there are also enough resources available in terms of functional units. Consider two instructions, inst1 already in an instruction group and a percolating instruction, inst2. If an anti-dependence occurs between the two there is no problem because the percolating instruction is added by default to the end of the instruction group. Where possible, output dependencies are dealt with by deleting inst1. Alternatively, if inst2 is guarded it may not always be executed whenever inst1 is executed. In this case it is not safe to delete inst1 and the output of inst2 is renamed. If a true data dependence occurs between the percolating instruction and an instruction in the group, it may be possible to remove it either through merging (Section 12) or combining (Section 13). However, for the moment we can assume that a true dependence halts the percolation process.

7.2.2 Checking for Instruction Passing

If a percolating instruction can coexist within an instruction group the percolation process will then call the function "CheckPass" in the file **hsp_checkpass.c** to determine whether the percolating instruction can move

past all the other instructions in the group. If there is an anti-dependence between an instruction in the group and the percolating instruction then the output register of the percolating instruction must be renamed before it can pass that instruction. In the case of instruction pairs with references to memory locations, memory disambiguation is invoked (Section 14), and a check is made to compare the addresses of the two instructions. If the addresses are different then the percolating instruction can move past the first instruction, otherwise it cannot.

7.3 Percolating Individual Instructions

When considering the percolation of an individual instruction its movement both within its initial basic block and its subsequent movement beyond the initial basic block must be examined. In general, an instruction may be able to move up several different paths during percolation. The scheduler even has to cope with multiple copies of a percolating instruction which may enter the same basic block from several different paths. Thus many problems have to be resolved to maintain the semantic validity of the program.

7.3.1 Instruction Percolation Within a Basic Block

The following fragments of code illustrate the process of calling "CheckCoexist" and "CheckPass" discussed in Sections 7.2.1 and 7.2.2.

The following example shows that although inst3 can pass all the instructions in LIW2 it has a true data dependence with the LD instruction in LIW1 and cannot percolate any further.

```
LIW0:   SUB R2, R3, #4
LIW1:   LD R1, (R0, R5);
LIW2:   MOV R7, R2;  ADD R2, R3, R4
inst3:  ADD R5, R1, R6    /*percolating instruction*/
```

inst3 can coexist within the LIW2 group as shown below:

LIW0: SUB R2, R3, #4
LIW1: LD R1, (R0, R5);
LIW2: MOV R7, R2; ADD R2, R3, R4; **ADD R5, R1, R6**
inst3:

In the following example, the LD instruction in LIW1 has been modified by changing the destination register R1 to R3 The percolating instruction can now move further up, as there is no longer a true data dependence between the LD instruction and the percolating instruction. However, there is still an anti-dependence between the two instructions, and the destination register of the percolating instruction will have to be renamed to pass the LD, as shown below. If the renaming option is not enabled a FAIL is returned and the instruction cannot percolate any further.

LIW0: SUB R2, R3, #4; **ADD R10, R1, R6**
LIW1: LD R3, (R0, R5)
LIW2: MOV R7, R2; ADD R2, R3, R4
inst3: MOV R5, R10 /*MOV instruction introduced during renaming*/

7.3.2 Instruction Percolation Beyond a Basic Block

If an instruction reaches the top of its basic block, as is shown in the above example, it is then percolated into both the sequential predecessor and branch target predecessor basic blocks. The instruction must succeed in percolating into all the predecessor blocks to retain the semantic validity of the program otherwise a FAIL is returned. Percolation therefore, in general, involves inserting one or more versions of an instruction in new locations and deleting or removing the original version of the instruction from the schedule.

However, there have been previous versions of the scheduler where, if one of the basic block predecessors is a sequential predecessor and the instruction fails to percolate into it but succeeds in percolating into the basic block's other predecessors, then to retain the semantic validity of the program the basic

block that the percolating instruction started from is duplicated and inserted immediately after the sequential predecessor (Fig 7.2).

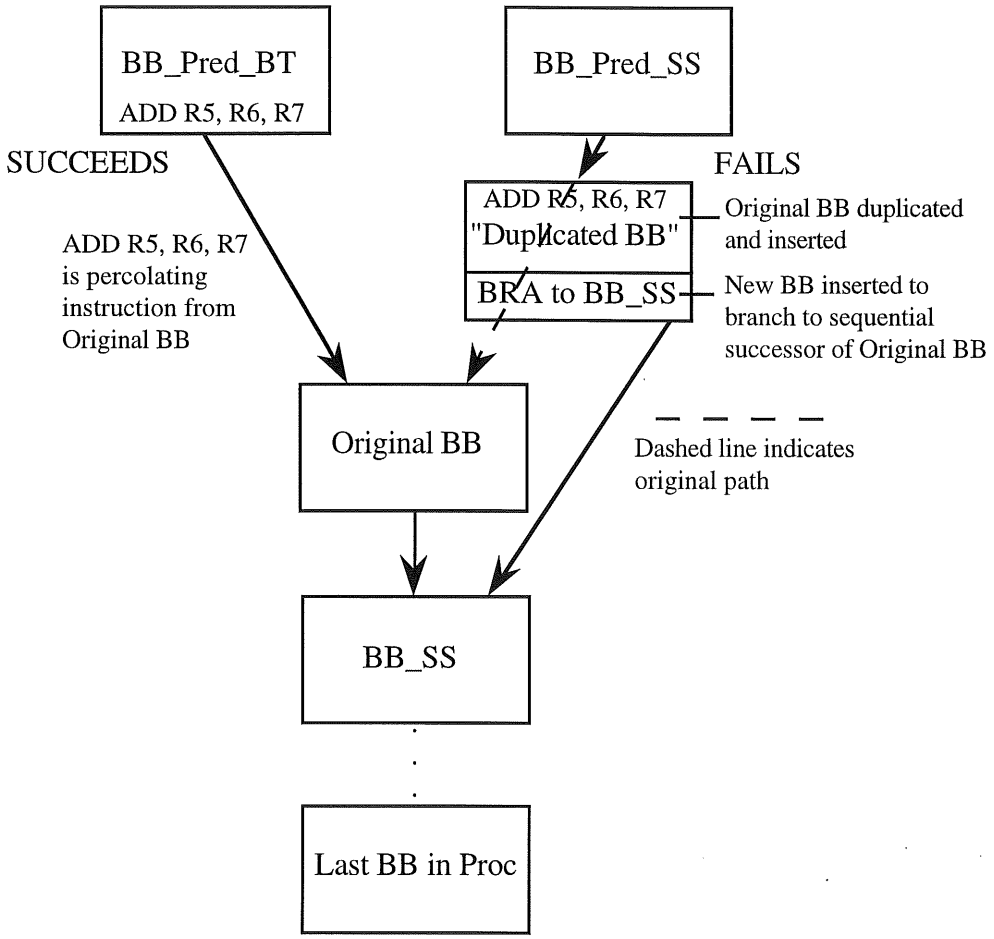

Fig 7.2 Percolating Instruction Failure in a Sequential Predecessor BB

Similarly, if one of the basic block predecessors is a branch target predecessor and the percolating instruction fails to percolate into it but succeeds in percolating into the basic block's other predecessors, the percolating instruction's original basic block is inserted at the end of the procedure and all branches are updated to retain the semantics of the program (Fig 7.3).

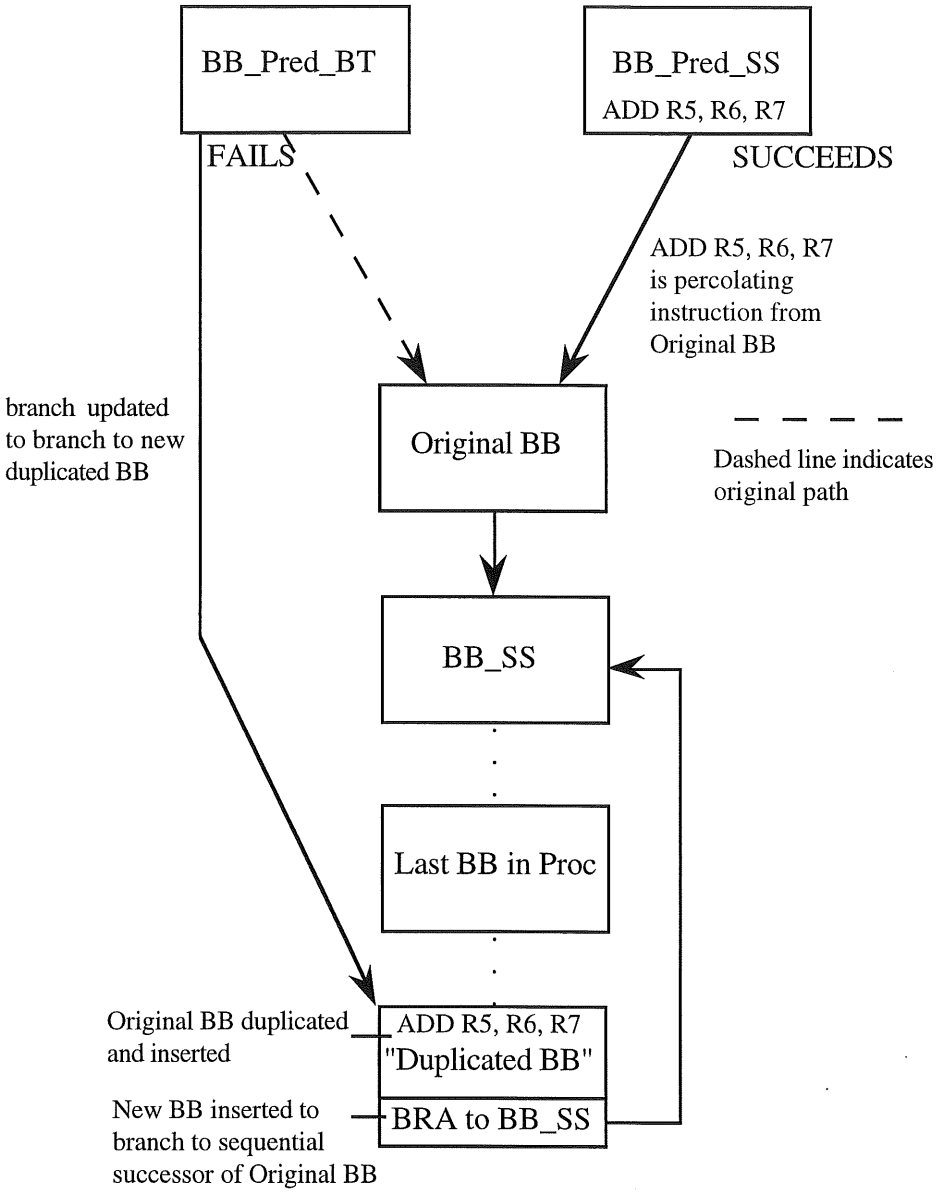

Fig 7.3 Percolating Instruction Failure in a Branch Target Predecessor BB

The problem of duplicate instructions was referred to at the start of this section and there are two major circumstances which will need to be resolved by the scheduler. Firstly, if an instruction moves up several different paths and meets itself in a basic block further up, it can only be successfully inserted into that block if any guards or other alterations acquired during percolation permit both copies of the instruction to be combined to form a single instruction; otherwise the second percolation to reach the basic block will fail. Secondly, if one copy of an instruction has already successfully percolated all the way through a basic block on one path, then a second copy of the

instruction following another path cannot be inserted into that block. It also has to percolate all the way through the block otherwise its percolation fails (Fig 7.4).

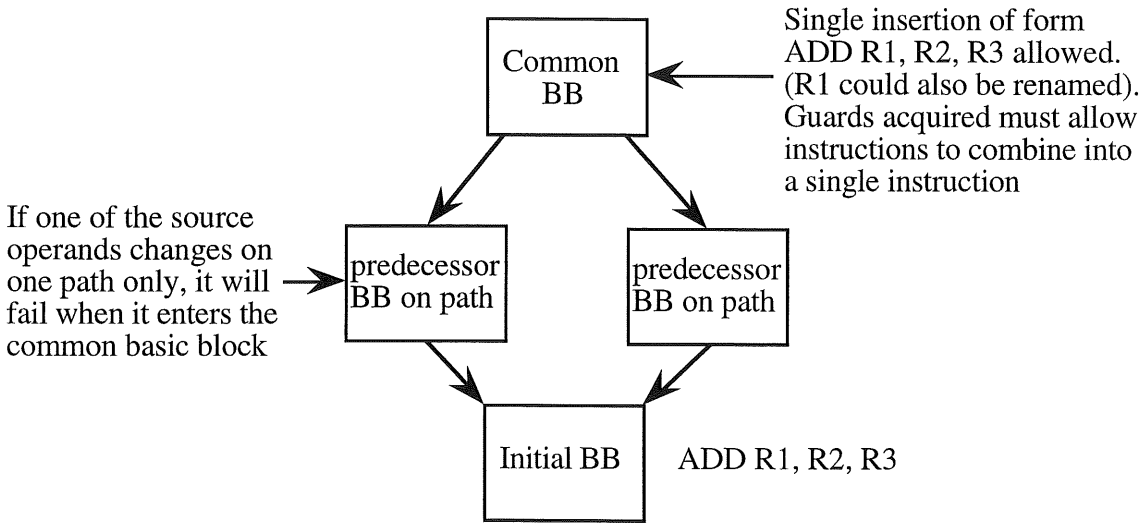

Fig 7.4 Conditions for Instruction Combining in a Common Basic Block

7.3.3 The Addition of Guards

On entry to a new basic block, a guard may be added to an instruction if the control flow from the block is determined by a conditional branch (BT or BF). Let us assume the branch is BT B1, Label. If the predecessor block is a sequential predecessor then the opposite guard, F B1, is added. If, on the other hand, the predecessor block is a branch target predecessor the same guard, T B1, is added. Any number of guards may be added up to a predefined number set by the user. As an instruction percolates through several basic blocks it may therefore acquire multiple guards. An example of instructions in sequential and branch target basic blocks percolating up is shown below:

```
NE B1, R11, R12
BT B1, L8
..
ADD R10, R6, #-1      /*sequential successor moving up*/
```

L8: MOV R5, #6 /*sequential branch target moving up*/

becomes:

```
NE B1, R11, R12
BT B1, L8
F B1 ADD R10, R6, #-1
T B1 MOV R5, #6
```

During percolation an instruction might acquire more than the maximum permitted number of guards. If the instruction is not a branch or a store then the guard which was acquired the earliest is removed and the destination register is renamed. In the case where the instruction is a branch or a store a different approach is required. Firstly, a branch instruction must never lose a guard. Secondly, a store instruction cannot be executed speculatively as to do so would irretrievably alter the state of the program. Therefore, in these two cases if the number of guards exceeds the permitted maximum, the percolation fails.

8. Dealing with Long Latency Instructions

Certain instructions require more than one cycle to execute and these are termed long latency instructions. The number of cycles a long latency instruction requires is defined by the user in file **hsp_const.h**. HSA simulations traditionally allow three cycles for a multiplication and 16 cycles for a divide while the number of cycles assigned to a load instruction directly depends on the number of cycles required to access the data cache. One cycle results in a load latency of one, while two cycles gives a load latency of two. When a long latency instruction is percolated up through the code a mechanism is required to deal with the extra cycles. For example, if a load instruction with a latency of two is moved up into a new instruction group, the following instruction group must not contain an instruction which uses the load instruction's destination register as a source operand. At run time, such an instruction would stall for one cycle while the load instruction completed

and therefore the schedule would not be optimal. The example below demonstrates the problem:

```
LIW1    ADD R1, R2, R3; LD R5, (R0, R6)
LIW2    SUB R8, R5, R4
```

The LD in LIW1 has R5 as the destination. The SUB in LIW2 will therefore stall waiting for the LD to complete. To obtain an optimal schedule, the scheduler must therefore insert an additional instruction group between LIW1 and LIW2.

HSS optimises its schedule by inserting placeholders, one for each cycle required to execute a long latency instruction, in the instruction groups immediately following the instruction group occupied by a long latency instruction. These placeholders are called virtual copies (VCOPYs) and take the form VCOPY Ri, Ri where Ri is the destination register of the long latency instruction. VCOPYs are just placeholders introduced during scheduling; they do not use any resources and are not included in the final output code that is executed by the simulator. They are merely there to enforce optimal scheduling. The above example is now transformed as shown below:

```
LIW1    ADD R1, R2, R3; LD R5; (R0, R6)
LIW2    VCOPY R5, R5
LIW3    SUB R8, R5, R4
```

The VCOPY has ensured that the SUB instruction cannot move into LIW2 as it has a true data dependence on register R5.

When VCOPYs were intially implemented in HSS, they were generated only once during the entire scheduling process for each long latency instruction. They were then treated in exactly the same way as any other instruction and were percolated up through the code in their turn. However, problems occurred when long latency instructions percolated up several different paths and were finally inserted in several different basic blocks. As there was only one initial generation of the VCOPYs, they could not be shared by all the

newly inserted long latency instructions. Furthermore, they tended to get separated from their long latency instructions during scheduling. This separation made renaming particularly difficult since the associated VCOPYs must also renamed. The implementation was changed and now whenever a long latency instruction is percolated and inserted into a group, the required number of VCOPYs is also generated and inserted into the following instruction groups. A distinct set of VCOPYs is therefore generated for each instantiation of the long latency instruction. This process involves retracing the percolation path, possibly through several basic blocks. Similarly, whenever a long latency instruction is deleted from the schedule, its associated VCOPYs are also deleted.

Fig 8.1 shows how the VCOPYs are inserted when a multiply instruction is successfully percolated into two basic blocks.

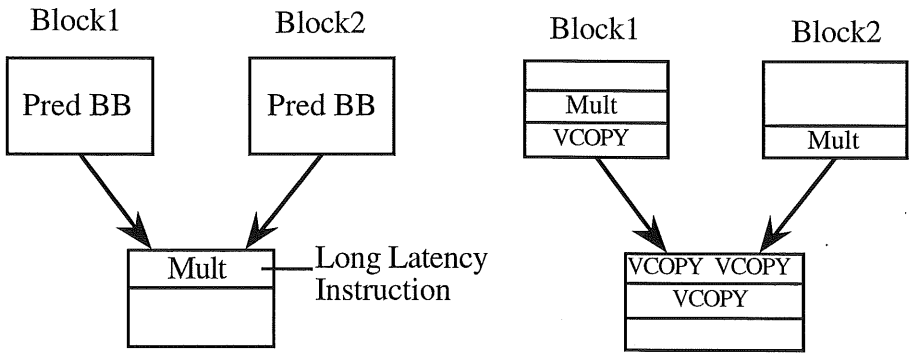

Fig 8.1 Insertion of VCOPYs after a Long Latency Instruction

A multiply instruction typically requires three cycles to execute and therefore requires two VCOPYs per instruction. In Block1 the multiply has percolated into the penultimate instruction group. In Block2 the multiply has percolated into the last instruction group. The VCOPYs are then inserted as follows: Block1 has one VCOPY in the last instruction group. The other VCOPY belonging to the multiply is inserted in the first instruction group of the successor block, that is, the block the multiply originally started from. In Block2 because the multiply is already in the last instruction group both VCOPYs have to be inserted in Block2's successor in the first and second

instruction groups If the long latency instruction is subsequently successfully repercolated, new VCOPYs will be inserted with every new instantiation of the multiply instruction, while the original VCOPYs will be deleted along with the earlier multiply instructions. To facilitate the deletion process each VCOPY's instruction node contains a pointer to the associated long latency instruction that caused it to be generated.

9. Delayed Branch Mechanism

HSA provides a delayed branch mechanism where the number of instructions to be executed after the branch is specified in the branch instruction as an immediate value. In the scheduler, however, it is more convenient to think in terms of the number of instruction groups, after the group containing a branch, that must be executed before the branch is taken. These groups are known as delay slots and their number is defined by the user.

```
LIW1    BT B1, Label (#3);  T B1 SUB R6, R11, #10
LIW2    T B1 ADD R5, R6, R7
LIW3    T B1 ASR R4, R5, #2
```

Section 2 described the HSA pipeline. A branch instruction is fetched in the IF stage and resolved in the ID stage (computing PC + offset, and testing the boolean register condition). There is therefore an inherent delay before an instruction fetch can be initiated from a branch target. If it takes one cycle to fetch an instruction from the ICACHE, and another cycle to resolve the branch, then the minimum number of delay slots required will always be equal to the number of fetch cycles in the ICACHE, that is, one. If the IF stage takes two cycles then the number of delay slots required will be two and so on.

The number of delay slots required after the branch is defined by the user by setting the parameter "ICACHE_CYCLES" in the file **hsp_const.h** to an appropriate integer value. "ICACHE_CYCLES", in turn, is used to give a value to the parameter "DELAY_SLOTS" which ultimately determines the number of slots inserted during scheduling. A slow icache is modelled by

setting the parameter to two and a fast icache is modelled by setting the parameter to one. The branch instruction then specifies the number of instructions to be executed after the branch instruction. This variable count is implemented by providing a count field in all branch instructions. Initially, the compiler sets all count fields to zero After instruction scheduling, HSS sets the count equal to the number of instructions successfully scheduled between the branch instruction and the end of the final delay slot group. For example, below is a fragment of code showing a branch instruction with two delay slots:

```
46    T B6 ADD R16, R19, #4; BT B6, L10 (#6); VCOPY R1, R1;
          VCOPY R7, R7; F B6 LD R17, 12 (SP); F B6 LD R18, 16 (SP);
              F B6 ADD R8, SP, #256
49    VCOPY R17, R17; VCOPY R18, R18; F B6 LD R19, 20 (SP)
          F B6 MOV R16, R7; F B6 MOV SP, R8
51    VCOPY R19, R19
```

The above fragment of code shows a branch instruction BT B6, L10 (#6) followed by several other instructions. The immediate value of six means that six instructions after the branch are to be executed, before the branch is taken. The branch will only execute if B6 is true. The virtual copy (VCOPY) instructions are only inserted during scheduling to enforce long instruction latencies. As VCOPYs are not included in the output code executed by the HSA simulator, they are not included in the branch count.

10. Branch Percolation

HSS maintains information concerning the branches associated with each basic block in a data structure called "listofbranches". Initially, for each basic block containing a branch, the 'branchlistptrhead' and 'branchlistptrtail' pointer fields in the basic block descriptor are set to point to a "listofbranches" data structure which contains information about the initial location of the branch. As the branch percolates up through predecessor basic blocks, these structures are updated to maintain the current state of the branch. With each new

instantiation of a branch in a predecessor basic block a new "listofbranches" node is added to the list (Fig 10.1). If a branch happens to be the only instruction in the basic block and it then moves up into a predecessor basic block the now empty original basic block is retained and its branch pointers are updated to point to the newly scheduled branch. Thus, HSS always retains the concept of only one branch per basic block.

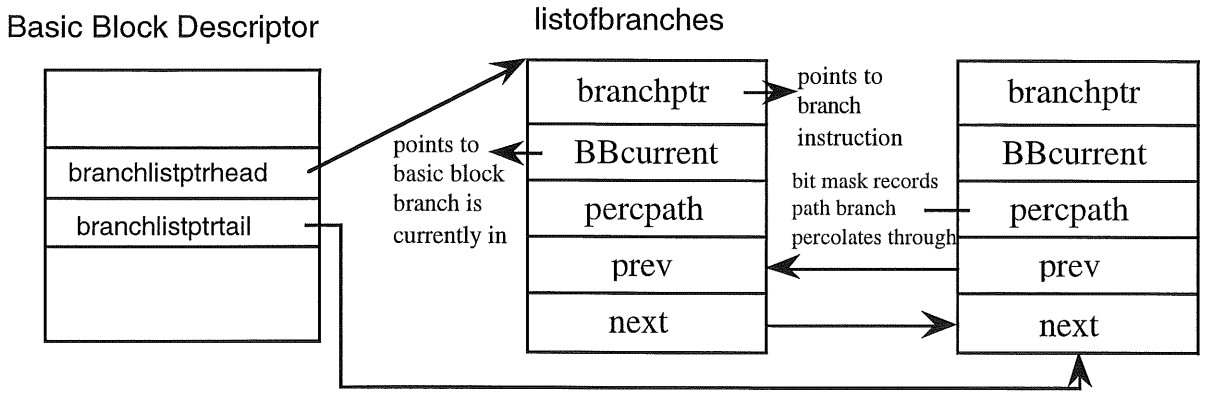

Fig 10.1 Branch Information Descriptors

HSS treats branch instructions in a similar manner to other instructions during percolation in that both CheckCoexist and CheckPass are invoked to see if there are any dependencies between the boolean register of a branch and the boolean registers of other instructions. However, the percolation of branches differs in two important respects. Firstly, branches can only move up the number of delay slots specified (Section 9). Secondly, if a branch (second branch) moves up into a predecessor basic block which already contains a branch instruction (first branch), then the second branch becomes an instruction in the first branch's delay slot, and other instructions in those delay slots after the second branch are also included in the second branch's delay slots. An example showing the situation in which a branch has moved up into a predecessor basic block is shown below:

```
36    SUB R4, R6, R4; BT B8, L10 (#6); F B8 BT B9, L10 (#14)
37    ADD R4, R4, #-60
38    ST (R0, R16), R4; MOV R16, R7; T B8 LD R4, _seed;
      F B8 T B9 LD R6, _seed
11    VCOPY R6, R6; T B9 MOV R17, #1; T B9 MOV R16, R10;
      F B9 MOV R16, R11; F B9 MOV R17, R8; F B9 MOV R18, R9;
      F B9 MOV R19, R12; F B9 MOV PC, RA (#2);
            F B9 MOV SP, R13
11    T B9 MOV R4, R6
0     NOP
.
.
L10:
      ADD R7, R16, #4; LES B8, R17, #9; etc...
```

The branches are highlighted in bold. The BT B8, L10 instruction is already in the predecessor basic block. The BT B9, L10 instruction has moved up from the sequential successor and obtained the guard F B8. This branch instruction then becomes one of the instructions in the first branch's delay slots. In turn, its own delay slots include LIWs from the sequential successor it moved up from. In the second branch's delay slots a MOV PC, RA instruction has also moved up from a sequential successor and obtained the guard F B9. In its delay slots an instruction has moved up from a branch target successor, labelled L10, and has obtained the guard T B9. A NOP is inserted in the final delay slot of the MOV PC, RA as two delay slots have been defined in the **hsp_const.h** file.

11. Merging

HSS uses a technique termed merging as one way of overcoming the limitations associated with true data dependencies. Merging involves combining or collapsing two instructions into a single instruction. The case analysis is very extensive and all the code concerned with merging can be

found in the HSS file **hsp_merge_instructions.c.**

There are three different categories of merged instructions. The first category involves a pair of instructions in which the first instruction is a MOV instruction (MOV merging). The second category involves a pair of instructions where both of the instructions contain an immediate source operand (immediate merging). The third category has a MOV instruction as the second or percolating instruction which can be converted to the same type as the first instruction (MOV Reabsorption). Merging is enabled by setting the parameters "MOV_MERGE" and "LD_MOV_MERGE" in file **hsp_const.h** to true.

11.1 MOV Merging

When a true data dependency occurs between a MOV instruction and any percolating instruction, a check is made to see whether the two instructions can be merged. If successful the percolating instruction can continue its progress through the basic block. A very comprehensive set of MOV merges has been implemented on HSS. Below are several examples to show the sorts of cases that have been implemented.

In the following example an ADD instruction merges with a MOV:

```
MOV R6, R7
ADD R3, R6, R5 /*percolating instruction*/
```

becomes:

```
MOV R6, R7; ADD R3, R7, R5
```

Thus the replacement of R6 by R7 in the ADD instruction removes the true data dependency.

In the following example both instructions contain an immediate operand:

```
MOV R6, #4
ADD R3, R6, #5 /*percolating instruction*/
```

becomes:

MOV R6, #4; MOV R3, #9

The immediate values have been added together and the ADD instruction has been changed to a MOV instruction.

The example below shows a ST instruction merging with a MOV containing a zero value immediate operand:

MOV R3, #0

ST (R1, R2), R3 /*percolating instruction*/

becomes:

MOV R3, #0; ST (R1, R2), R0

The R3 in the ST instruction has been changed to R0 (in HSP R0 always has the value zero).

The code below shows a relational instruction merging with a MOV instruction containing an immediate operand:

MOV R4, #4

GT B1, R4, R3 /*percolating instruction*/

becomes:

MOV R4, #4; LTE B1, R3, #4

The true data dependence is between the register R4 in both instructions and therefore the GT has become LTE to allow the operands of the relational instruction to be exchanged.

There is another related group of merge cases that involve the Boolean guards which are used by HSP to conditionally execute instructions. Here the true data dependence is between a Boolean register set by the first instruction, either a relational instruction or a MOV Bi, Bj instruction, and a percolating instruction which is guarded. In these cases the guard itself can be changed or removed to allow the percolation to continue.

The following example shows the result of an ADD instruction guarded by Boolean register B3 merging with a special relational instruction:

```
EQ B3, R0, R0                    /*B3 := true*/
T B3 ADD R10, R11, R12
```

becomes:

```
EQ B3, R0, R0; ADD R10, R11, R12
```

The instructions EQ Bi, R0, R0 and NE Bi, R0, R0 are only used because of the absence of the instruction forms MOV Bi, #true or MOV Bi, #false in HSA. Because we know that B3 is always going to be true, the guard T B3 can be removed from the ADD instruction. If B3 always evaluated to false the percolating instruction would be replaced by a NOP.

The example below shows a LD instruction guarded by Boolean register B1 merging with a MOV instruction where both of its operands are Boolean registers:

```
MOV B1, B2
T B1 LD R4, (R0, R6) /*percolating instruction with guard*/
```

becomes:

```
MOV B1, B2; T B2 LD R4, (R0, R6)
```

The guard has now changed from B1 to B2 allowing the LD to coexist with the MOV. Relational moves will also combine with branch instructions in the same way as above and this is shown below:

```
MOV B1, B2
BT B1, Label
```

becomes:

```
MOV B1, B2; BT B2, Label
```

If the boolean is a constant, the branch will either be removed or altered to a branch always (BRA) instruction. For example:

```
EQ B1, R0, R0
BT B1, Label
```

becomes:

```
BRA Label
```

11.2 Immediate Merging

Immediate merging involves any pair of instructions which both have immediate values as their third operands. For example:

SUB R3, R6, #3

ADD R4, R3, #1

becomes:

SUB R3, R6, #3; ADD R4, R6, #-2

In this case, the first immediate value has been subtracted from the second immediate value, and has been added to the substituted first source operand in the ADD instruction.

11.3 MOV Reabsorption

In this type of merging the second instruction which is a MOV is converted to the same type as the first instruction:

ADD R3, R4, R5

MOV R6, R3 /*percolating instruction*/

becomes:

ADD R3, R4, R5; ADD R6, R4, R5

Thus R3 in the MOV instruction has been replaced by R4 and R5 in the first ADD instruction thereby converting the MOV to an ADD instruction. The idea behind this type of merging is to reabsorb MOVs generated by renaming and hence reduce code expansion. In the case where the first instruction is a Load, the parameter "LD_MOV_MERGE" can be set independently from "MOV_MERGE" to disable this type of merging. Two parameters are provided because duplication of loads can reduce speedup by pre-empting the use of a limited number of cache read ports.

There are two major complications associated with merging. Firstly, merging can result in instructions being inserted in the middle of instruction groups rather than at the end. Secondly, if an instruction is inserted in the middle it may have to be renamed. An instruction may have to be inserted in the middle of a group if it has passed several other instructions in the group and

then merges with an instruction. Since merging in general alters the source operands of a percolating instruction, merging may introduce a false data dependency with one of the instructions that has already been passed. To remove this false dependence the percolating, and newly merged, instruction will have to be inserted directly before the dependent instruction. Insertion in the middle of a group may in turn result in further false dependencies since the destination operand may become the source operand of an instruction further towards the end of the same instruction group. This false dependence can be removed by renaming the destination register of the inserted instruction. An example will clarify the above points:

LIW1 ADD R1, R2, R3; MOV R7, R8; LD R8, (R0, R5);
SUB R9, R3, #4

LIW2 ADD R3, R7, R4 /*percolating instruction*/

The ADD instruction in LIW2 will merge with the MOV instruction in LIW1 creating the new instruction ADD R3, R8, R4 and will need to be inserted in front of the LD instruction to avoid a false dependency with register R8 in that instruction. However, the SUB instruction in LIW1 now has a true data dependency on register R3 with the new ADD instruction and therefore register R3 in the new ADD instruction will have to be renamed. The final situation is shown below:

LIW1 ADD R1, R2, R3; MOV R7, R8; **ADD R6, R8, R4**;
LD R8, (R0, R5); SUB R9, R3, #4

LIW2 MOV R3, R6 /*introduced because of renaming*/

Finally, the instruction that a percolating instruction merges with is never altered or deleted for two reasons: Firstly, the first instruction is always retained in case its destination value is required by other instructions. Secondly, even if the destination is clearly dead, the MOVs cannot be deleted during the percolation process since only potential insertion points are being selected at this stage. Only later will a higher level control mechanism determine whether the insertions and hence the merges actually take place. As a result the MOV instruction can only be deleted if and when a further

percolating instruction reaches the group and determines that R6 is dead.

12. Combining

Instruction combining, also called static data dependence collapsing, is discussed in detail in our recent paper (Steven et al, 1998). Instruction combining is identical in principle to instruction merging in that it overcomes the problems associated with true data dependencies by combining and changing operands in instructions to allow the percolating instruction to continue moving through a basic block. In fact, the IBM VLIW team led by Kemal Ebcioglu (Nakatani, 1989) uses immediate merging examples as examples of instruction combining. However, whereas merging restricts the instructions that can be combined to those that retain a maximum of three operands, combining removes this restriction. To implement combining, special instructions with four operands are required. Appendix A.2 shows the full set of combined instruction formats available in the HSA architecture. Two examples are shown below:

```
ADD Ri, (Rj + Rk), Rl /*Ri := Rj + Rk + Rl*/
AND Ri, (Rj ASL #imm), Rk /*Ri := (Rj << #imm) AND (Rk);
                                  0 <= imm < 32*/
```

When a percolating instruction combines, the result is logically a single instruction with three source operands. The first instruction is also retained. In both respects merging and combining are therefore conceptually identical. Nonetheless, no permanent coupling is attempted at the percolation stage. Instead the combined instruction pair is inserted into the schedule as two separate instructions. Only a tag on each instruction indicates that the instructions have been combined.

The crucial advantage of this arrangement is that the first instruction of a combined instruction pair can be repercolated at a later stage of the percolation process. It may even recombine with another instruction. Repercolation therefore dismantles a combined pair into two separate instructions.

It is during the final instruction output that a check is made for any instructions which can be permanently combined and it is then that they are output as one combined instruction. The advantages of this approach are that it is extremely flexible. The instructions are still treated separately by the scheduler so that the first or both instructions in a pair may repercolate further through the code at a later stage of the scheduling process. An example is shown below to clarify these points:

```
MULT R7, R9, #14
ADD R6, R7, R5
```

becomes:

```
MULT R7, R9, #14; MULT R7, R9, #14; ADD R6, R7, R5
                        /*combined but logically one instruction*/
```

and during the final output becomes:

```
MULT R7, R9, #14; ADD R6, (R9 * #14), R5
```

Results using trace driven simulations to study combining with several different compiler optimisations have shown a potential improvement in performance of 50 to 75% (loc. cit).

13. Memory Disambiguation

Data dependencies do not only occur between registers; they also occur between the memory locations referenced in LD and ST instructions and, as with other data dependencies, they can severely degrade program performance if not dealt with.

HSS uses a technique termed static memory disambiguation to differentiate between the memory locations referenced by two instructions. The code to do this is in the HSS file **hsp_disambiguate.c**. For example, to decide whether a LD can percolate ahead of a ST instruction, which is only safe if the two addresses are different, the two addresses are compared and one of three values is returned as follows:

Different: Addresses are always different.

Same: Addresses are always the same.

Fail: Address cannot be distinguished.

If the value returned is different, the LD instruction can percolate ahead of the ST. Also, if the value returned is the same, the LD can be replaced by a MOV instruction as shown below:

```
ST (R0, R5), R6
LD R10, (R0, R5)
```

becomes:

```
MOV R10, R6
ST (R0, R5), R6
```

Since the value required is already in the register R6, the register itself can be used to put the value in R10 rather than loading R10 from memory.

On the other hand, if the case involves a LD instruction followed by a ST instruction, i.e. the opposite case, then the addresses **must** be different if the ST instruction is to percolate ahead of the LD instruction.

Memory locations cannot always be disambiguated at compile time and therefore dynamic memory disambiguation is one technique that could be considered to deal with the problem. In this case a pair of load and store instructions is replaced with code that compares the two addresses dynamically at run-time. For example, consider the pair of instructions below:

```
ST 4(R5), R8
LD R9, 8(R6)
```

Dynamic code required:

```
ADD R3, R5, #4      /*Compute store address*/
ADD R4, R6, #8      /*Compute load address*/
LD R9, 8(R6)        /*If addresses differ perform load*/
EQ B1, R3, R4       /*Compare addresses for equality*/
T B1 MOV R9, R8     /*If same obtain value from register*/
ST 4(R5), R8
```

The use of guarded instruction execution has removed the need for two branch instructions. The LD instruction is now ahead of both the ST instruction and the relational instruction.

14. Inlining

Inlining is an effective technique for uncovering additional instruction scheduling opportunities. Inlining is a mechanism whereby the sequence of instructions comprising a procedure or function is duplicated and inserted into the calling procedure in place of the function call. The function call and return instructions can be removed along with many of the instructions which manipulate the stack frame and save and restore registers during function entry and exit.

HSS can optionally inline procedures before scheduling proceeds (Fig 7.1). There are several parameters provided which control inlining. All the inlining parameters can be found in file **hsp_const.h** (Appendix B). The parameter "INLINEPARAM" represents the value true or false and controls whether inlining is invoked by the scheduler. "Inline_Recursive_Calls" also represents the value true or false and determines whether recursive procedures are inlined or not. "Inline_Proc_Size1_Threshold" defines the maximum number of basic blocks a procedure called from within a loop can contain for it to be inlined. "Inline_No_Of_Calls1_Threshold"[3] denotes the maximum number of times a procedure can be called from within a loop for it to be inlined. "Inline_Proc_Size2_Threshold" represents the maximum number of basic blocks a procedure called from outside a loop can contain for it to be inlined. "Inline_No_Of_Calls2_Threshold" defines the maximum number of times a procedure can be called from outside a loop for it to be inlined. "Inline_Proc_Size3_Threshold" denotes the maximum number of basic blocks a procedure called from within a recursive procedure can contain for it to be inlined. "Inline_Nesting_Threshold" defines the maximum number of nested inlinings that can occur when a procedure which has been

[3] "Inline_No_Of_Calls1_Threshold is not used at present.

inlined contains procedure calls which may themselves be inlined. A threshold is set which avoids endless inlining. Finally, "UNCALLEDPROCSDELETED" represents the value true or false and if at the end of inlining, a procedure is no longer called because all possible inlinings of it have taken place, and this parameter is set to true, then the procedure is deleted.

Inlining is implemented in the file **hsp_inline.c.** First, information regarding both the procedures and the basic blocks within them is recorded. Those procedures which are recursive are identified and recorded in the 'recursiveproc' field in the procedure node concerned. The number of occasions a procedure is called is recorded in the proc_call_frequency field in the procedure node concerned. The number of basic blocks in a particular procedure is recorded in a local variable as the inlining takes place. This value changes as inlining proceeds because inlined procedures add additional basic blocks to the enclosing procedure.

The function "InlineProcCalls" in **hsp_inline.c** checks each basic block for a BSR instruction which calls a procedure. The basic block that the BSR is in is then passed to the function "ShouldInline" to determine whether inlining should occur. The function "ShouldInline" checks various cases against the inlining parameters in **hsp_const.h** to decide about inlining a particular procedure. There are three major cases to consider. Firstly, the call might be from within a loop. Secondly, the call might be within a recursive procedure but outside a loop. Thirdly, the call might be from outside a loop.

If the call to a procedure is contained within a loop then aggressive inlining is used. If the call within the loop is directly recursive, and the parameter "Inline_Recursive_Calls" is set to false inlining does not occur. However, if "Inline_Recursive_Calls" is set to true and inlining can occur, then either the number of basic blocks within the called procedure must not exceed the value defined by the parameter "Inline_Proc_Size1_Threshold" or the number of

calls to the procedure must not be greater than one.

If a non-recursive call is within a recursive procedure but outside a loop a different set of parameters is used. In this case inlining will occur if the number of basic blocks in the called procedure is not greater than the value defined by the parameter "Inline_Proc_Size3_Threshold" or the number of calls to that procedure is not greater than one, inlining can occur.

If the call is not within a loop or a recursive procedure then less aggressive inlining is used and a third set of parameters define the limits. If both the number of basic blocks in the called procedure is not greater than the value denoted by the parameter "Inline_Proc_Size2_Threshold" and the number of calls to the procedure is not greater than the number denoted by the parameter "Inline_No_Of_Calls2_Threshold" or the number of calls to the procedure is not greater than one then inlining can occur. Note that library routines cannot be inlined

Once it has been decided that a procedure can be inlined the function "Inline" is invoked. This function passes in the nesting level of the procedure as one of its parameters. The very first time a procedure is inlined it has a nesting level of one. If the procedure, which has been inlined, has calls to procedures which can in turn be inlined, then this is reflected by incrementing the nesting level for each nested inlining. The nesting level is compared with the parameter "Inline_Nesting_Threshold" and if the nesting level is less than this parameter the function "InlineProcCalls" is called again with the nesting level incremented. Thus, "Inline_Nesting_Threshold" ensures that procedures are not inlined indefinitely. If a procedure can be inlined the 'proc_call_frequency' field in the procedure node is decremented to reflect the fact that the number of calls to this procedure is now one less than before. Any procedure called by the newly inlined procedure has its 'proc_call_frequency' field incremented to indicate that extra calls to it occur owing to duplication of the calling procedure.

Once the inlined procedure has been inserted all labels in the inlined procedure are renamed and all branch targets are updated. Any redundant procedure entry and exit instructions are also removed. These include any loads and stores which save registers on procedure entry and restore them on procedure exit where the registers concerned are not used by the calling procedure. The BSR calling the inlined procedure is also no longer required so it is deleted. Finally, the MOV PC, RA - the last instruction in the inlined procedure - is no longer needed, so it too is deleted (Fig 14.1).

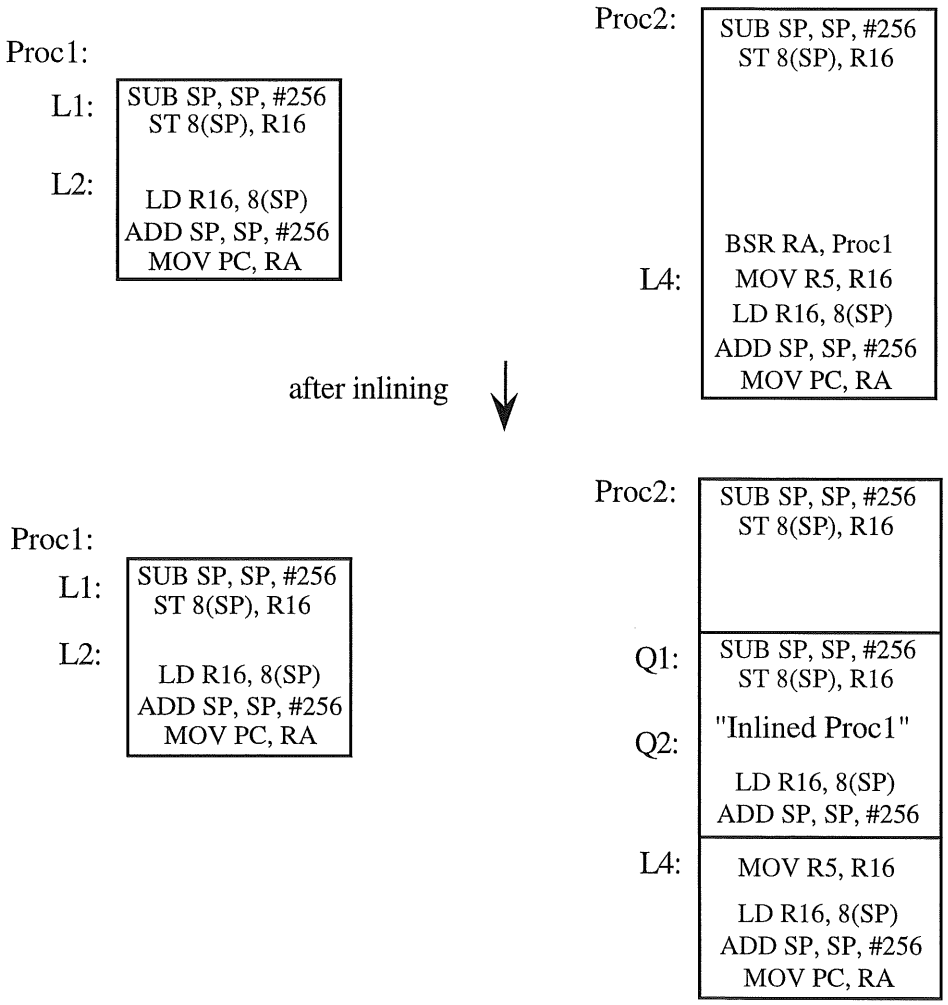

Fig 14.1 Inlining a Procedure

If a procedure has no more calls to it, that is, if every call to it has been inlined, the 'proc_call_frequency' field will have the value zero. Then if the

parameter "UNCALLEDPROCSDELETED" is set to true the now uncalled original procedure will be deleted.

Once inlining has been completed, the now modified benchmarks are presented to the HSS backedge percolation for scheduling (Fig 7.1).

15. The HSS Algorithm

The objectives of the HSS algorithm are two-fold. The first objective is to achieve software pipelining with arbitrarily complex loops. The second objective is to reduce code expansion by avoiding unproductive code motion across the loop back edge.

The HSS Scheduling Algorithm is shown in Fig 15.1.

Phase 1

Schedule loop with percolation across loop back edge(s) disabled.

Phase 2

Do{

Percolate each instruction from the first instruction group in the loop around the loop back edge provided:

1. There exists a chain of dependent instructions from the percolating instruction to an instruction in the final instruction group before the loop back edge. Note the chain may involve an antidependence. (In the case of a delayed branch, it is sufficient for the chain to reach the branch)
2. The instruction chain cannot be collapsed using either instruction merging or instruction combining.

If at least one instruction has been moved, recompact the loop with the back edge disabled.}

until (no further instructions can be moved)

Fig 15.1 The HSS Scheduling Algorithm

To make the HSS algorithm clearer a single basic block loop is scheduled below showing how the iteration interval is reduced. It is assumed in this example that all instructions have a latency of one and that there are no delay slots after the branch instruction.

Example: Original Code

```
Loop:
        LD R16, (R1)
        ADD R16, R16, #17
        ST (R3), R16
        ADD R1, R1, #4;
        ADD R3, R3, #4
        SUB R2, R2, #1
        NE B1, R2, #0
        BT B1, Loop
```

The initial loop requires eight cycles to execute. During the first pass each instruction is scheduled in turn, starting at the top of the loop, and each instruction is moved or percolated as far up as possible:

```
Loop:
        LD R16, (R1); ADD R1, R1, #4; ADD R4, R3, #4;
                SUB R2, R2, #1
        ADD R16, R16, #17;  NE B1, R2, #0
        ST (R3), R16; MOV R3, R4; BT B1, Loop
```

The third ADD instruction has had its R3 register renamed. Each loop iteration now requires three cycles instead of eight.

Instructions are then percolated from the first instruction group around the loop back edge. There must be a chain of dependencies from the percolating instruction to the end of the loop and instructions are not allowed to percolate back into the first group. During the first percolation around the back edge a loop prelude is created. Each instruction that moves around the back edge is also percolated into the loop prelude. The loop is then rescheduled with the loop back edge percolation disabled. Once this rescheduling has been done the back edge percolation is re-enabled and instructions from the first group are again moved around the loop back edge. This continues until the final code is produced which is shown below:

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
                LD R17, (R1); ADD R1, R1, #4
Loop1:
        SUB R2, R2, #1; ST (R3), R16; ADD R3, R3, #4;
                ADD R16, R17, #17; LD R17, (R1); ADD R1, R1, #4
                BT B1, Loop1; NE B1, R2, #0;
```

The loop prelude comprises partial iterations at different stages of completion. The loop body completes the iterations started in the loop prelude, and has now been reduced to a single instruction group.

16. Implementation of HSS

HSS is a large program, written in 'C', currently comprising 20 modules and approximately 12,000 lines of code.

In order that HSS can schedule the benchmarks, the assembly instructions of the benchmark have to be inserted into appropriate data structures or descriptors. The descriptor has several fields, where each field is devoted to a particular piece of information about the instruction (Fig 16.1).

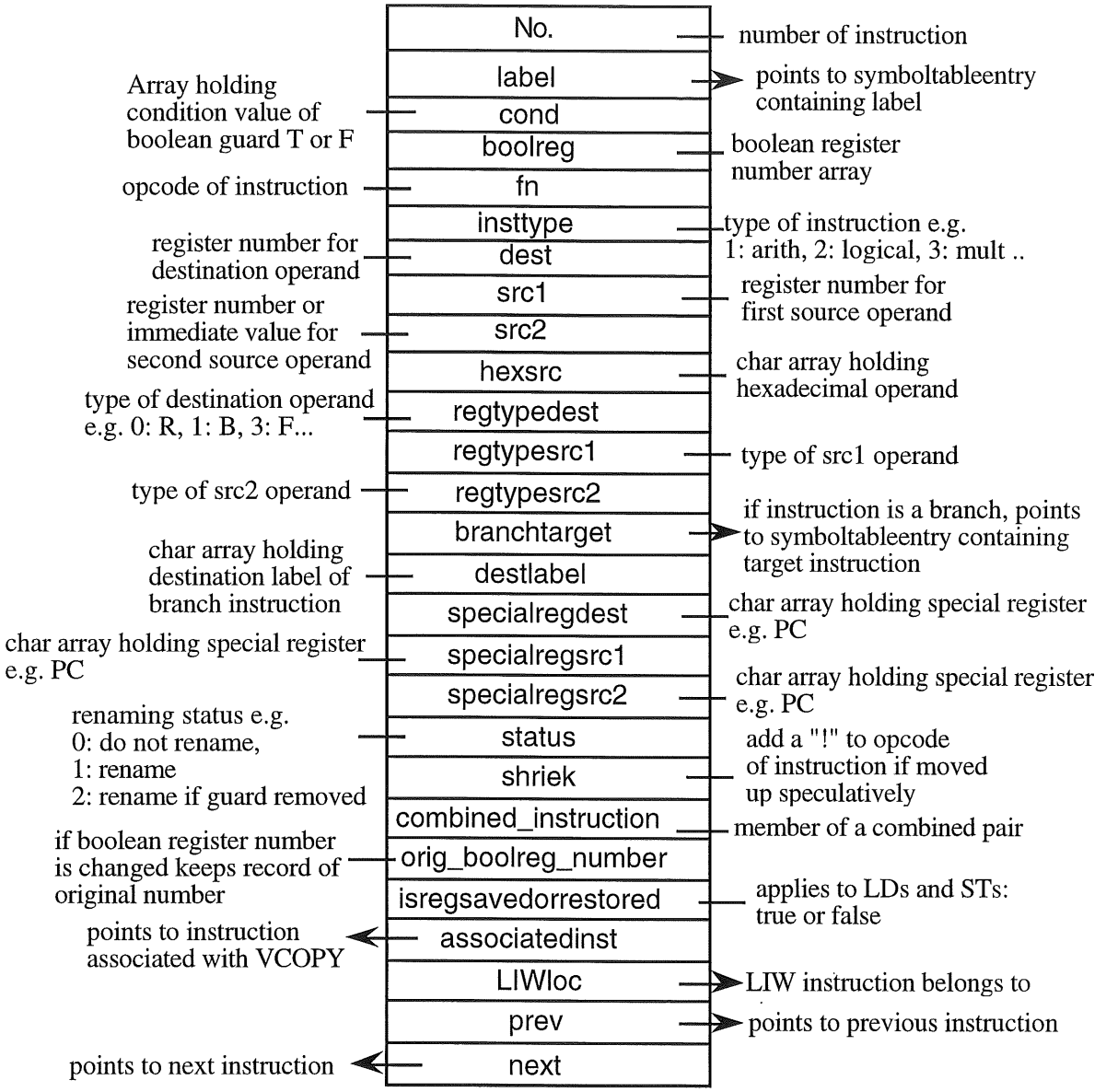

Fig 16.1 The Fields in the HSS Instruction Node Descriptor

The benchmarks are presented to HSS in the form of HSA assembly instructions. The file **hsp_sched.c** contains the code necessary to read in these instructions and to construct the data structures manipulated by the scheduler. The instructions are read in one character at at time and once a token has been recognised in the form of a meaningful group of characters, its value is inserted into the appropriate field in the data structure instnode. For example, if an instruction is guarded, the boolean condition is recorded by setting the appropriate field in an array of conditions to true or false. The

boolean register number is recorded by setting the appropriate field in the 'boolreg' array to true. Thus T B5 would set cond[5] to true and boolreg[5] to true. The instruction opcode or function is inserted in the 'fn' array as a string of characters. The operands of the instruction are inserted in the 'dest', 'src1' and 'src2' fields and their types are inserted in the 'regtypedest', 'regtypesrc1' and 'regtypesrc2' fields. For example, upon recognising a destination register, R6, the value 6 would be inserted in the 'dest' field and a zero would be inserted in the 'regtypedest' field, indicating that R6 was an integer register. Had it been B6 rather than R6, the 'regtypedest' field would have been set to 2 for a boolean register and so on. When an instruction is output the 'regtypedest' field is checked and the appropriate character B, R or F is concatenated with the 6 to form a register.

As each instruction is read into an instruction node, it is then assigned to a LIW node and the LIW nodes are linked together (Fig 16.2). LIW nodes form the focus of the instruction groups generated during code motion.

If any instruction is labelled, for example:

_Permute: SUB SP, SP, #256

then the label is inserted into a data structure called symboltableentry in the field 'instlabel' which is an array of char. A pointer in the symbol table then points back to the instruction associated with the label (Fig 16.2).

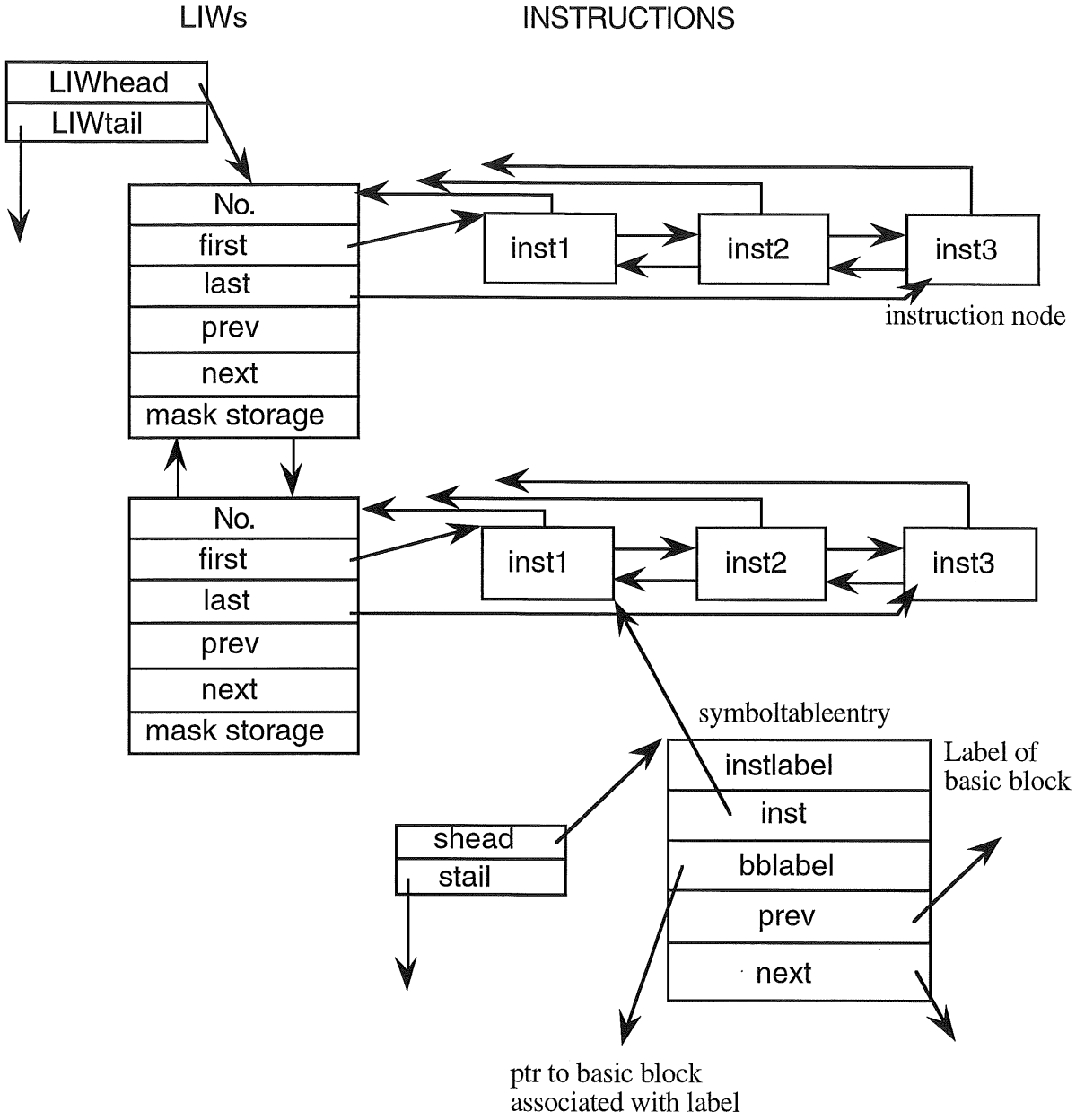

Fig 16.2 HSS Instruction Groups and Symbol Table

Once all the instructions have been read in, code in the file **hsp_sched1.c** divides groups of instructions into basic blocks (Fig 16.3). Basic blocks are sequences of code that have one entry point and one exit point. Information about basic block successors, predecessors, branch targets and predecessor branch targets are inserted into the basic block nodes. The symbol table entry field 'inst' is then compared with the first instruction in the basic block. If it matches, the field 'bblabel' in symboltableentry is updated to point to that

basic block (Fig 16.2). In turn, the field 'Label' in the basic block points back to the appropriate symboltableentry node (Fig 16.3).

Procedures are detected and information regarding procedures is inserted in the procedure nodes (Fig 16.3). For example, pointers to the first and last basic blocks in the procedure are inserted in 'headbb' and 'tailbb' fields respectively.

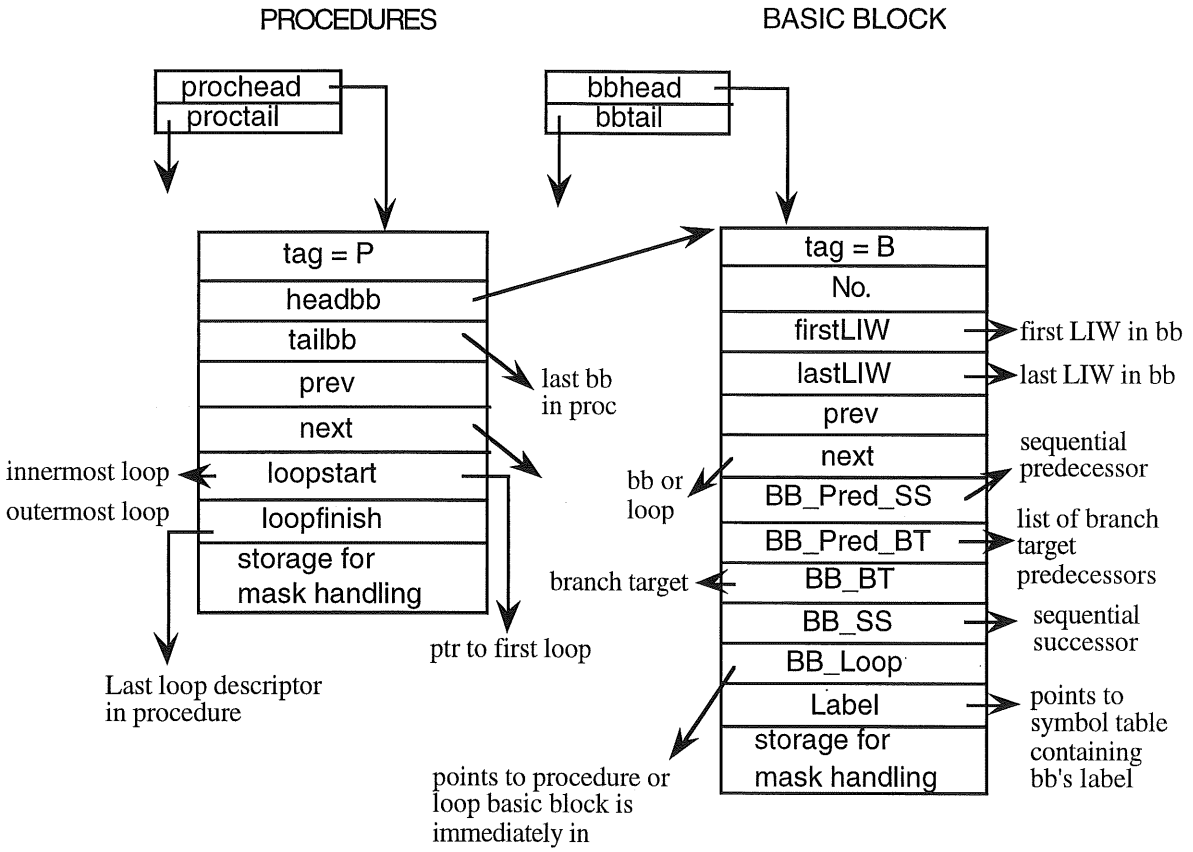

Fig 16.3 HSS Procedure and Basic Block Descriptors

Finally, loops are detected and information about loops is inserted in the loop nodes (Fig 16.4). Information about loops is also inserted into the procedure nodes (Fig 16.3).

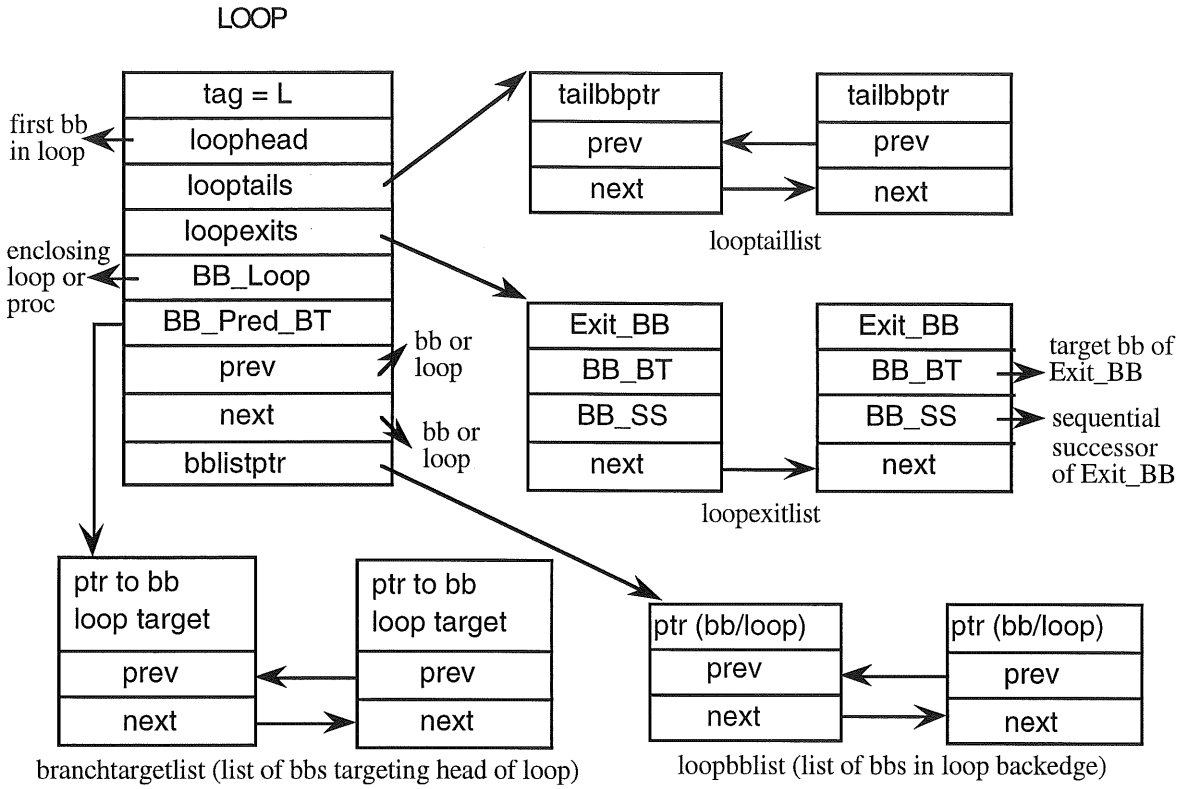

Fig 16.4 HSS Loop Descriptor

Pointers to the innermost loop and outermost loop are inserted in the fields 'loopstart' and 'loopfinish' respectively. These fields are used to find which loops to schedule first. The innermost loop is always scheduled first. A linked list of loops joins the loopstart to the loopfinish and determine the order in which loops are scheduled.

Live range analysis is carried out in file **hsp_mask_handling.c**. Each basic block descriptor records whether each register is live on entry to the basic block in the fields marked "storage for mask handling" (Fig 16.3). Similar fields in the LIW descriptors record whether each register is live immediately after all the instructions in the LIW group have been executed (Fig 16.2).

Once the instructions have been inserted into the appropriate instruction nodes and all necessary information has been gathered and inserted into the other data structures, the benchmark is ready for scheduling.

Although procedure nodes, loop nodes and basic blocks nodes have different functions to perform within the scheduler, they share a single union type within the scheduler. The tag field indicates whether a particular structure is a procedure (P), a loop (L) or a basic block (B). This arrangement allows the HSS scheduler data structures to directly reflect the static structure of the code being scheduled. In particular, procedure descriptors can point to nested loops and basic blocks using a single pointer type. Similarly, basic blocks and loops can point to the data structures which encapsulate them, irrespective of whether the structure involved is a loop or a procedure.

17. Conclusions

This document has introduced the HSS scheduler. Although this document has endeavoured to be as comprehensive as possible, many details and features have been omitted in order to give the reader a clear and straightforward impression of the scheduler. The scheduler is constantly evolving and therefore this document should not be regarded as definitive. Future documents will record further changes to HSS's implementation.

References

Collins, R. "Developing A Simulator for the Hatfield Superscalar Processor," Division of Computer Science Technical Report No. 172, University of Hertfordshire, December 1993.

Collins R and Steven G B "An Explicitly Declared Delayed-Branch Mechanism for a Superscalar Architecture," presented at EuroMicro94, Liverpool, September 1994. Published in Microprocessing and Microprogramming, Vol.40 , No.10-12, December 1994, pp677-680.

Collins, R. "Exploiting Instruction-Level Parallelism in a Superscalar Architecture," PhD Thesis, University of Hertfordshire, October 1995.

Ebcioglu, K. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," Proceedings of the 20th Annual Workshop on Microprogramming, ACM Press, 1987, pp69-79

Ebcioglu, K, Groves, R.D., Kim, K. Silberman, G.M. and Ziv, I "VLIW Compilation Techniques in a Superscalar Environment," SigPlan94, Orlando, Florida, 1994, pp36-48.

Egan, C, Steven, F L and Steven, G B "Delayed Branches versus Dynamic Branch Prediction in a High-Performance Superscalar Architecture," Euromicro97, Budapest, September 1997.

Fisher, J A "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, C-30, (7), July 1981, pp478-490.

Lam, M S "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," SIGPLAN 88 Conference of Programming Language Design and Implementation, Georgia, USA, June 1988, pp318-328

Malik, N, Eickemeyer, R.J., and Vassiliadis, S. "Instruction-Level Parallelism from Execution Interlock Collapsing," Computer Architecture News, September 1992, pp38-43.

Nakatani, T. and Ebcioglu, K. "Combining as a Compilation Technique for VLIW Architectures," 22nd Annual International Workshop on Microprogramming and Microarchitecture, SIGMICRO Newsletter, Vol 20, No. 3, September 1989, pp43-55.

Nicolau, A "Uniform Parallelism Exploitation in Ordinary Programs," Proceedings of the International Conference on Parallel Processing, August 1985, pp614-618.

Potter R and Steven G B "Investigating the Limits of Fine-Grained Parallelism in a Statically Scheduled Superscalar Architecture," 2nd International Euro-Par Conference Proceedings, Vol.2, Lyon, France, August 1996, pp779-788. (Published as Lecture Notes in Computer Science 1124 by Springer).

Rau, B R "Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops," Micro 27, November 1994, San Jose, California, pp63-73.

Rau, B R and Fisher, J. A. "Instruction-Level Parallel Processing: History, Overview and Perspective," The Journal of Supercomputing, Vol.7, No. 1/2, 1993, pp9-50.

Sazeides, Y. and Vassiliadis, S. "The Performance Potential of Data Dependence Speculation & Collapsing," IEEE Micro 29, 1996, pp238-247.

Stallman, R M "Using and Porting GNU CC," Free Software Foundation, 1989.

Steven G B and Collins R "Instruction Scheduling for a Superscalar Architecture," Proceedings of the 22nd Euromicro Conference, Prague, September 1996, pp643-650.

Steven G B, Christianson D B, Collins R, Potter R. and Steven F L "A Superscalar Architecture to Exploit Instruction Level Parallelism," Microprocessors and Microsystems, Vol.20, No 7, March 1997, pp391-400.

Steven F L, Steven G B and Wang L "Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor," IEE Proceedings - Computers and Digital Techniques, Vol.142, No.1, January 1995, pp 23-31.

Steven F L, Potter, R D, Steven G B and Vintan, L "Static Data Dependence Collapsing in a High-Performance Superscalar Processor," To be

presented at the 3rd International Conference on Massively Parallel Computing Systems, Colorado, April 1998.

Vassiliadis S, Blaner, B and Eickemeyer, R.J. "On the Attributes of the SCISM Organization," Computer Architecture News, September 1992, pp44-53.

Warte, N.J, Mahlke S A, Hwu W W and Rau B R "Reverse If-Conversion," SigPlan 93, Albuquerque, New Mexico, June 1993, pp290-299.

Appendix A HSP Instruction Set

Appendix A.1: Basic HSP Instruction Set

A1.1 Arithmetic Unit Instructions

ADD R_i, R_j, R_k	$R_i := R_j + R_k$
ADD R_i, R_j, #Imm	$R_i := R_j$ + #Imm
ADDV R_i, R_j, R_k	$R_i := Rj + R_k$; trap on signed arithmetic overflow
ADDV R_i, Rj, #Imm	$R_i := R_j$ + #Imm; trap on signed arithmetic overflow
SUB R_i, Rj, R_k	$R_i := R_j - R_k$
SUB R_i, R_j, #Imm	$R_i := R_j$ - #Imm
SUBV R_i, R_j, R_k	$R_i := R_j - R_k$; trap on signed arithmetic overflow
SUBV R_i, Rj, #Imm	$R_i := R_j$ - #Imm ; trap on signed arithmetic overflow
ADDC R_i, R_j, R_k	$R_i := R_j + R_k$ + Carry; Carry flag set by result
ADDC R_i, R_j, #Imm	$R_i := R_j$ + #Imm + Carry ; Carry flag set by result
SUBC R_i, R_j, R_k	$R_i := R_j - R_k$ + Carry ; Carry flag set by result
SUBC R_i, R_j, #Imm	$R_i := R_j$ - #Imm + Carry ; Carry flag set by result
DIV R_i, R_j, R_k	$R_i := R_j$ DIV R_k ; 32bits X 32 bits --> 32 bits
DIV R_i, R_j, #Imm	$R_i := R_j$ DIV Imm ; 32bits X 32 bits --> 32 bits
DIVV Ri, Rj, R_k	$R_i := Rj$ DIV R_k ; 32bits X 32 bits --> 32 bits; trap on overflow
DIVV R_i, R_j, #Imm	$R_i := R_j$ DIV Imm ; 32bits X 32 bits --> 32 bits; trap on overflow
MOD R_i, Rj, R_k	$R_i := R_j$ MOD R_k ; non pipelined; 32bits X 32 bits --> 32 bits
MOD R_i, R_j, #Imm	$R_i := R_j$ MOD Imm ; non pipelined; 32bits X 32 bits --> 32 bits
MOV R_i, B_j	If $B_j = 1$ $R_i = 1$ else $R_i = 0$
MOV R_i, SR	R_i := Status Register
MOVSF F_i, R_j	$F_i := R_j$; single-length transfer
MOVSF F_i, #Imm	F_i := #Imm; single-length transfer

A1.2 Shift Unit Instructions

ASL R_i, R_j, R_k	$R_i := R_j << (R_k$ AND 31)
ASL R_i, R_j, #Imm	$R_i := R_j <<$ #Imm where (0 <= Imm < 32)
ASLV R_i, R_j, R_k	$R_i := R_j << (R_k$ AND 31); trap on overflow

ASLV R_i, R_j, #Imm	$R_i := R_j$ << #Imm where (0 <= Imm < 32); trap on overflow
ASR R_i, R_j, R_k	$R_i := R_j$ >> (R_k AND 31)
ASR R_i, R_j, #Imm	$R_i := R_j$ >> #Imm where (0 <= Imm < 32)
LSR R_i, R_j, R_k	$R_i := R_j$ >> (R_k AND 31)
LSR R_i, R_j, #Imm	$R_i := R_j$ >> #Imm where (0 <= Imm < 32)
AND R_i, R_j, R_k	$R_i := R_j$ AND R_k
AND R_i, R_j, #Imm	$R_i := R_j$ AND #Imm
OR R_i, R_j, R_k	$R_i := R_j$ OR R_k
OR R_i, R_j, #Imm	$R_i := R_j$ OR #Imm
EOR R_i, R_j, R_k	$R_i := R_j$ EOR R_k
EOR R_i, R_j, #Imm	$R_i := R_j$ EOR #Imm
EXT R_i, R_j	$R_i := R_j$ (byte sign extended);
BIC Ri, R_j, R_k	$R_i := R_j$ AND ~(R_k)
BIC R_i, R_j, #Imm	Ri := R_j AND ~(#Imm)

A1.3 Multiply Unit Instructions

MULT R_i, R_j, R_k	$R_i := R_j$ MULT R_k ; 32bits X 32 bits --> 32 bits
MULT R_i, R_j, #Imm	$R_i := R_j$ MULT #Imm ; 32bits X 32 bits --> 32 bits
MULTV R_i, R_j, R_k	$R_i := R_j$ MULT R_k; 32bits X 32 bits --> 32 bits ; trap on signed overflow
MULTV R_i, R_j, #Imm	$R_i := R_j$ MULT #Imm; 32bits X 32 bits --> 32 bits ; trap on signed overflow

A1.4 Relational Unit Instructions

Signed:

GTS B_i, R_j, R_k	B_i := (R_j > R_k)
GTS B_i, R_j, #Imm	B_i := (R_j > #Imm)
GES B_i, R_j, R_k	B_i := (R_j >= R_k)
GES B_i, R_j, #Imm	B_i := (R_j >= #Imm)
LTS B_i, R_j, R_k	B_i := (R_j < R_k)

LTS B_i, R_j, #Imm	$B_i := (R_j < \#Imm)$
LES B_i, R_j, R_k	$B_i := (R_j =< R_k)$
LES B_i, R_j, #Imm	$B_i := (R_j =< \#Imm)$

Unsigned:

GTU B_i, R_j, R_k	$B_i := (R_j > R_k)$
GTU B_i, R_j, #Imm	$B_i := (R_j > \#Imm)$
GEU B_i, R_j, R_k	$B_i := (R_j >= R_k)$
GEU B_i, R_j, #Imm	$B_i := (R_j >= \#Imm)$
LTU B_i, R_j, R_k	$B_i := (R_j < R_k)$
LTU B_i, R_j, #Imm	$B_i := (R_j < \#Imm)$
LEU B_i, R_j, R_k	$B_i := (R_j =< R_k)$
LEU B_i, R_j, #Imm	$B_i := (R_j =< \#Imm)$

Signed and Unsigned

EQ B_i, R_j, R_k	$B_i := (R_j = R_k)$
EQ B_i, R_j, #Imm	$B_i := (R_j = \#Imm)$
NE B_i, R_j, R_k	$B_i := (R_j <> R_k)$
NE B_i, R_j, #Imm	$B_i := (R_j <> \#Imm)$

Boolean Instructions

AND B_i, B_j, B_k	$B_i := B_j$ AND B_k
OR B_i, B_j, B_k	$B_i := B_j$ OR B_k
EQ B_i, B_j, B_k	$B_i := B_j = B_k$
NE B_i, B_j, B_k	$B_i := B_j <> B_k$
GT B_i, B_j, B_k	$B_i := B_j > B_k$
LT B_i, B_j, B_k	$B_i := B_j < B_k$
LE B_i, B_j, B_k	$B_i := B_j <= B_k$
GE B_i, B_j, B_k	$B_i := B_j >= B_k$
MOV B_i, R_j	$B_i := R_j$; B_i := least significant bit of R_j;
MOV SR, R_i	SR := R_i ; Status Register contains Boolean registers.
MOV SR, #Imm	SR := #Imm

	cleared
ST (R_j, R_k), B_i	Mem[R_j + R_k] := B_i ; least significant bit of memory byte set to B_i ; other bits cleared
STB offset(R_j), R_i	Mem[offset + Rj] := R_i ; store byte only
STB (Rj, R_k), R_i	Mem[Rj + R_k] := R_i ; store byte only
STH offset(Rj), R_i	Mem[offset + Rj] := R_i; store half word only
STH (Rj, R_k), R_i	Mem[Rj + R_k] := R_i; store half word only
ST offset(Rj), R_i	Mem[offset + Rj] := R_i
ST (Rj, R_k), R_i	Mem[Rj + R_k] := R_i
STD offset(Rj), R_i	Mem[offset + Rj] := R_i: R_i +1; store double words
STD (Rj, R_k), R_i	Mem[Rj + R_k] := R_i: R_i +1
STQ offset(Rj), R_i	Mem[offset + Rj] := R_i: R_i +1: R_i +2: R_i +3; store quad words
STQ (Rj, R_k), R_i R_i	Mem[R_j + R_k] := R_i: R_i+1: R_i +2: R_i +3
STSF offset(R_j), F_i	Mem[offset + R_j] := F_i
STSF (R_j, R_k), F_i	Mem[R_j + R_k] := F_i
STDF offset(R_j), F_i	Mem[offset + R_j] := F_i; store two words
STDF (R_j, R_k), F_i	Mem[R_j + R_k] := F_i: F_i +1
STQF offset(R_j), F_i	Mem[offset + R_j] := F_i: F_i +1 ; store four words
STQF (R_j, R_k), F_i	Mem[R_j + R_k] := F_i: F_i +1
ST offset(R_j), SR	Mem[offset + R_j] := SR; 32 bits stored
ST (R_j, R_k), SR	Mem[R_j + R_k] := SR; 32 bits stored

A1.6 Branch Instructions

BT B_i, label (#delay-count)	Branch if B_i = true
BF B_i, label (#delay-count)	Branch if B_i = false
BSR R_i, label (#delay-count)	Save return address in R_i
MOV PC, R_i (#delay-count)	

A1.5 Memory Reference

LD B_i, offset(R_j)	B_i := Mem[offset + R_j] ; B_i set to lsb of memory byte
LD B_i, (R_j, R_k)	B_i := Mem[R_j + R_k] ; B_i set to lsb of memory byte
LD B_i, Label	B_i := Mem[Label]
LDB R_i, offset(R_j)	R_i := Mem[offset + R_j] ; load sign extended byte
LDB R_i, (R_j, R_k)	R_i := Mem[R_j + R_k] ; load sign extended byte
LDB R_i, Label	
LDH R_i R_i, offset(R_j)	Ri := Mem[offset + R_j] ; load sign extended half word
LDH R_i, (R_j, R_k)	Ri := Mem[R_j + R_k] ; load sign extended half word
LDH R_i, Label	
LD R_i, offset(R_j)	R_i := Mem[offset + R_j] ; load word
LD R_i, (R_j, R_k)	R_i := Mem[R_j + R_k] ; load word
LD R_i, Label	
*LDD R_i, offset(R_j)	R_i: R_i +1 := Mem[offset + R_j] ; load double words
*LDD R_i, (R_j, R_k)	R_i: R_i +1 := Mem[R_j + R_k]
*LDQ R_i, offset(R_j)	R_i: R_i +1: R_i +2: R_i +3 := Mem[offset + R_j] ; load quad words
*LDQ R_i, (R_j, R_k)	R_i: R_i +1: R_i +2: R_i +3 := Mem[R_j + R_k]
LDSF F_i, offset(R_j)	F_i := Mem[offset + R_j]; load word
LDSF F_i, (R_j, R_k)	F_i := Mem[R_j + R_k]; load word
LDSF F_i, Label	
LDDF F_i, offset(R_j)	F_i := Mem[offset + R_j] ; load double words
LDDF F_i, (R_j, R_k)	F_i := Mem[R_j + R_k]
LDDF F_i, Label	
*LDQF F_i, offset(R_j)	F_i: F_i +1 := Mem[offset + R_j] ; load quad words
*LDQF F_i, (R_j, R_k)	F_i: F_i +1 := Mem[R_j + R_k]
LD SR, offset(R_j)	SR := Mem[offset + R_j] ; load word
LD SR, (R_j, R_k)	SR := Mem[R_j + R_k] ; load word
LD SR, Label	
ST offset(R_j), B_i	Mem[offset + R_j] := B_i; least significant bit of memory byte set to B_i ; other bits

	cleared
ST (R_j, R_k), B_i	Mem[R_j + R_k] := B_i ; least significant bit of memory byte set to B_i ; other bits cleared
STB offset(R_j), R_i	Mem[offset + Rj] := R_i ; store byte only
STB (Rj, R_k), R_i	Mem[Rj + R_k] := R_i ; store byte only
STH offset(Rj), R_i	Mem[offset + Rj] := R_i; store half word only
STH (Rj, R_k), R_i	Mem[Rj + R_k] := R_i; store half word only
ST offset(Rj), R_i	Mem[offset + Rj] := R_i
ST (Rj, R_k), R_i	Mem[Rj + R_k] := R_i
STD offset(Rj), R_i	Mem[offset + Rj] := R_i: R_i +1; store double words
STD (Rj, R_k), R_i	Mem[Rj + R_k] := R_i: R_i +1
STQ offset(Rj), R_i	Mem[offset + Rj] := R_i: R_i +1: R_i +2: R_i +3; store quad words
STQ (Rj, R_k), R_i R_i	Mem[R_j + R_k] := R_i: R_i+1: R_i +2: R_i +3
STSF offset(R_j), F_i	Mem[offset + R_j] := F_i
STSF (R_j, R_k), F_i	Mem[R_j + R_k] := F_i
STDF offset(R_j), F_i	Mem[offset + R_j] := F_i; store two words
STDF (R_j, R_k), F_i	Mem[R_j + R_k] := F_i: F_i +1
STQF offset(R_j), F_i	Mem[offset + R_j] := F_i: F_i +1 ; store four words
STQF (R_j, R_k), F_i	Mem[R_j + R_k] := F_i: F_i +1
ST offset(R_j), SR	Mem[offset + R_j] := SR; 32 bits stored
ST (R_j, R_k), SR	Mem[R_j + R_k] := SR; 32 bits stored

A1.6 Branch Instructions

BT B_i, label (#delay-count)	Branch if B_i = true
BF B_i, label (#delay-count)	Branch if B_i = false
BSR R_i, label (#delay-count)	Save return address in R_i
MOV PC, R_i (#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)	

A1.7 Special Purpose Instructions

EI	Enable interrupts
DI	Disable interrupts

A1.8 Floating-point Add Unit

ADDSF F_i, F_j, F_k	$F_i := F_j + F_k$
ADDDF F_i, F_j, F_k	$F_i := F_j + F_k$
SUBSF F_i, F_j, F_k	$F_i := F_j - F_k$
SUBDF F_i, F_j, F_k	$F_i := F_j - F_k$
ABSSF F_i, F_j	If $F_j >= 0$ then $F_i := F_j$ else $F_i := - F_j$
ABSDF F_i, F_j	If $F_j >= 0$ then $F_i := F_j$ else $F_i := - F_j$
EXTF F_i, F_j	F_i (double-length) := F_j (single-length)
TRUNCF F_i, F_j	F_i (single-length) := F_j (double-length)
FLTSF F_i, F_j	F_i := FLOAT (Single length F_j); F_i is single length
FLTDF F_i, F_j	F_i := FLOAT (Single length F_j); F_i is double length
FIXSF F_i, F_j	F_i := FIX (Single length F_j); integer result
FIXDF F_i, F_j	F_i := FIX (Double length F_j); integer result

A1.9 Floating-point Multiply Unit

MULTSF F_i, F_j, F_k	$F_i := F_j$ MULT F_k
MULTDF F_i, F_j, F_k	$F_i := F_j$ MULT F_k

A1.10 Floating-point Divide Unit

DIVSF F_i, F_j, F_k	$F_i := F_j$ DIV F_k
DIVDF F_i, F_j, F_k	$F_i := F_j$ DIV F_k

A1.11 Floating-Point Relational Unit

GTSSF B_i, F_j, F_k	$B_i := (F_j > F_k)$
GTSDF B_i, F_j, F_k	$B_i := (F_j > F_k)$

GESSF B_i, F_j, F_k	$B_i := (F_j >= F_k)$
GESDF B_i, F_j, F_k	$B_i := (F_j >= F_k)$
LTSSF B_i, F_j, F_k	$B_i := (F_j < F_k)$
LTSDF B_i, F_j, F_k	$B_i := (F_j < F_k)$
LESSF B_i, F_j, F_k	$B_i := (F_j =< F_k)$
LESDF B_i, F_j, F_k	$B_i := (F_j =< F_k)$
EQSF B_i, F_j, F_k	$B_i := (F_j = F_k)$
EQDF B_i, F_j, F_k	$B_i := (F_j = F_k)$
NESF B_i, F_j, F_k	$B_i := (F_j <> F_k)$
NEDF B_i, F_j, F_k	$B_i := (F_j <> F_k)$

A1.12 Floating-Point Move Unit

MOVSF F_i, F_j	$F_i := F_j$
MOVDF F_i, F_j	$F_i := F_j$
MOVSF R_i, F_j	Ri := Single-length F_j; integer result

*Indicates those instructions not yet implemented.

A1.13 Assembly Language Abreviations

The following instructions are used as convenient assembly language abreviations for more complex HSP instructions. These instructions are not present as distinct HSP instructions. They are simply shortened versions of existing instructions which are used to inprove clarity at the assembly language level.

The extended forms of these instructions often rely on the fact that R0 is always zero and that B0 is always false.

MOV Ri, #imm	ADD R_i, R0, #imm
MOV R_i, R_j	ADD R_i, R_j, R0
MOV B_i, B_j	OR B_i, B0, B_j
CLR R_i	ADD R_i, R0, R0
NEG R_i , R_j	SUB R_i, R0, R_j
NOT R_i , R_j	EOR R_i, R_j, #-1
BRA label	BF B0, label

Appendix A.2 Combined Instructions

HSA will also support combined instructions. In general these will have the form:

Dst := F(G(src1,src2),src3) or Dst := F(src1,G(src2,src3))

The syntax adopted follows HARP syntax as closely as possible using infix notation to improve readability.

Our latest implementation ideas involve marking pairs of instructions as being combined. We are therefore no longer contrained by the number of instruction bit patterns available in 32bits. I have therefore used both formats and gone for orthogonality. My objective is to simplify the programming in both the scheduler and the simulator by minimising the requirement for special cases.

B.1 Computational (ALU) Instructions

First Instruction	Second Instruction	Combined Instruction
ADD Ri, Rj, Rk	ADD Rm, Ri, Rn	ADD Rm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj + Rk)
ADD Ri, Rj, Rk	ADD Rm, Ri, #Imm	ADD Rm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm	ADD Rm, Ri, Rn	ADD Rm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj + #Imm)
ADD Ri, Rj, #Imm	ADD Rm, Ri, #Imm	Reduces to two operand instruction
ADD Ri, Rj, Rk	SUB Rm, Ri, Rn	SUB Rm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj + Rk)
ADD Ri, Rj, Rk	SUB Rm, Ri, #Imm	SUB Rm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm	SUB Rm, Ri, Rn	SUB Rm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj + #Imm)
ADD Ri, Rj, #Imm	SUB Rm, Ri, #Imm	Reduces to two operand instruction
SUB Ri, Rj, Rk	ADD Rm, Ri, Rn	ADD Rm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj - Rk)
SUB Ri, Rj, Rk	ADD Rm, Ri, #Imm	ADD Rm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm	ADD Rm, Ri, Rn	ADD Rm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj - #Imm)
SUB Ri, Rj, #Imm	ADD Rm, Ri, #Imm	Reduces to two operand instruction
SUB Ri, Rj, Rk	SUB Rm, Ri, Rn	SUB Rm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj - Rk)
SUB Ri, Rj, Rk	SUB Rm, Ri, #Imm	SUB Rm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm	SUB Rm, Ri, Rn	SUB Rm, (Rj - #Imm), Rn

SUB Ri, Rj, #Imm	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj - #Imm)
SUB Ri, Rj, #Imm	SUB Rm, Ri, #Imm	Reduces to two operand instruction
ASL Ri, Rj, Rk	ADD Rm, Ri, Rn	ADD Rm, (Rj ASL Rk), Rn
ASL Ri, Rj, Rk	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj ASL Rk)
ASL Ri, Rj, Rk	ADD Rm, Ri, #Imm	ADD Rm, (Rj ASL Rk), #Imm
ASL Ri, Rj, #Imm	ADD Rm, Ri, Rn	ADD Rm, (Rj ASL #Imm), Rn
ASL Ri, Rj, #Imm	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj ASL #Imm)
ASL Ri, Rj, #Imm1	ADD Rm, Ri, #Imm2	ADD Rm, (Rj ASL #Imm1), #Imm2
ASL Ri, Rj, Rk	SUB Rm, Ri, Rn	SUB Rm, (Rj ASL Rk), Rn
ASL Ri, Rj, Rk	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj ASL Rk)
ASL Ri, Rj, Rk	SUB Rm, Ri, #Imm	SUB Rm, (Rj ASL Rk), #Imm
ASL Ri, Rj, #Imm	SUB Rm, Ri, Rn	SUB Rm, (Rj ASL #Imm), Rn
ASL Ri, Rj, #Imm	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj ASL #Imm)
ASL Ri, Rj, #Imm1	SUB Rm, Ri, #Imm2	SUB Rm, (Rj ASL #Imm1), #Imm2

B.2 Relational Unit Instructions

First Instruction	Second Instruction	Combined Instruction
Signed		
ADD Ri, Rj, Rk	GTS Bm, Ri, Rn	GTS Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	GTS Bm, Rn, Ri	GTS Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	GTS Bm, Ri, Rn	GTS Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	GTS Bm, Rn, Ri	GTS Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	GTS Bm, Ri, #Imm	GTS Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	GTS Bm, Ri, #Imm2	GTS Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	GTS Bm, Ri, Rn	GTS Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	GTS Bm, Rn, Ri	GTS Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	GTS Bm, Ri, Rn	GTS Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	GTS Bm, Rn, Ri	GTS Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	GTS Bm, Ri, #Imm	GTS Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	GTS Bm, Ri, #Imm2	GTS Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	GES Bm, Ri, Rn	GES Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	GES Bm, Rn, Ri	GES Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	GES Bm, Ri, Rn	GES Bm, (Rj + #Imm), Rn

ADD Ri, Rj, #Imm	GES Bm, Rn, Ri	GES Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	GES Bm, Ri, #Imm	GES Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	GES Bm, Ri, #Imm2	GES Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	GES Bm, Ri, Rn	GES Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	GES Bm, Rn, Ri	GES Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	GES Bm, Ri, Rn	GES Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	GES Bm, Rn, Ri	GES Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	GES Bm, Ri, #Imm	GES Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	GES Bm, Ri, #Imm2	GES Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	LTS Bm, Ri, Rn	LTS Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	LTS Bm, Rn, Ri	LTS Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	LTS Bm, Ri, Rn	LTS Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	LTS Bm, Rn, Ri	LTS Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	LTS Bm, Ri, #Imm	LTS Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	LTS Bm, Ri, #Imm2	LTS Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	LTS Bm, Ri, Rn	LTS Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	LTS Bm, Rn, Ri	LTS Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	LTS Bm, Ri, Rn	LTS Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	LTS Bm, Rn, Ri	LTS Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	LTS Bm, Ri, #Imm	LTS Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	LTS Bm, Ri, #Imm2	LTS Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	LES Bm, Ri, Rn	LES Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	LES Bm, Rn, Ri	LES Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	LES Bm, Ri, Rn	LES Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	LES Bm, Rn, Ri	LES Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	LES Bm, Ri, #Imm	LES Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	LES Bm, Ri, #Imm2	LES Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	LES Bm, Ri, Rn	LES Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	LES Bm, Rn, Ri	LES Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	LES Bm, Ri, Rn	LES Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	LES Bm, Rn, Ri	LES Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	LES Bm, Ri, #Imm	LES Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	LES Bm, Ri, #Imm2	LES Bm, (Rj - #Imm1), #Imm2

Unsigned

ADD Ri, Rj, Rk	GTU Bm, Ri, Rn	GTU Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	GTU Bm, Rn, Ri	GTU Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	GTU Bm, Ri, Rn	GTU Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	GTU Bm, Rn, Ri	GTU Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	GTU Bm, Ri, #Imm	GTU Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	GTU Bm, Ri, #Imm2	GTU Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	GTU Bm, Ri, Rn	GTU Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	GTU Bm, Rn, Ri	GTU Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	GTU Bm, Ri, Rn	GTU Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	GTU Bm, Rn, Ri	GTU Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	GTU Bm, Ri, #Imm	GTU Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	GTU Bm, Ri, #Imm2	GTU Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	GEU Bm, Ri, Rn	GEU Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	GEU Bm, Rn, Ri	GEU Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	GEU Bm, Ri, Rn	GEU Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	GEU Bm, Rn, Ri	GEU Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	GEU Bm, Ri, #Imm	GEU Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	GEU Bm, Ri, #Imm2	GEU Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	GEU Bm, Ri, Rn	GEU Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	GEU Bm, Rn, Ri	GEU Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	GEU Bm, Ri, Rn	GEU Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	GEU Bm, Rn, Ri	GEU Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	GEU Bm, Ri, #Imm	GEU Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	GEU Bm, Ri, #Imm2	GEU Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	LTU Bm, Ri, Rn	LTU Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	LTU Bm, Rn, Ri	LTU Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	LTU Bm, Ri, Rn	LTU Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	LTU Bm, Rn, Ri	LTU Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	LTU Bm, Ri, #Imm	LTU Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	LTU Bm, Ri, #Imm2	LTU Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	LTU Bm, Ri, Rn	LTU Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	LTU Bm, Rn, Ri	LTU Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	LTU Bm, Ri, Rn	LTU Bm, (Rj - #Imm), Rn

SUB Ri, Rj, #Imm	LTU Bm, Rn, Ri	LTU Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	LTU Bm, Ri, #Imm	LTU Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	LTU Bm, Ri, #Imm2	LTU Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	LEU Bm, Ri, Rn	LEU Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	LEU Bm, Rn, Ri	LEU Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	LEU Bm, Ri, Rn	LEU Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	LEU Bm, Rn, Ri	LEU Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	LEU Bm, Ri, #Imm	LEU Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	LEU Bm, Ri, #Imm2	LEU Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	LEU Bm, Ri, Rn	LEU Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	LEU Bm, Rn, Ri	LEU Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	LEU Bm, Ri, Rn	LEU Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	LEU Bm, Rn, Ri	LEU Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	LEU Bm, Ri, #Imm	LEU Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	LEU Bm, Ri, #Imm2	LEU Bm, (Rj - #Imm1), #Imm2

Signed and Unsigned

ADD Ri, Rj, Rk	EQ Bm, Ri, Rn	EQ Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	EQ Bm, Rn, Ri	EQ Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	EQ Bm, Ri, Rn	EQ Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	EQ Bm, Rn, Ri	EQ Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	EQ Bm, Ri, #Imm	EQ Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	EQ Bm, Ri, #Imm2	EQ Bm, (Rj + #Imm1), #Imm2
SUB Ri, Rj, Rk	EQ Bm, Ri, Rn	EQ Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	EQ Bm, Rn, Ri	EQ Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	EQ Bm, Ri, Rn	EQ Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	EQ Bm, Rn, Ri	EQ Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	EQ Bm, Ri, #Imm	EQ Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	EQ Bm, Ri, #Imm2	EQ Bm, (Rj - #Imm1), #Imm2
ADD Ri, Rj, Rk	NE Bm, Ri, Rn	NE Bm, (Rj + Rk), Rn
ADD Ri, Rj, Rk	NE Bm, Rn, Ri	NE Bm, Rn, (Rj + Rk)
ADD Ri, Rj, #Imm	NE Bm, Ri, Rn	NE Bm, (Rj + #Imm), Rn
ADD Ri, Rj, #Imm	NE Bm, Rn, Ri	NE Bm, Rn, (Rj + #Imm)
ADD Ri, Rj, Rk	NE Bm, Ri, #Imm	NE Bm, (Rj + Rk), #Imm
ADD Ri, Rj, #Imm1	NE Bm, Ri, #Imm2	NE Bm, (Rj + #Imm1), #Imm2

SUB Ri, Rj, Rk	NE Bm, Ri, Rn	NE Bm, (Rj - Rk), Rn
SUB Ri, Rj, Rk	NE Bm, Rn, Ri	NE Bm, Rn, (Rj - Rk)
SUB Ri, Rj, #Imm	NE Bm, Ri, Rn	NE Bm, (Rj - #Imm), Rn
SUB Ri, Rj, #Imm	NE Bm, Rn, Ri	NE Bm, Rn, (Rj - #Imm)
SUB Ri, Rj, Rk	NE Bm, Ri, #Imm	NE Bm, (Rj - Rk), #Imm
SUB Ri, Rj, #Imm1	NE Bm, Ri, #Imm2	NE Bm, (Rj - #Imm1), #Imm2

B.3 Shift Unit

First Instruction	Second Instruction	Combined Instruction
ASL Ri, Rj, Rk	AND Rm, Ri, Rn	AND Rm, (Rj ASL Rk), Rn
ASL Ri, Rj, Rk	AND Rm, Rn, Ri	AND Rm, Rn, (Rj ASL Rk)
ASL Ri, Rj, #Imm	AND Rm, Ri, Rn	AND Rm, (Rj ASL #Imm), Rn
ASL Ri, Rj, #Imm	AND Rm, Rn, Ri	AND Rm, Rn, (Rj ASL #Imm),
ASL Ri, Rj, Rk	AND Rm, Ri, #Imm	AND Rm, (Rj ASL Rk), #Imm
ASL Ri, Rj, #Imm1	AND Rm, Ri, #Imm2	AND Rm, (Rj ASL #Imm1), Imm2
ASR Ri, Rj, Rk	AND Rm, Ri, Rn	AND Rm, (Rj ASR Rk), Rn
ASR Ri, Rj, Rk	AND Rm, Rn, Ri	AND Rm, Rn, (Rj ASR Rk)
ASR Ri, Rj, #Imm	AND Rm, Ri, Rn	AND Rm, (Rj ASR #Imm), Rn
ASR Ri, Rj, #Imm	AND Rm, Rn, Ri	AND Rm, Rn, (Rj ASR #Imm),
ASR Ri, Rj, Rk	AND Rm, Ri, #Imm	AND Rm, (Rj ASR Rk), #Imm
ASR Ri, Rj, #Imm1	AND Rm, Ri, #Imm2	AND Rm, (Rj ASR #Imm1), Imm2
ASL Ri, Rj, Rk	OR Rm, Ri, Rn	OR Rm, (Rj ASL Rk), Rn
ASL Ri, Rj, Rk	OR Rm, Rn, Ri	OR Rm, Rn, (Rj ASL Rk)
ASL Ri, Rj, #Imm	OR Rm, Ri, Rn	OR Rm, (Rj ASL #Imm), Rn
ASL Ri, Rj, #Imm	OR Rm, Rn, Ri	OR Rm, Rn, (Rj ASL #Imm),
ASL Ri, Rj, Rk	OR Rm, Ri, #Imm	OR Rm, (Rj ASL Rk), #Imm
ASL Ri, Rj, #Imm1	OR Rm, Ri, #Imm2	OR Rm, (Rj ASL #Imm1), Imm2
ASR Ri, Rj, Rk	OR Rm, Ri, Rn	OR Rm, (Rj ASR Rk), Rn
ASR Ri, Rj, Rk	OR Rm, Rn, Ri	OR Rm, Rn, (Rj ASR Rk)
ASR Ri, Rj, #Imm	OR Rm, Ri, Rn	OR Rm, (Rj ASR #Imm), Rn
ASR Ri, Rj, #Imm	OR Rm, Rn, Ri	OR Rm, Rn, (Rj ASR #Imm),
ASR Ri, Rj, Rk	OR Rm, Ri, #Imm	OR Rm, (Rj ASR Rk), #Imm
ASR Ri, Rj, #Imm1	OR Rm, Ri, #Imm2	OR Rm, (Rj ASR #Imm1), Imm2

B.4 Multiply Unit

First Instruction	Second Instruction	Combined Instruction
MULT Ri, Rj, Rk	ADD Rm, Ri, Rn	ADD Rm, (Rj * Rk), Rn
MULT Ri, Rj, Rk	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj * Rk)
MULT Ri, Rj,#Imm	ADD Rm, Ri, Rn	ADD Rm, (Rj * #Imm), Rn
MULT Ri, Rj, #Imm	ADD Rm, Rn, Ri	ADD Rm, Rn, (Rj * #Imm)

MULT Ri, Rj, Rk	ADD Rm, Ri, #Imm	ADD Rm, (Rj *Rk), #Imm
MULT Ri, Rj, #Imm1	ADD Rm, Ri, #Imm2	ADD Rm, (Rj * #Imm1), #Imm2
MULT Ri, Rj, Rk	SUB Rm, Ri, Rn	SUB Rm, (Rj * Rk), Rn
MULT Ri, Rj, Rk	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj * Rk)
MULT Ri, Rj,#Imm	SUB Rm, Ri, Rn	SUB Rm, (Rj * #Imm), Rn
MULT Ri, Rj, #Imm	SUB Rm, Rn, Ri	SUB Rm, Rn, (Rj * #Imm)
MULT Ri, Rj, Rk	SUB Rm, Ri, #Imm	SUB Rm, (Rj *Rk), #Imm
MULT Ri, Rj, #Imm1	SUB Rm, Ri, #Imm2	SUB Rm, (Rj * #Imm1), #Imm2

B.5 Memory Reference

No examples

B.6 Boolean Instructions

No examples

B.7 Branch Instructions

All signed

First Instruction	**Second Instruction**	**Combined Instruction**
EQ Bi, Rj, #0	BT, Bi, label (#delay-count)	BEQ Ri, label (#delay-count)
NE Bi, Rj, #0	BT, Bi, label (#delay-count)	BNE Ri , label (#delay-count)
GTS Bi, Rj, #0	BT, Bi, label (#delay-count)	BGTS Ri, label (#delay-count)
GES Bi, Rj, #0	BT, Bi, label (#delay-count)	BGES Ri, label (#delay-count)
LTS Bi, Rj, #0	BT, Bi, label (#delay-count)	BLTS Ri, label (#delay-count)
LES Bi, Rj, #0	BT, Bi, label (#delay-count)	BLES Ri, label (#delay-count)

B.8 Special Purpose

No examples

B.9 Floating-point

No examples

Appendix B HSS Configuration Parameters

```
/*hsp_const.h*/
/*constant definitions for HSP Scheduler*/

#define true 1
#define false 0
#define dummy -9

#define TENPIPE
#define HADES_OPTION false
#define HADES2_OPTION false /*allows pairing of unscheduled twopipe code*/
#define BRANCHINLINE NO /*allows branch to be moved irrespective of slot no.*/


#define LSIZE 160 /*maximum number of characters per line*/
#define TOKEN_SIZE 20 /*maximum number of characters per token*/
#define LINECOUNT 3
#define opcode_length 10 /*maximum number of characters per opcode/function*/


#define BOOLEAN_REGS 16
#define ROTATEBOOLS true /*change input code to use all available bools*/
#define MAX_GUARDS 1 /*maximum guards per instruction*/
#define BR_MAX_GUARDS 2 /*maximum guards per branch instruction*/
#define PERCOLATION_THRESHOLD 100 /*%age of successful node required*/
#define BACKEDGE_THRESHOLD 100      /*before percolation succeeds*/
#define BACKEDGEPARAM true /*enable percolation across backedge*/
#define PERC_BACKEDGE_COUNT 1000 /*determines how many loops will get code
                                   percolated across backedges*/
#define BACKEDGE_ITERATIONS 1000 /*Maximum number of iterations of backedge
                                   algorithm for each loop*/
#define EXTRA_COPY 2
#define Renaming true
#define global_scheduling true
#define BB_duplication true
#define instruction_issue_rate 16
#define instruction_fetch_rate 16

#define MAX_SLOTS instruction_issue_rate
#define ICACHE_CYCLES 2
#define DELAY_SLOTS ICACHE_CYCLES
#define VCOPY_ON false

#define MOV_MERGE true
#define LD_MOV_MERGE false /*allows LD to merge with following MOV*/
#define PERCOLATE_MOVS true /*percolate MOVs inserted during renaming*/
#define GLOBAL_MOV_PERCOLATION true /*percolate MOVs from initial BB*/
        /*note MOVs are percolated across back edges with either setting*/
#define PERCOLATE_BRANCH true /*percolate branch at end of BB percolation*/
#define REC_GLOBAL_PERCOLATION true
#define ADDGUARDTOREMOVEOUTPUTDEPENDENCE false
#define INTERPROCPERC true

#define ARITH_UNIT 16
#define MULT_UNIT 16
#define REL_UNIT 16
```

```
#define MEM_REF_UNIT 4
#define LD_UNIT 2
#define ST_UNIT 2
#define BR_UNIT 3
#define FP_ADD_UNIT 2
#define FP_MULT_UNIT 2
#define FP_REL_UNIT 2

#define FAIL 0
#define SUCCEED 1
#define LIWDELETED 2
#define YES 1
#define NO 0
#define RENAME 1
#define NORENAME 0

#define RENAMEIFREMOVEGUARD 2
#define RENAMEIFNOTREGSAVED NO /*if reg not in LD and ST*/
#define INSERTATBEGINNING 2
#define RENAMEIFPRELUDE NO
#define ALLREGSPRESERVED NO
#define PERCOLATEINTOBSR YES
#define PERCOLATEINTOCONDBRANCH YES
#define DUPLICATEDELAYSLOTS YES
#define PRINTMASKS NO


#define Pred_SS 0
#define Pred_BT 1

#define labellength 150
#define condlength 4
#define reglength 4
#define reg_locations 2
#define BSR_upward_reg 0x0000ffff
#define BSR_bool_upward_reg 0x00000001
#define inst_no 5

/*****************************************************************************/
/*debugging constants*/

#define DEBUG true /*will - eventually - disable debugging print statements when false*/
#define PRINTBRANCHTARGETS false /*conditionally prints branch targets when
                                   perc_count expires*/
#define PRINTBBSCHEDULES false /*conditionally prints BB schedules when
                                   perc_count expires*/
/*****************************************************************************/
/*constants for latency representation*/

#define INTOPSLATENCY false
#define MULT 3
#define DIV 16
#define LD 2
#define ST 1
#define INTOPS 2
```

```
/*****************************************************************************/

/*mask stuff*/

#define int_proc_live_regs 0x0000ffff
#define float_proc_live_regs 0x0000ffff
#define bool_proc_live_regs 0x00000001

/*register details*/

#define SP 2
#define FIRSTREGSAVED 16
#define LASTREGSAVED 31
#define FIRSTRESERVEDREG 0
#define LASTRESERVEDREG 4
#define FIRSTPARAMREG 5
#define LASTPARAMREG 15
#define RESULTREG 5
#define INT_REGS 32
#define FIRSTBOOLRESERVEDREG 0

/*****************************************************************************/
/*Inlining Parameters*/
/*To view inlined code only in *cd.ps, set exit_count to 0 in hsp_backedge.c*/

#define Inline_Proc_Size1_Threshold 5
/*proc called from withon a loop will only be inlined if number of basic blocks in proc
is less than or equal to this threshold*/

#define Inline_No_Of_Calls1_Threshold 0
/*proc called from within a loop will only be inlined if static number of calls to proc is less
than or equal to this threshold*/

#define Inline_Proc_Size2_Threshold 5
/*proc called from outside a loop will only be inlined if number of basic blocks in proc
is less than or equal to this threshold*/

#define Inline_No_Of_Calls2_Threshold 1
/*proc called from outside a loop will only be inlined if static number of calls to the proc
is less than or equal to this threshold*/

#define Inline_Proc_Size3_Threshold 5
/*proc called from within a recursive proc will only be inlined if number of basic blocks
in proc is less than or equal to this threshold*/

#define Inline_Recursive_Calls false
/*if true directly recursive calls are inlined.  If false they are not*/

#define Inline_Nesting_Threshold 5
/*an inlined procedure may itself contain proc calls which may themselves be in turn inlined.
To avoid endless inlining, the number of nested inlinings must not exceed this threshold*/

#define INLINEPARAM true
/*if true inline, if false do not inline*/
```

```
#define UNCALLEDPROCSDELETED true
/*if true delete uncalled procs, if false do not*/
```

Appendix C Loop Scheduling

Example: Original Code

Loop:

```
LD R16, (R1)
ADD R16, R16, #17
ST (R3), R16
ADD R1, R1, #4;
ADD R3, R3, #4
SUB R2, R2, #1
NE B1, R2, #0
BT B1, Loop
```

First Pass

Schedule each instruction in turn, starting at the top of the loop. Move or percolate each instruction as high as possible.

Loop:

```
LD R16, (R1); ADD R1, R1, #4; ADD R4, R3, #4;
        SUB R2, R2, #1
ADD R16, R16, #17;  NE B1, R2, #0
ST (R3), R16; MOV R3, R4; BT B1, Loop
```

Note renaming of R3 in third addition.
Each loop iteration now requires 3 cycles instead of 7.

Result of First Pass Repeated

Loop:

```
LD R16, (R1); ADD R1, R1, #4; ADD R4, R3, #4;
        SUB R2, R2, #1
ADD R16, R16, #17;  NE B1, R2, #0
ST (R3), R16; MOV R3, R4; BT B1, Loop
```

Start of Second Pass

Percolate instructions from the first instruction group round the loop back edge. There must be chain of dependencies from the percolating instruction to end of loop. Instructions are not allowed to percolate back into the first group.

Loop prelude:

```
    LD R17, (R1)
```

Loop1:

```
MOV R16, R17;  ADD R1, R1, #4;  ADD R4, R3, #4;
        SUB R2, R2, #1
ADD R16, R16, #17;   NE B1, R2, #0;  LD R17, (R1)
ST (R3), R16;  MOV R3, R4;  BT B1, Loop1
```

Note: The disambiguation module on this scheduler is particularly good and has determined that R1 != R3.

```
Loop prelude:
        LD R17, (R1)
Loop1:
        MOV R16, R17;  ADD R1, R1, #4;  ADD R4, R3, #4;
                SUB R2, R2, #1
        ADD R16, R16, #17;   NE B1, R2, #0;  LD R17, (R1)
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1
```

Consider ADD R1, R1, #4.

No chain of dependencies, so no percolation attempted.

```
Loop prelude:
        LD R17, (R1)
Loop1:
        MOV R16, R17;  ADD R1, R1, #4;  ADD R4, R3, #4;
                SUB R2, R2, #1
        ADD R16, R16, #17;   NE B1, R2, #0;  LD R17, (R1)
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1
```

Percolate ADD R4, R3, #4

No chain of dependencies, so no percolation attempted.

```
Loop prelude:
        LD R17, (R1)
Loop1:
        MOV R16, R17;  ADD R1, R1, #4;  ADD R4, R3, #4;
                SUB R2, R2, #1
        ADD R16, R16, #17;   NE B1, R2, #0;  LD R17, (R1)
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1
```

Percolate SUB R2, R2, #1

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1
Loop1:
        MOV R16, R17;  ADD R1, R1, #4;  ADD R4, R3, #4;
        ADD R16, R16, #17;   NE B1, R2, #0;  LD R17, (R1);
                SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1
```

RESCHEDULE LOOP WITH BACK EDGE DISABLED

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1
Loop1:
        MOV R16, R17;  ADD R1, R1, #4;  ADD R4, R3, #4;
        ADD R16, R16, #17;   NE B1, R2, #0;  LD R17, (R1);
                SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1
```

Compacted code:

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1
Loop1:
        ADD R1, R1, #4;  ADD R4, R3, #4; ADD R16, R17, #17;
                NE B1, R2, #0; SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1
```

The loop body has now been reduced to two instruction groups.

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1
Loop1:
        ADD R1, R1, #4;  ADD R4, R3, #4; ADD R16, R17, #17;
                NE B1, R2, #0; SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1
```

Percolate ADD R1, R1, #4;

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17;  NE B1, R2, #0;
                SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4
```

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17;  NE B1, R2, #0;
                SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4
```

Percolate ADD R4, R3, #4

No chain of dependencies so percolation fails.

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17;  NE B1, R2, #0;
                SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4
```

Percolate ADD R16, R17, #17

Percolation fails.

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17;  NE B1, R2, #0;
                SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
```

```
        BT B1, Loop1; ADD R1, R1, #4
```

Percolate NE B1, R2, #0

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17; SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4; NE B1, R2, #0
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17; SUB R2, R2, #1
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4; NE B1, R2, #0
```

Percolate SUB R2, R2, #1

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4; NE B1, R2, #0;
                SUB R2, R2, #1
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17
        LD R17, (R1); ST (R3), R16;  MOV R3, R4;
                BT B1, Loop1; ADD R1, R1, #4; NE B1, R2, #0;
                SUB R2, R2, #1
```

Compact loop

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17; LD R17, (R1);
                ADD R1, R1, #4; NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2
```

Loop has not been shortened but since some percolation took place across back edge process must be repeated.

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17; LD R17, (R1);
                ADD R1, R1, #4; NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2
```

Percolate ADD R4, R3, #4

No dependent chain, so percolation not attempted.

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1
Loop1:
        ADD R4, R3, #4; ADD R16, R17, #17; LD R17, (R1);
                ADD R1, R1, #4; NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2
```

Percolate ADD R16, R17, #17

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17
```

```
Loop1:
        ADD R4, R3, #4; LD R17, (R1); ADD R1, R1, #4;
                NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2; ADD R16, R17, #17

Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17
Loop1:
        ADD R4, R3, #4; LD R17, (R1); ADD R1, R1, #4;
                NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2; ADD R16, R17, #17
```

Percolate LD R17, (R1)

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
                LD R17, (R1)
Loop1:
        ADD R4, R3, #4; ADD R1, R1, #4; NE B2, R2, #0;
                SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2; ADD R16, R17, #17; LD R17, (R1)
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
                LD R17, (R1)
Loop1:
        ADD R4, R3, #4; ADD R1, R1, #4; NE B2, R2, #0;
                SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2; ADD R16, R17, #17; LD R17, (R1)
```

Percolate ADD R1, R1, #4

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
```

```
            LD R17, (R1); ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1,B2; ADD R16, R17, #17; LD R17, (R1);
                ADD R1, R1, #4

Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
                LD R17, (R1); ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1, B2; ADD R16, R17, #17; LD R17, (R1);
                ADD R1, R1, #4
```

Percolate NE B2, R2, #0

No critical chain so percolation not attempted. Note chain could pass through antidependence!

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
                LD R17, (R1); ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; NE B2, R2, #0; SUB R2, R2, #1
        ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
                MOV B1, B2; ADD R16, R17, #17; LD R17, (R1);
                ADD R1, R1, #4
```

Percolate SUB R2, R2, #1

No critical chain so percolation not attempted.

```
Loop prelude:
        LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
        NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
                LD R17, (R1); ADD R1, R1, #4
Loop1:
        ADD R4, R3, #4; NE B2, R2, #0; SUB R2, R2, #1
```

```
ST (R3), R16;  MOV R3, R4;  BT B1, Loop1;
        MOV B1, B2; ADD R16, R17, #17; LD R17, (R1);
        ADD R1, R1, #4
```

Compact loop with back edge disabled

```
Loop prelude:
    LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
    NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
        LD R17, (R1); ADD R1, R1, #4
Loop1:
    NE B2, R2, #0; SUB R2, R2, #1; ST (R3), R16;
        ADD R3, R3, #4; ADD R16, R17, #17; LD R17, (R1);
        ADD R1, R1, #4
    BT B1, Loop1; MOV B1, B2
```

```
Loop prelude:
    LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
    NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
        LD R17, (R1); ADD R1, R1, #4
Loop1:
    NE B2, R2, #0; SUB R2, R2, #1; ST (R3), R16;
        ADD R3, R3, #4; ADD R16, R17, #17; LD R17, (R1);
        ADD R1, R1, #4
    BT B1, Loop1; MOV B1, B2
```

We are left with the above schedule with a Boolean copy instruction left in the final group. This copy is the result of an earlier "unnecessary" promotion of the NE instruction. The final code required is given below, where the NE has been combined with the branch & inserted after the branch instruction.

```
Loop prelude:
    LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4
    NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
        LD R17, (R1); ADD R1, R1, #4
Loop1:
    SUB R2, R2, #1; ST (R3), R16; ADD R3, R3, #4;
        ADD R16, R17, #17; LD R17, (R1); ADD R1, R1, #4
        BT B1, Loop1; NE B1, R2, #0;
```

Appendix D

HSS comprises currently 20 modules and approximately 12,000 lines of code. The modules are listed below:

hsp_backedge.c	hsp_basicblocknode.c	hsp_checkcoexist.c
hsp_checkpass.c	hsp_code_motion.c	hsp_code_motion1.c
hsp_const.h	hsp_declare.h	hsp_disambiguate.c
hsp_inline.c	hsp_instructionnode.c	hsp_LIWnode.c
hsp_loopnode.c	hsp_mask_handling.c	hsp_type.h
hsp_merge_instructions.c	hsp_procnode.c	hsp_rotate_bools.c
hsp_sched.c	hsp_sched1.c	hsp_symboltable.c

The data structure definitions are in the file **hsp_type.h.** These are as follows:

typedef struct instnode
typedef struct percolatinginstnode
typedef struct LIWnode
typedef struct branchtargetlist
typedef struct loopexitlist
tyepdef struct loopbblist
typedef struct BBpercolationfailsnode
typedef struct listofbranches
typedef struct guardrecordlist
typedef struct storeguardinfo
typedef struct pathrecord
typedef struct VCOPYinfo
tyepdef struct func_alias
typedef struct insertionlistnode
typedef struct basicblocknode
typedef struct symboltableentry
typedef struct listnode
typedef struct looptaillist
typedef struct taillist
typedef struct copylistnode
typedef struct percolationnodes
typedef struct renamemask
typedef struct LIWinfo
typedef struct perclistnode
typedef struct insertinfo