McGill University
School of Computer Science

SuperDLX
A Generic Superscalar Simulator

ACAPS Technical Memo 64
April 13, 1993

Cecile Moura

Advanced Compilers, Architectures and
Parallel Systems Group

3480 University St • Montreal • Canada • H3A 2A7

Contents

1 Introduction **1**

1.1 Superscalar Processors – An Overview 1

1.2 SuperDLX - A Superscalar Simulator 5

1.3 Document Organization 6

2 Overview of the Execution Model **7**

2.1 The Processor Model 7

2.2 The Different Pipeline Stages 9

2.2.1 Fetch Stage 9

2.2.2 Decode Stage 10

2.2.3 Execute Stage 11

2.2.4 Write Back Stage 12

2.2.5 Result Commit Stage 12

3 Underlying Data Structures **13**

3.1 Instruction Tables 13

3.2 Basic Processor Elements 15

3.2.1 Memory 15

3.2.2 Register File 15

3.2.3 Functional Units 15

3.3 Superscalar Elements 17

3.3.1 Branch Target Buffer 17

3.3.2 Instruction Fetch Queue 18

3.3.3 Instruction Windows 19
3.3.4 Reorder Buffers 20
3.3.5 Load and Store Buffers 21
3.4 A Running Example 23

4 Detailed Functional Description **24**
4.1 Overall Functional Organization 24
4.2 Fetch Function 24
4.3 Decode Functions 28
4.3.1 Creating an Instruction Window Entry 28
4.3.2 Operand Searching 30
4.3.3 Creating a Reorder Buffer Entry 30
4.4 Execute Functions 31
4.4.1 Instruction Issue 31
4.4.2 Instruction Selection 32
4.4.3 Instruction Execution 32
4.4.4 Memory Access 34
4.5 Write Back Functions 36
4.6 Result Commit Functions 37

5 User Interface Description **38**
5.1 Overall Functional Organization 38
5.2 User Parameters 39
5.3 Processing User Commands 41

5.3.1 Load Command 41
5.3.2 Go Command 41
5.3.3 Step Command 43
5.3.4 Next Command 43
5.3.5 Print Command 44
5.3.6 Stats Command 44
5.3.7 Reset Command 44

6 Future Directions 45

A User Manual 47

B The Opcode Description Table 53

C An Example of a Machine Configuration File 56

D An Example of Statistical Output 58

E A Running Example 63

F Prototype of a SuperDLX X Window Interface 72

List of Figures

1 The Processor Model 8

2 State Diagram for the Branch Predictor 18

3 Keeping Instructions in Order in the Central Instruction Window 19

4 Reorder Buffer Entries 21

5 Load and Store Processing 23

6 Overall Functional Diagram 25

7 Functions Coordination 26

8 User Interface Functional Diagram 40

9 X-Interface Front Panel 72

10 Control Menu 73

11 Step Command Panel 74

12 Load Command Panel 74

13 Floating Point Instruction Window Display 75

14 Floating Point Reorder Buffer Display 75

15 Machine Configuration Parameters 76

Abstract

The objective of this document is to present superDLX, a cycle by cycle superscalar simulator using the DLX instruction set. The simulator was developed as a Master's project, under the supervision of Professor Gao, for the ACAPS (Advanced Compiler, Architectures and Parallel Systems) laboratory of McGill University.

This superscalar simulator achieves the most aggressive instruction issue policy: out-of-order issue with out-of-order completion. To implement such instruction processing, several powerful hardware mechanisms were selected: an instruction window, from where instructions issue out of order to multiple functional units; a reorder buffer, where storage conflicts are resolved. To sustain the processor execution rate, branch prediction is performed, which enables the issue logic to dynamically schedule instructions past conditional branches. Also, memory access is optimized, with the use of load and store buffers.

The first part of this document outlines some superscalar concepts that have been dealt within this project. The rest of the report is then devoted to the description of the implemented simulator: the superscalar processor model that was chosen, the data structures that represent the different hardware elements, the functions that simulate the pipeline stages, and the user interface. Finally, the last chapter gives some outlines for areas of improvements for the simulator.

1 Introduction

1.1 Superscalar Processors – An Overview

Superscalar processors are hardware organizations capable of executing several instructions per clock cycle. They may be considered as the most promising uniprocessor architectures of the post RISC era. Some commercial superscalar architectures have appeared on the market: for example, the IBM RS/6000 [Gro90], the intel80960 [McG90], the Motorola 88110 [AD92]. Another notable architecture has been implemented , but is not commercialized: the Metaflow Lightning SPARC [BD91].

Although they can be viewed as an evolution of RISC architectures, superscalar architectures are subject to many more trade-offs than simply widening the pipeline. The term superscalar encompasses a variety of mechanisms and designs and finding a standardized description of superscalar architectures is not easy. We will try to highlight some of the major concepts surrounding superscalar processors that have been tackled within this project. For an in-depth descriptions of superscalar issues, a very interesting reference is Mike Johnson's thesis [Joh91].

1. **Instruction Issue Policies**

   In a superscalar machine, pipeline stages are concurrent processes that may take care of a variable number of instructions per cycle. The overall coordination between processes, as well as their degrees of freedom, have a significant impact on the processor ability to exploit the instruction parallelism available in a program. In particular, instruction issue, which is the process of letting an intruction move from the decode stage to the execute stage, determines the processor capacity to dicover instructions that can be executed concurrently.

   There are three policies for instruction issue as detailed below:

   - **In-order issue, in-order completion**

     This policy issues instructions in strict program order and also writes their results in program order. This is the simplest policy to implement and the safest: the saved process state (PC, registers and memory) is always consistent with the sequential architectural model; so precise interrupts are easier to implement. However, it is not very efficient: not only does the pipeline stall as soon as a true dependency or a functional unit conflict arise, but also, the instructions are held at the issue stage after slower instructions, even though they have no dependency and would otherwise issue.

- **In-order issue, out-of-order completion**

  In this case, instructions pass through the issue stage in exact program order, but can bypass each other in the execute stage and may finish execution in different order. So instruction issuing is continuous unless there are structural hazards or true dependencies. In particular, instruction issuing is not stalled after a long latency operation, which improves performance for floating point operations.

  The Intel i960 supports this issue policy in a "classical" sense: multiple in-order issue to multiple execution units, with possible out-of-order completion between any of the instructions .

  However, there are notable variations of this scheme: in the IBM RS/6000, instructions issued in program sequence can complete out-of-order between the units (fixed point unit or floating point unit) but have to complete in-order within the same functional unit. In fact, the floating point pipeline is extremely fast, with a two cycle multiply-add operation, and so out of order completion of floating point operations is not that critical. However, floating point memory reads can overlap floating point operations and bypass them.

  Finally, as far as the MC88110 is concerned, instructions leave the *instruction sequencer* [AD92] in strict program order to multiple execution units, but not all instructions *begin execution* in order: branch and stores may wait in reservation stations until their required operands are available. Thus, in a way, branches and stores may issue out-of-order with respect to other instructions. So MC88110 is on the border line between in-order issue and out-of-order issue.

- **Out-of-order issue, out-of-order completion**

  There are severe limitations of the techniques used so far. If an instruction is stalled in the pipeline, no further instruction can proceed. Instead, with out-of-order issue capability, the processor is provided with a larger set of instructions selectable for issue. Strictly speaking, individual instructions must still interlock because of data hazards and structural conflicts, but the processor can look beyond the stalled instructions to discover others which are ready to issue. This increases the processor "look-ahead capability" [Kel75] For this to be possible, the decode and execute stage have to be "decoupled": some kind of instruction pool gathers instructions after decode and before issue. This instruction pool can be a central instruction window which buffers all the instructions, or reservation stations in front of each functional unit [Joh91].

2. **Data Hazards**

When instructions are issued in-order and compete in-order, there is a one to one correspondence between registers and values. When instructions complete out-of-order, and issue out-of order, this correspondence is broken because of storage conflicts between values. Several levels of complexity and performance can be used in the hardware mechanisms that enforce data dependencies; they have a significant impact on the efficiency of the instruction issue/completion policy.

Out-of-order completion creates **output dependencies** because instructions write their results in different order than they appear. There exists different techniques to aleviate those dependencies:

- **Scoreboarding**

  This simple mechanism uses a reservation table where a picture of the data dependencies is constructed and an additional bit is assigned to each register in the register file [Joh91, WS84]. This bit is used to indicate if the register is in pending update, and to interlock an instruction on an output dependence. The MC88110 and i960 implement scoreboard mechanisms to handle their output dependencies.

- **Register Renamming**

  Scoreboarding only allows one pending update of a register. Instead, with register renamming, a new instance of a register can be generated for *every* assignment to a register. **Tomasulo's algorithm** [Tom67] implements register renaming by associating a tag to each register instance. This tag is then used in place of the register identifier. Appropriate tag management ensures in-order writing in each register. The IBM RS/6000 uses a variant of Tomasulo's algorithm, to rename load destination floating point registers [Gro90].

  Other more complex (but more powerful) implementations use some type of buffering device which provides extra storage for instructions results. An example of such device is the **reorder buffer**, [SP85], a FIFO queue where issued instructions place their results. While instructions complete out-of-order, the buffer "reorders" the results, so that they can be written in strict program order to the register file. There are variants of the reorder buffer: for example, the **register update unit**, which integrates both reorder buffer and instruction window features within the same device. The Metaflow Lightning SPARC uses such a unit, called DCAF (Dataflow Content Addressable FIFO), which stores both instructions and their results.

Out-of-order issue creates **anti-dependencies** because register values are not accessed in program order. Antidependencies are enforced by identifying the semantic instances of source registers at decode stage, and associate them with the instructions in the instruction pool. Operands are thus copied with the instructions in the instruction pool in the form of values (if they are ready), register number (scoreboarding case), or tags (register renaming case). The mechanism must ensures that the operands of an instruction will not be overwritten, and that the instruction will eventually issue when the proper operand values are ready and forwarded.

3. **Control Hazards**

   We now address techniques for avoiding stalls due to control dependencies. Control hazards cause a severe performance loss in scalar pipelined processors when the fetcher is stalled until a branch outcome is known. In superscalar machines, the effect of branches is even more critical as instructions need to be consumed at a prodigious rate. Instead of stalling the instruction fetch stage in the presence of branches, most superscalar processors use **branch prediction**, a mechanism by which they guess which direction the branch is likely to go. There are two types of branch prediction techniques:

   - **Static branch prediction**

     This technique can be performed by the compiler , in which case the prediction is associated with the instruction and the processor always predicts that branch taken or not taken. The instruction scheduler of the i960 uses compiler-set prediction bits to choose the direction of unresolved conditional branches [Lai92].

     Some form of static branch prediction can also be performed by the hardware, which simply predicts that all branches will be either taken or not taken. The IBM RS/6000 predicts that unresolved conditional branches will not be taken [GKTM90] and dispatches sequential instructions past the branch to the execution units; however, it also prefetches the branch-taken path in case the prediction is wrong. Hardware can also predict a branch outcome based on other static criterias, such as the direction of the branch.

   - **Dynamic branch prediction**

     This technique is performed at run time by the hardware; the results of past executions of a branch are used to a predict future outcome. One of the most popular prediction mechanism uses a two-bit entry branch target buffer [LS84]. The Metaflow architecture uses such history information about each branch to predict its direction.

   Unfortunately, branch prediction is not guaranteed to be correct; when execution is allowed on a predicted path, there must be some ways to undo the effects of all instructions issued after an incorrectly predicted branch. Again there are different techniques for branch repairs, which rely on post-issue instruction invalidation (instruction flush). Such invalidation is possible when the instruction has not modified the processor state. Array-based schemes exist to support instruction invalidation: the reorder buffer [SP85], history buffer [SP85], future file [SP85] and the DCAF of the Metaflow architecture. They hold the results of branch dependent instructions until the branch outcome is safely determined. When instructions can be undone, branch prediction is well integrated in the out-of-order issue scheme, as it extends the processor look-ahead capability.

However, when there is no instruction undoing capability, the predicted branch path cannot not pass the decode stage.

4. **Memory Access**

   Memory access efficiency is critical in order to sustain the high issue rate of the processor. Methods have been devised to reduce the effect of memory bottleneck on superscalar instruction execution. They rely on the buffering of stores in a store buffer.

   - **Load bypassing** allows loads to bypass stores even though the stores appear first in the program sequence. Giving loads a higher priority over stores accelerates instruction processing because loads produce results that are likely to be needed by close subsequent instructions, whereas stores can wait. However, to ensure correctness, a load cannot bypass a store that accesses the same memory location.
   - **Load forwarding** goes further by allowing a load to bypass any store, even if there is a memory conflict between them: in case of a dependency, the load simply takes its value from the store buffer.

1.2 SuperDLX - A Superscalar Simulator

There are different ways to evaluate the benefits of design ideas for an architecture, in terms of hardware cost versus performance. The most accurate way is certainly to build a prototype, but it is also too time consuming and expensive. A more efficient way is to build a trace-driven simulator, which uses an instruction trace, generated by a trace generator of a machine. Trace-driven simulation is fast in a run-time sense, because the simulation is only concerned by modelling the processor features that affect performance: it does not record values in the register file or memory, because it only needs to keep track of dependencies between intructions; also, it does not compute any results. However, because the simulated program is not executed, correctness of simulation is not ensured: one might argue that this is not necessarily a negative point since strict correctness is not indispensable to assess the performance of an architecture model. A trace-driven superscalar simulator, MATCH, [Joh91, SLH90] was developped by Mike Johnson, at Stanford University

SuperDLX, the superscalar simulator developped in the scope of our project, is not trace driven. We took this "execution" oriented approach for different reasons:

- **Accuracy**

  We wanted the results of simulated assembly instructions to be effectively computed and the contents of the different hardware elements to be recorded on a cycle basis. Correctness of simulated program output was very important for us to assess the proper coordination of all the different simulated hardware components.

- **Portability**

  We wanted a simulator executing assembly programs generated by a cross compiler, so that it be portable to different machines.

- **Usability**

  Not only did we want an architecture evaluation tool, but also a pedagogical tool aimed at students to learn superscalar mechanisms. The user interface permits one to run the simulator on a cycle basis, and examine the contents of the various hardware elements at a given cycle.

Our simulator implements the most sophisticated superscalar instruction processing policy: multiple out-of-order issue, multiple out-of-order completion. To achieve this policy, efficient hardware mechanisms were selected for simulation: a central window, buffering instruction for issue, and a reorder buffer, supporting register renaming. Other features were built around them to go even further in performance: branch prediction, load and store buffering. Taken individually, these mechanisms were very attractive because they seemed rather powerful while conceptually simple. Finding the way to effectively orchestrate them was certainly the most decisive and challenging part of implementation.

SuperDLX can be viewed as a "superscalar version" of DLXsim , the DLX simulator developped at Berkeley [HB92]: it uses the DLX instruction set produced by the DLXcc compiler; it also reuses part of the DLX simulation environment, which proved very useful. But the simulation cores of both simulators are fundamentally different, as DLXsim is the simulator of a scalar MIPS-like architecture.

1.3 Document Organization

Chapter 2 of this report presents the processor model that was chosen for the simulator. It also gives an overview of the different pipeline processes that had to be simulated. This chapter is intended for those who wish to have a general idea for the superscalar features involved in the simulator. Chapter 3 presents the main data structures modeling the different processor elements. Starting from this chapter, the document goes into more details in the implemented software. This chapter and the following ones are intended for those who wish to study or modify the program. The fourth chapter is devoted to the functional description of the simulation core of superDLX. The main functions implementing the different overlapping pipeline stages are detailed. Chapter 5 briefly introduces the on-line interface that interacts with the simulator.This chapter, although dealing with the interface, is still implementation oriented: that is why a user manual is provided in appendix A. Finally, the last chapter gives some outlines for future improvements of the simulator.

2 Overview of the Execution Model

2.1 The Processor Model

The basic processor model is represented in figure. 1.

It consists of two operational units: an integer and a floating point unit. These operational units are supplied with instructions coming from the instruction queue, where fetched instructions are buffered. Each operational unit contains a set of functional units where instructions are executed, and a register file where results are written . Those hardware elements are not different from those of a classical scalar processor.

But other elements have been added to support superscalar techniques:

- **multiple out-of-order issue**. In each operational unit, the decoder places instructions in program order in a central instruction window that decouples instruction decoding from instruction execution to perform dynamic scheduling . The instruction-issue logic examines the instructions in the window, selects some of them for issue, not necessarily in program order, and dispatches them to their appropriate functional units. Any number of instructions is allowed to be in execution in the functional units, as long as resource conflicts are resolved.
- **multiple out-of-order completion**. Functional units have various latencies, and instruction issuing is not stalled when a functional unit take more than one cycle to complete its operation. So instructions can complete out of program order. Hardware mechanisms must insure that results are written in correct order in registers : storage conflicts are resolved with **register renaming**, using a reorder buffer where results of the instructions are placed once computed.

Other mechanisms play their part to accelerate instruction processing and thus enhance the above superscalar features:

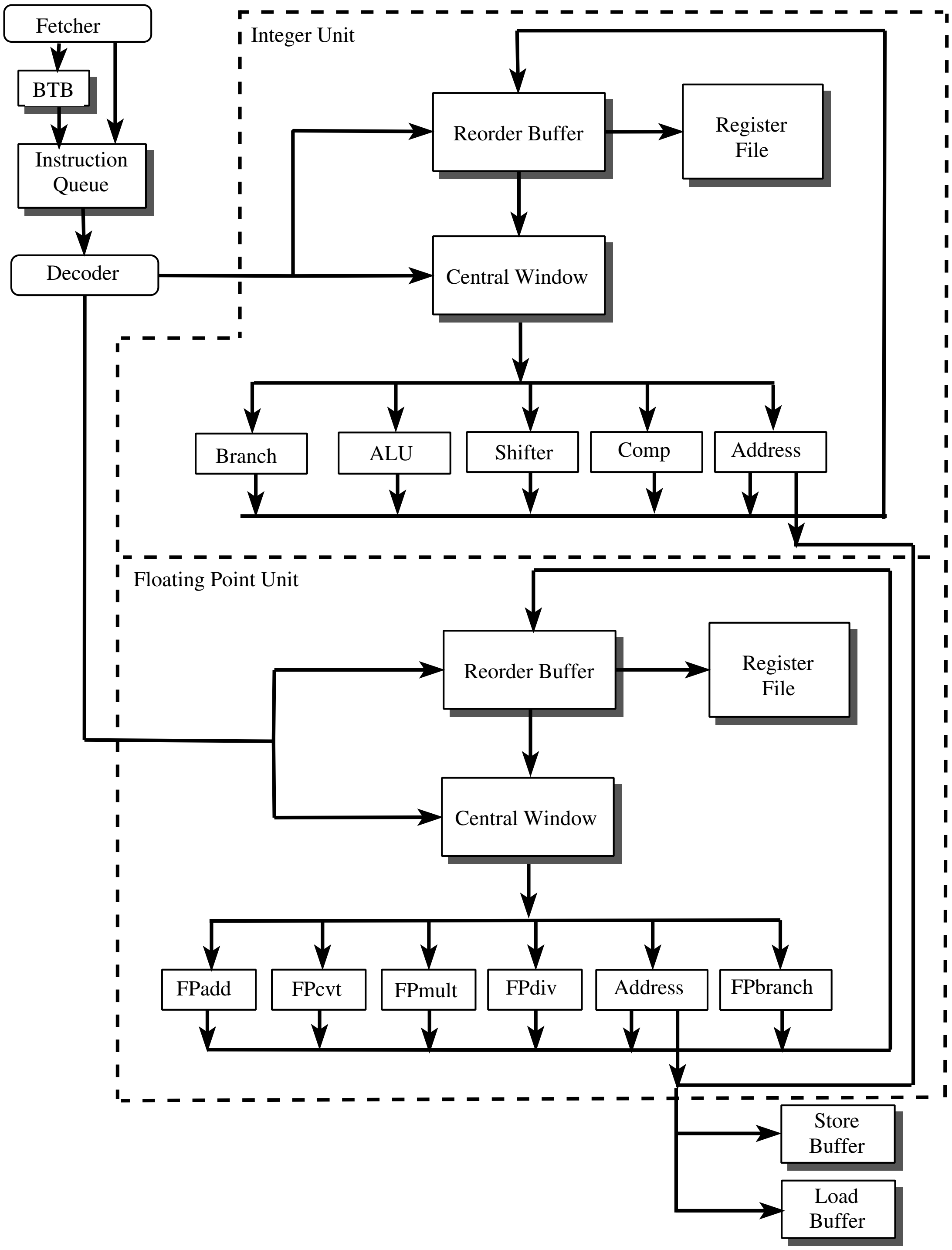

Figure 1: The Processor Model

- a **branch target buffer** between the instruction fetcher and the decoder. This branch target buffer enables branch prediction to be performed by the instruction fetcher. It allows the processor to execute instructions past conditional branches; the reorder buffer is then used to recover from any mispredicted branch.
- a **load and a store buffer**, common to both operational units. They permit the load / store address calculations to be decoupled from memory accesses, and allow loads to bypass stores, provided there is no memory address conflicts. By giving loads a higher priority, the overall instruction processing is accelerated.

2.2 The Different Pipeline Stages

The idea of pipeline stages is kept since each processed instruction needs to be

- **fetched**: it is entered in the instruction queue;
- **decoded**: it is placed in the instruction window of the integer or the floating point operational unit;
- **issued**: when its operands are ready, it can leave the instruction window to be executed;
- **written back**: its completed result is written in the reorder buffer;
- **committed**: its result is written to the register file. The **commit point** was identified by [PGH$^+$87] to be the time when the instruction modifies the processor state.

Five overlapping stages or processes simulate the multiple instruction pipeline, each process running at its own pace. They are detailed below.

2.2.1 Fetch Stage

This stage fetches instructions from the cache and places them in an instruction queue. The maximum number of instructions fetched per clock cycle is parameterized (see chapter 5).

The presence of branches creates two major problems that disrupt the fetch mechanism [Joh91]:

1. instruction fetching depends on the results of branch instruction execution;

2. branches cause instructions to be mis-aligned with respect to the decoder. That is, because of the presence of a branch in a cache block, instructions in the right hand side of the block may be non valid. Similarly, targets of branches are not necessarily aligned to the left hand side of a block.

The first problem is resolved by branch prediction: instead of stalling when it recognizes branches, the fetch process performs branch prediction based on the past behavior of the branch, recorded in the branch target buffer. The branch prediction mechanism used is described in chapter 2. The second problem is not modeled in the simulator: instructions are fetched based on their own PCs, and thus we do not consider them as parts of fixed sized cache blocks which could contain non valid instructions. As a consequence, instruction alignment is ensured. Similarly, instructions from different runs are automatically merged.

Thus branch prediction, combined with instruction aligning and merging, limits the negative effects caused by branches in instruction fetching. However, mechanisms must be provided to recover from wrong predictions: the penalty caused by such recovery is then effective later in the pipeline, as we will see, in chapter 4.

Direct jumps are handled easily, as their targets can be calculated at fetch stage; moreover, the instructions after the jump can be merged in the instruction queue without penalty. However, the presence of indirect jumps cause more problems. The fetch process has to stall when it meets an indirect jump, because the value of the jump register is not guaranteed to be correct. This does not severely affect instruction fetching since the percentage of indirect jumps is usually small in DLX instruction mix see [HP90];

2.2.2 Decode Stage

This stage takes instructions from the instruction queue, decodes and dispatches them to their appropriate operational unit (integer or floating point). The maximum number of instructions decoded per clock cycle is a simulation parameter.

Each decoded instruction is assigned an entry in the *reorder buffer of its operational unit*, where is placed

- its destination register identifier;
- its result value when produced.

The entry is assigned in program order (which is the decode order) at the tail of the reorder buffer, which acts as a FIFO queue. The point is that all instructions go to the reorder buffer, even those who don't need to be take part in the register renaming process. As a consequence "dummy entries", which do not produce any results for the register file, can be created in the reorder buffer: this simplifies the processing of stores and branches, as we will see in chapter 4.

At the same time, the decoded instruction enters the *central instruction window of its operational unit*, from which it will be issued. Copies of the instruction's operands are needed there, to be available for instruction issue and to avoid antidependencies: to obtain the copies, each operand is first searched in the reorder buffer. If it can be found there, but is not yet available (i.e not yet computed), the corresponding reorder buffer entry number is taken in place of the operand value. If the operand is available in the reorder buffer, then the computed value can be taken. Provision must be made to get the most recent value/entry in the reorder buffer. Finally, if this operand is not found in any entry of the reorder buffer, then its value is taken from the register file.

If the instruction is a load or store, it is also recorder in the load or store buffer. These buffers are common to the floating point and integer units.

Generally speaking, each instruction is assigned an entry in the reorder buffer and and the instruction window of the **same** operational unit, but the instruction operands can be taken from the other operational unit: for example a floating point load has its memory address calculated on the basis of integer registers, but its loaded result is placed in the floating point reorder buffer. Branches are again a special case: they can be provided an entry in both integer and floating point reorder buffers, for synchronization purposes.

2.2.3 Execute Stage

The issue logic examines the instructions in the instruction window and selects the "ready" ones:

- those for which operands are available;
- those for which a functional unit is available.

When two instructions conflict for the same functional unit, the selection is made based on the age of the instruction in the window: the oldest one has the highest priority.

The latency of the various functional units has to be simulated: when a long latency operation is issued, its result is calculated at once and placed in the reorder buffer entry of the instruction, with a "ready" field indicating at which clock cycle the result is available.

The execute process permits the processor to determine the effective outcome of predicted branches: if a branch prediction happens to be wrong, the instructions following the branch in both reorders buffers are "flushed". In this case, the branch target buffer is updated with new prediction information.

Memory is considered as another functional unit. Loads and stores are issued from their respective load and store buffers to the cache (a perfect cache is assumed). But while stores are constrained to modify memory in order, loads can be sent out of order. Moreover, loads have a higher priority and can bypass the stores which do not affect the same memory locations. Load bypassing accelerates instruction processing, because loads provide the processor with values that are needed to continue with the execution of instructions.

2.2.4 Write Back Stage

The write back logic identifies the completed operations in the reorder buffer and frees the corresponding functional units. The completed results are validated and are forwarded to the instructions that need them in the instruction windows.

2.2.5 Result Commit Stage

The result commit stage sends the validated results in the reorder buffer to the register file: the register writes are processed in order, from the head to the tail of the reorder buffer, until an instruction is found with an incomplete result. The committed instructions are removed from the reorder buffer. Invalidated instructions (that follow a wrong branch prediction) are simply discarded.

The reorder buffers and the store buffers have to be synchronized to ensure that stores are safely sent to memory (when all previous branches have completed). The mechanism is the following: when a valid store entry appears at the head of the reorder buffer, it is allowed to proceed to the data-cache, in a future execute stage. So, while the store buffer aids in keeping the stores in order, the reorder buffer defers the completion of a store until previous instructions complete.

Full synchronization is not necessary between the integer and floating point reorder buffers: this is because they write in different register files and also because stores are ordered in the store buffer. However, there are cases where their synchronization is necessary: for example, instructions cannot bypass a branch from the other operational unit until the outcome of the branch is safely determined. That is why an entry is created for each branch in both reorder buffers.

Synchronization of the different buffers is a delicate part of the result commit process: this is because we have two operational units, and thus two reorder buffers. So the program order is kept over two devices, and there exists a risk of modifying the process state out of program order. Metaflow resolves this problem by sending all the program instructions to its integer DCAF, even if the floating point instructions are also entered into the floating point DCAF, from where they are issued [BD91].

3 Underlying Data Structures

As we have seen, the simulator needs to model two types of components, in the processor:

- the *basic processors elements*, which model memory, the register file and the functional units.
- other more complex structures that were added to efficiently support superscalar mechanisms: the branch target buffer (BTB), the instruction queue, the instruction windows, the reorder buffers, the load and store buffers. We call them the *superscalar hardware elements*, although this "superscalar" appellation is not quite appropriate for all of them: the branch target buffer, associated with an instruction queue, is a mechanism also used in scalar processor to improve fetching efficiency.

The main characteristics of the processor components (size, number...) are described in a file, the **machine configuration file**, which contains the user's parameters.

The DLX instruction set [HP90] needs also to be described for the simulator to able to handle it; various instruction tables are used for this purpose.

3.1 Instruction Tables

Each opcode in the instruction set is given a number that serves as an index to the *opcode description table* of the simulator. This table allows, at each processor stage, to get the characteristics of an instruction, and thus determine its behavior. This table reflects all the choices that were made concerning the processing of each instruction in the superscalar processor:

- choices on the operational unit where to execute instructions. As a general rule, the instructions are sent to the operational unit that correspond to their result type. But this rule is not so simple: the result is that some instructions (floating point loads and stores) are computed in one operational unit, but their operands are taken from the other operational unit;
- choice on the functional units to be implemented; in particular, we chose to have separate address calculation units for branch and load/store operations, in both operational units;
- choices on the handling of special instructions such as traps, moves from floating point to integer registers (and vice versa).

The main idea behind this table is to gather instructions into groups that follow the same rules. But, as we will see, it turns out that a lot of instructions are special cases. And this table also proved to be very useful to handle peculiarities.

Each table entry give the following information:

- **opName**: the opcode string name;
- **opUnit**: the operational unit where the instruction must be processed: INTEGER or FLOAT;
- **class**: the class of the instruction: this class field groups instructions into categories: branches, loads, stores, etc...;
- **funcUnit**: the functional unit where the instruction is issued to for execution;
- **firstOperand**: the type of the first operand, if it exists (NONE, INTEGER, FP_SIMPLE, FP_DOUBLE);
- **secondOperand**: the type of the second operand, if it exists (NONE, IMMEDIATE, INTEGER, FP_SIMPLE, FP_DOUBLE);
- **result**: the type of the result, if it exists (NONE, INTEGER, FP_SIMPLE, FP_DOUBLE).

The contents of the table is given in appendix B.

Other tables are reused from DLXsim ("opTable", "specialTable", "FParithTable"), as they are needed by functions that also have been reused (mainly the functions that deal with the assembly code format). For more information on those three tables, see the DLXsim reference manual [HB92].

3.2 Basic Processor Elements

This section briefly describes the basic processor elements, common in all processors: memory, register file, functional units.

3.2.1 Memory

Memory is organized as a table, **MemPtr[NUM_WORDS]**, accessing the memory locations and indexed by the memory location addresses (addresses of data, PCs of instructions). The memory size is a parameter.

The memory locations, (**wordPtr**), can contain either a value or an instruction, and consist of the following fields.

- **value**: content of the memory location (for data accesses);
- **opcode**: opcode (for instructions): number which permits to access the **opcode description table**; it is initialized to NOP;
- **rs1,rs2,rd**: three registers (for instructions): 2 sources and 1 result;
- **extra**: (for instructions) immediate, target, or offset.

3.2.2 Register File

The integer and floating point register files are implemented using a single table, **regs[TOTAL_REGS]**, accessed by register number.

The first 32 registers are general purpose integer registers, followed by 32 floating point registers, by the floating point status register (reg[FP_STATUS]), and by the current fetched pc (regs[PC]).

3.2.3 Functional Units

Two tables represent the functional units in the Simulator.

- **intUnits[MAX_UNITS]**: integer functional units

- **fpUnits[MAX_UNITS]**: floating point functional units

Each element of those tables characterizes a functional unit type, with the following fields:

- **latency**: number of cycles needed between issue time and write back time for an instruction using this type of functional unit: this is used to fill the ready field of an instruction in the reorder buffer, when this instruction is issued; it is a user set parameter;
- **num_units**: number of available instances of that type of functional unit; it is also parameterized;
- **num_used**: number of functional units of that type which are currently used: this permits the resolution of structural hazards.

Each functional unit type information is indexed by its identifier. Functional unit identifiers are associated to instruction types in the opcode description table, so that the instruction issue logic knows where to send instructions at issue time:

- in the integer unit we find the following functional units:
  - **INT_ALU**: for all integer additions, subtractions, logical operations and register to register moves;
  - **INT_SHIFT**: for all integer shift operations;
  - **INT_DATA**: for all address calculation of integer load and stores;
  - **INT_COMP**: for all integer set conditionals;
  - **INT_BRANCH**: for all integer conditional branch address calculations;
- in the floating point operational unit we find:
  - **FP_ADD**: for all floating point addition, subtraction, and fp register to fp register moves operations;
  - **FP_DIV**: for all floating point division operations;
  - **FP_MULT**: for floating point and integer multiplications;
  - **FP_CONV**: for type convert instructions;
  - **FP_DATA**: for floating point load and store address calculations. As we already mentioned, although address computations involves only integer registers, choice was made to execute these computations in the floating point unit.
  - **FP_COMP**: for double and single floating point compares;
  - **FP_BRANCH**: for BFPT and BFPF address calculations.

3.3 Superscalar Elements

This section details the mechanisms used to support the superscalar aspects of superDLX.

3.3.1 Branch Target Buffer

The branch prediction scheme used by the simulator is based on the two-bit prediction scheme of [LS84]. The idea behind this mechanism is that the past behavior of a branch may help the fetch function to predict the next outcome for that branch, and thus fetch the predicted path. SuperDLX uses a table, **branchBuffer**, which is indexed by the lower-order bits of branch addresses (given by the address modulo the size of the table). When a branch is fetched, this table is consulted to get the predicted outcome of the branch. The branch target buffer size is specified in the machine configuration file.

Each entry in the buffer records the following information:

- **address**: the address of the branch instruction for which the entry has been set up. This field is necessary to resolve conflicts between branches that have the same lower order bits.
- **predictState**: a two bit number giving the next predicted outcome for the branch. This number can take four values:
  - TAKEN (11), TAKEN? (10) which make the branch be predicted taken;
  - NOT_TAKEN? (01), NOT_TAKEN (00) which make the branch be predicted not taken.

  Initially (when the branch is met for the first time), the information is set to TAKEN. It is then updated every time the effective outcome of the branch is determined by the execute stage; taking the branch causes the prediction number to increment, not taking it causes it to decrement. We do not wrap around when the field reaches its maximum or minimum. Figure 2 shows the finite-sate machine for the two bit prediction scheme. As we can see, a two bit scheme minimizes oscillation that could occur in a one bit scheme.
- **predictTarget**: this gives the target of the branch if taken.

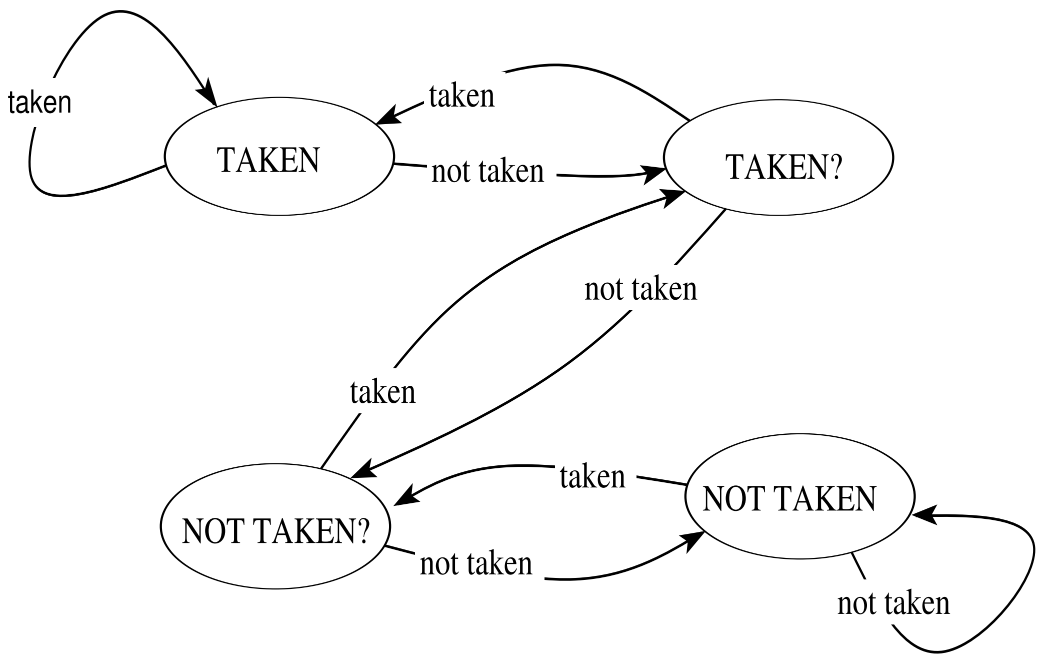

Figure 2: State Diagram for the Branch Predictor

3.3.2 Instruction Fetch Queue

The Instruction Queue is a FIFO queue where the fetcher adds a fixed number of elements at the top, and the decoder takes a fixed number of elements at the bottom. The simulator uses a doubly linked list; each item contains the following information:

- **wordPtr**: memory word fetched;
- **address**: instruction address (pc);
- **insCount**: dynamic instruction count (incremented every time a new instruction is fetched);
- **prediction**: prediction information for branches: TAKEN, NOT_TAKEN. This is the prediction made by the fetch process when it encounters a branch;

The maximum number of elements in the instruction queue (**numEntries**) is a user set parameter from the machine configuration file.

3.3.3 Instruction Windows

Two instructions windows are modeled, one for each operational unit: **iWindow** and **fpWindow**. Their sizes (**numEntries**) are parameters from the machine configuration file.

In the processor, instruction windows are compressible stacks that keep the instructions in order as shown by figure 3 (taken from [Joh91]). Decoded instructions are placed at the top of the stack, but they can be issued from any position in the stack. Keeping instructions in order makes it simpler to prioritize among ready instructions, as older instructions appear at the bottom of the stack. But this requires hardware mechanisms to allocate and compress the window. SuperDLX simply models this mechanisms with linked lists.

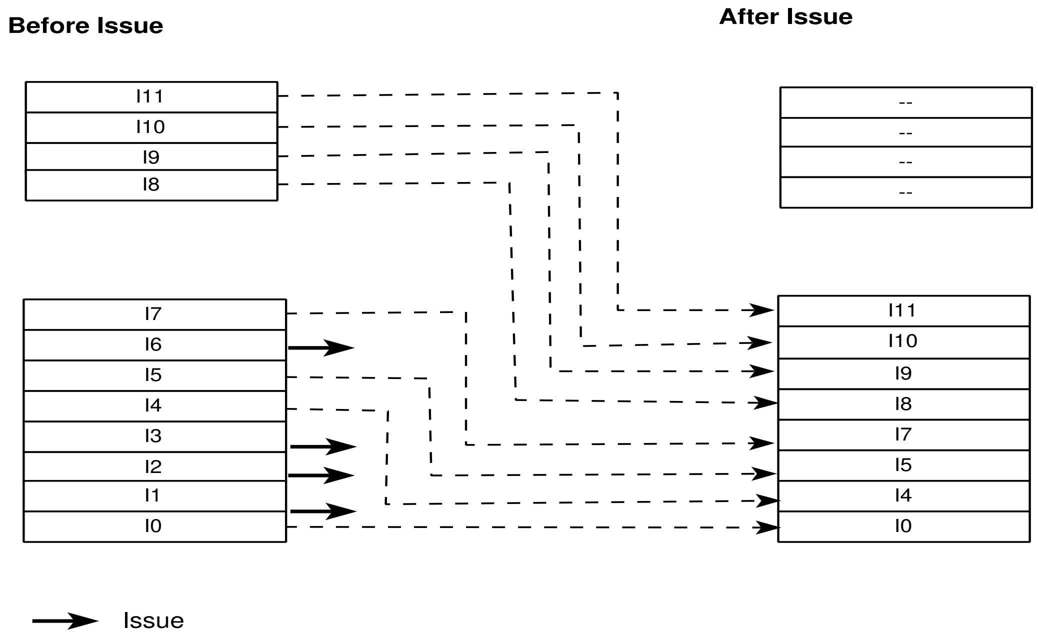

Figure 3: Keeping Instructions in Order in the Central Instruction Window

Each item in the instruction window represents an instruction that has been decoded, and contains basically the following fields:

- **opCode**: the decoded opcode of the instruction;
- **class**: the instruction class;
- **unit**: the functional unit where the operation must take place; it is determined by the decode function, using the opcode description table (see appendix A);

- **reorderEntry**: the reorder buffer entry the instruction is assigned to. A link must be kept between the operation in the window and its corresponding entry in the reorder buffer, for the issue logic to process the instruction properly, as we will will see in chapter 4;
- **prediction**: the prediction information (TAKEN, NOT_TAKEN) for branches;
- **firstOperand**: the information on the first operand, if it exists;
  - *value[2]*, which contains the operand value or the reorder buffer entry from where the value will be forwarded when computed;
  - *type*, which contains the type of the operand (integer, floating point double, floating point simple). It is not necessarily the same as the instruction's result;
  - *valid*, which it indicates if the operand is ready to be used;
- **secondOperand**: the information on the second operand, if it exist (same structure as the first operand, but this time, the type of the operand can also be immediate);

3.3.4 Reorder Buffers

Again, there are two instances of reorder buffers in each operational units: **iReorderBuffer** and **fpReorderBuffer**, and their respective sizes are parameters. Each of them is a circular queue, with a head position (bottom of the queue) and tail (top of the queue) as shown by figure 4. New decoded instructions are assigned an entry at the tail of the queues, and they commit in order at the head. With this mechanism, instruction entries do not have to move towards the bottom or top of the queue: only the head and tail move, and thus the buffers entry numbers can be used to rename the result registers.

Each entry in the reorder buffer consists of the following:

- **registerNb**: the result register number. It is used when an operand is searched in the reorder buffer (associative search), and it indicates which register to write to when the instruction commits at the head of the buffer;
- **opCode**: the opcode of the instruction;
- **class**: the instruction class which the instruction belongs to; the result commit and write back logics use this field to know what to do with the instruction;
- **result[2]**: the computed result of the instruction (two numbers for double floating points). Some instructions (such as stores) produce no result;

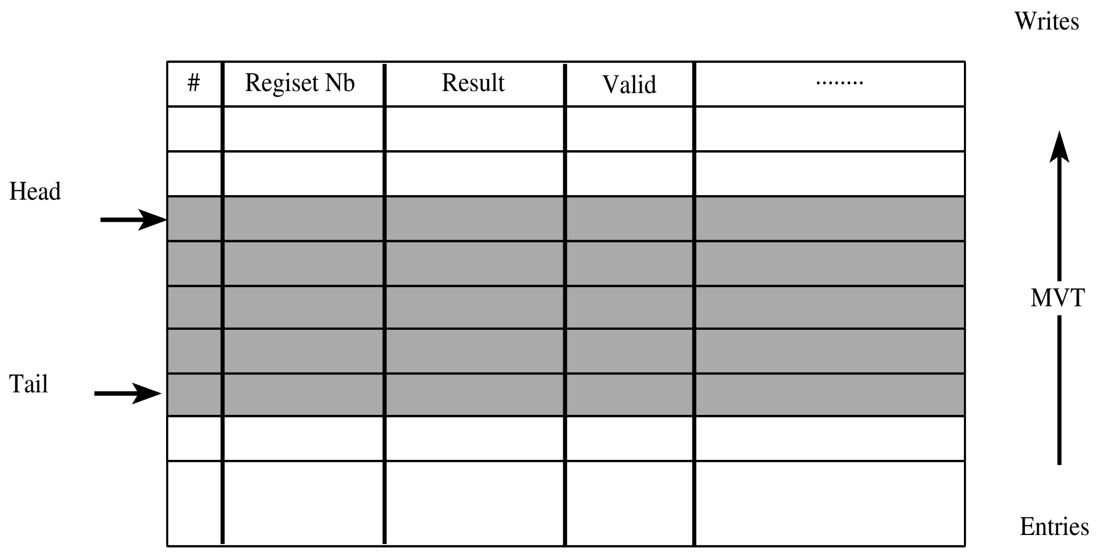

Figure 4: Reorder Buffer Entries

- **valid**: the validity bit of the result, which indicates if the operation has been completed, and thus if the result is available;
- **ready**: it indicates at what clock cycle the computation of the result is going to complete;
- **insCount**: the dynamic instruction count calculated when the instruction was fetched;
- **flush**: the flush bit. It is set to 1 when the instruction follows a wrong branch prediction

A list is maintained, which links reorder entries of currently executing operation. This list of pending operations (*listOfExec*) makes the work of the write back stage easier and faster.

3.3.5 Load and Store Buffers

The load buffer and store buffer are common to the integer and floating point operational units. This is for practical reasons: load and stores can "compete" for the same memory locations, no matter if they are floating point or integer operations. The use of a single store buffer and a single load buffer facilitates load bypassing: a higher priority can be given to loads, and loads can bypass stores when they do not affect the same memory location. As far as stores are concerned, they must proceed in exact program order to the cache and they can only be sent to cache when all instructions preceding them in program sequence have completed.

Again, the sizes of the load buffer and the store buffer are parameters. Each entry consists of the following fields:

- **opCode**: the opcode of the operation, which give the size and type of the data to access;
- **address**: the address of the data to store/load;
- **reorderEntry**: the corresponding reorder entry number of the store/load instruction;
- **insCount**: the instruction count, used for load bypassing checks;
- **dataToStore**: (for stores only) the description of the data to store, using the same format as an operand;
- **released**: the release bit, which indicates if the load/store is allowed to proceed to memory; for loads, it is set to one when the load address calculation is complete; for stores, it is set to one when a store with a completed address arrives at the head of the reorder buffer;
- **flush**: the flush bit, set to one if the instructions follows a mispredicted branch.

As a general mechanism for handling load and stores in their respective buffer, we propose the scheme illustrated by figure 5.

Load and store processing is divided into two distinct steps:

- address calculation
- access to data-cache

Loads and stores are placed in the instruction windows, to be issued to the address calculation units. They are also provided with an in-order entry in the load/store buffers. From the window, loads and stores can issue out of order (like any other instructions) to the address calculation units and the calculated address is placed in the load/store buffer. From the store buffer, stores must process strictly in order to the data cache, and they can only be sent once all previous instructions have completed successfully. Loads can process out of order to the

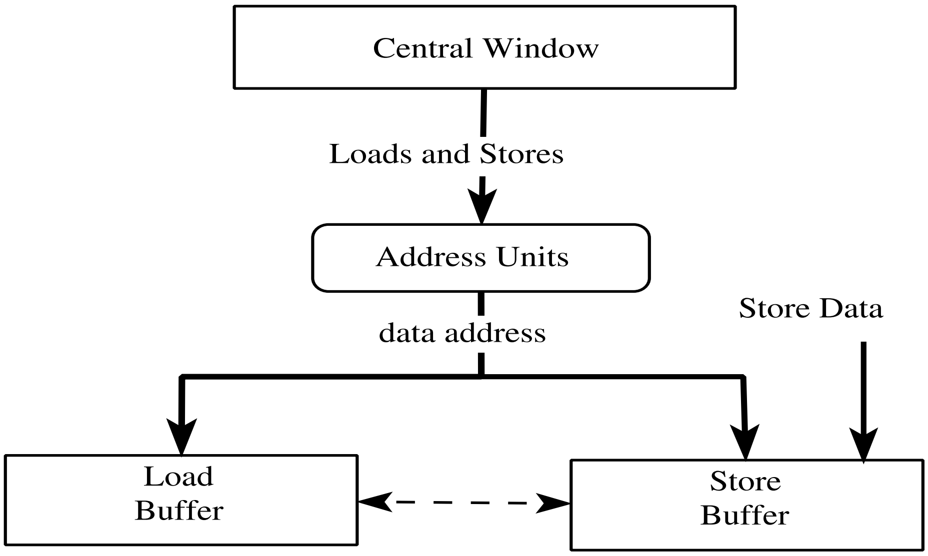

Figure 5: Load and Store Processing

cache, and they can also bypass older stores that are in the store buffer, provided they do not have conflicting addresses.

The number of memory accesses per clock cycle, is subject to debate: in [Joh91] the approach taken with one store buffer but no load buffer limits the accesses to one per clock cycle. By adding a load buffer, we decouple load address calculation from data access: this way, we do not limit a priori the number of loads that are sent to the cache on one clock cycle. Besides, using this model, the loads can be sent out of order to memory, once their address calculation is performed.

3.4 A Running Example

Appendix E illustrates how information (data, instructions) flows in the different machine elements, with a running example. The example also illustrates the use of the on-line interface described in chapter 5.

4 Detailed Functional Description

4.1 Overall Functional Organization

Figure 6 gives the overall functional organization of the simulation core of superDLX. The processes on the diagram represent groups of functions, that we will detail, using a top down approach.

How are these groups of functions coordinated in the simulator? The problem was to find a way to simulate the overlapping of the pipeline functions while using a sequential programming model. To understand the importance of this function coordination, we can refer to the diagram. As we can see, functions consume data that have been produced by others: for example, the decode functions produce instruction information that is placed in the instruction windows and is then removed by the execute functions as operations to be executed. So there is a consumer/producer relationship between functions, and superDLX uses this relationship to coordinate the different functions: it works on a per clock cycle basis, calling each function in a consumer-to-producer order. That is, the result commit functions are called first, then the write back functions etc..and this ends with the fetch process, as shown by figure 7.

This organization prevents data from different cycles to be mixed up in the shared resources such as the instruction window or the reorder buffer. This is easy to understand: suppose the fetch functions were called before the decode functions at a given cycle; then, at the end of the fetch stage, there would be instructions in the instruction queue which could only be decoded on the next clock cycle; the decoder would thus not have a uniform view of the instruction queue. To avoid this type of situation, the pipeline is somewhat "reversed". But, in reality , what does "pipeline order" mean from a superscalar point of view? The pipeline stages are in fact independent processes: they only communicate via shared resources.

This function coordination is performed by a function called **Simulate**, in supersim.c.

4.2 Fetch Function

The fetch stage takes place in a single function: **FetchInstruction** in superfetch.c. The function reads one instruction at a time from memory . Instructions are accessed based on their individual PCs. Thus instruction misalignment, caused by branches causes disrupting the sequentiality of instruction addressing (see [Joh91]) is of no concern for us here. And instructions from different branch runs are naturally merged. Each instruction is processed depending on its class (see appendix B):

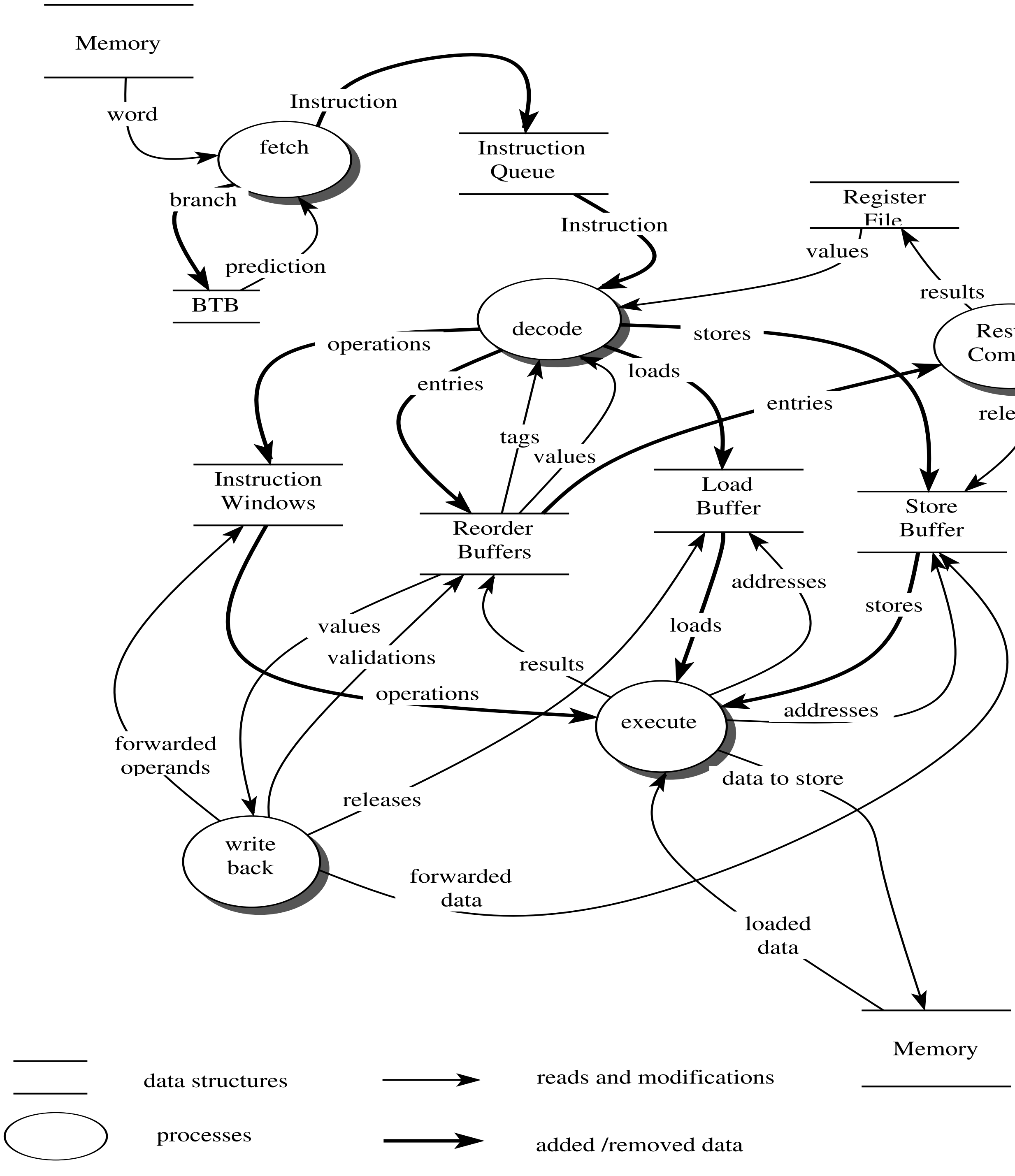

Figure 6: Overall Functional Diagram

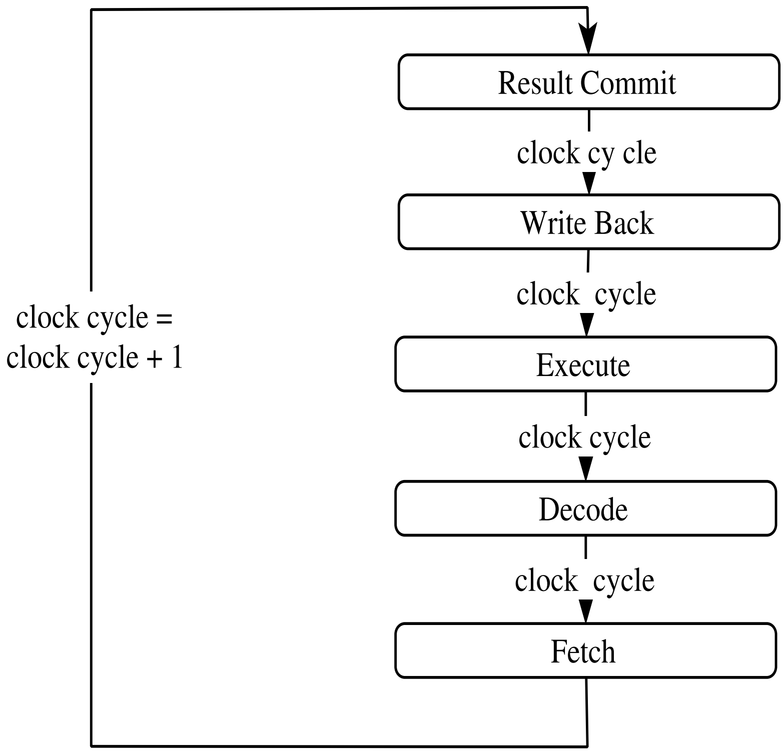

Figure 7: Functions Coordination

- **Nops**
  They appear in DLXcc generated code, because DLXcc is generating assembly code intended for DLXsim, a scalar MIPS-like RISC machine. We have to ignore those NOPs. If too many NOPs are fetched consecutively though, this means the simulator is reading a part of memory where no program has been loaded. In that case, an error flag is raised (the user must certainly have entered a bad starting address).

- **Branches**
  When a branch is fetched, two cases may happen:

  - **the branch prediction option is set**; then the branch searched in the branch target buffer, and again there are two cases:
    * if absent there; this means the branch is encountered for the first time or a collision has taken place before, that has overwritten the branch entry. Then the branch is entered in the branch target buffer, with its prediction field information initialized to TAKEN. The PC is then set equal to the branch target address;
    * if present: the prediction information is read from the branch entry, and the PC is either set equal to the branch target address or is simply incremented, depending on whether the branch is predicted TAKEN of not.
  - **the branch prediction option is not set**; the fetch process is stopped until the branch outcome is determined by the execute stage;

- **Indirect jumps**
  Because their target depends on a register, they give even more troubles than branches: to calculate their targets we would have to search for the target registers, either in the integer reorder buffer or in the register file; but nothing ensures that the correct register values would be found here, as they could be modified later by instructions preceding the jump and still in the instruction queue. Therefore there is no other solution but stopping the fetch process, until the target of the jump is determined later on, by the execute stage;

- **Direct jumps**
  When a direct jump is encountered, the jump offset is added to the PC: since the direct jump is executed by the fetch function, it does not need to be entered in the instruction queue;

- **Traps**
  Traps are used by superDLX as in DLXsim, to give control to the operating system, when a function of the C library (printf, strcpy..) is called by the simulated program.

SuperDLX processes them as single instructions: but they must be handled carefully because they can only be executed once all the stores before them have completed. We will see how this is ensured by the result commit function. The fetch stage itself is only concerned with the trap that exits the simulated program (trap #0): when such a trap is met, a flag is raised to stop the fetch process. However, this exit trap may follow a wrong branch prediction: in that case, after the wrong prediction is discovered by the execute stage, the fetch process is restarted;

- **any other instructions**
  They are simply entered in the instruction queue.

4.3 Decode Functions

The decode functions are gathered in **superdecode.c**. They consist of:

- **DecodeInstruction**, which is the main decode function. It takes each instruction from the instruction queue and creates an operation to be inserted in the instruction window.
- **SearchOperand**, which is called by DecodeInstruction to search for operand values either in a reorder buffer or in the register file.
- **EnterRBuffer**, which is called by DecodeInstruction to create a reorder buffer entry for the decoded operation;
- **EnterDBuffer**, which is called by DecodeInstruction to create a load/store buffer entry for the decoded instruction;

4.3.1 Creating an Instruction Window Entry

For each instruction to decode, **DecodeInstruction** uses the opcode description table (see appendix B), indexed by the instruction opcode, to:

1. determine which operational unit the instruction must be sent to (and thus which reorder buffer and instruction window): integer or floating point;
2. identify the functional unit where the operation will be executed;
3. identify the instruction class, which indicates what to do with the instruction;

4. determine the instruction operand types so as to know if operand copies are needed, and in what type of reorder buffer or registers they must be searched.

Once this is done, the decoding takes several steps:

1. identify **rd** (destination register), **rs1** (source register 1), **rs2** (source register 2) of the instruction;
2. create an **operation** to be inserted in the appropriate instruction window;
3. get the first operand: SearchOperand is called to get the value of rs1;
4. get the second operand if it exists: the way to fill it depends on the instruction class:
   - **Trap**
     The trap number, given by the instruction extra field, is placed in the second operand of operation, and the second operand is marked valid;
   - **Indirect jump**
     We have seen that an indirect jump caused the fetch process to stall. Rs1 is the jump register: if a value has been found for it, the PC is set equal to this value and the fetch process can be restarted. There is no need to establish a second operand, as the operation does not need to be entered in the instruction window: the instruction execution is complete. On the other hand, if rs1 is not valid, the operation must proceed in the pipeline, and an empty second operand is set up which is validated;
   - **Store**
     The immediate field of the instruction is assigned to the second operand and is validated; it will be added to the first operand value for the store address calculation. A store entry is created in the store buffer; the result register rd gives the data to store, which is processed as an operand (that is, it is searched with a call to SearchOperand and is placed in the store buffer entry);
   - **Load**
     The second operand receive the extra field of the instruction and is validated. A load entry is created in the load buffer;
   - **any other class of operation**
     If the second operand type is:
     - IMMEDIATE: the second operand value receives the immediate field of the decoded instruction, and it is validated;
     - NONE: the second operand value is initialized to 0 and is validated;

  - INTEGER, FP_SIMPLE, FP_DOUBLE: SearchOperand is called to get the value of rs2.

5. enter the operation in the instruction window.

4.3.2 Operand Searching

SearchOperand searches for copies of operands, on requests from DecodeInstruction.

Firstly the type of the reorder buffer where to look for the value is identified, depending on the type of the operand. Then the operand value is looked for in the reorder buffer: the reorder buffer is examined from tail to head, and the operand search ends when:

- a reorder buffer result register matches the operand register. In that case, there are two possibilities:
  - the result is valid (i.e has been computed): it is taken and the operand is validated;
  - the result is not valid (i.e has not been computed or is currently being computed): the reorder entry number of the result is taken, to be associated with the operand in the instruction window.
- the operand is a simple floating point and its register number matches the *second* register of a double floating point reorder buffer result: then we find the same two possibilities as above, but we only consider the second part of the reorder buffer result;
- the reorder buffer has been searched entirely without finding the operand; then the operand value is taken from the register file and is marked as valid.

The traversal of the reorder buffer from tail to head ensures that the most recent write to the searched register is considered (the result register may appear in several entries of the reorder buffer).

4.3.3 Creating a Reorder Buffer Entry

Once the operation and its operands have been placed in the instruction window, an entry is assigned for the decoded instruction in the reorder buffer. To do this, EnterRBuffer simply increments the tail modulo the size of the buffer and fills the newly created tail entry (see 3.2.4). All fields are known at this stage, except the result register number, when it is not encoded in the instruction:

- **Traps**
  The result register number is set equal to r1 as the trap handlers write their results in r1 by default (32 bit precision result);

- **Stores**
  Stores do not produce any result in the reorder buffer; they are "dummy" store entries (i.e entries where no result is written), only created for synchronization with the store buffer. The result register number is set equal to r0 . Generally speaking, all instructions that produce no result have their result register number set to r0 in their reorder buffer entry;

- **Branches**
  As for stores, the result register is set equal to r0. Branches are also created an entry in the other reorder buffer, for synchronization of the two reorder buffers;

- **Floating point compares**
  The result register number is set equal to the floating point status register number (FP_STATUS);

- **Jump and Link instructions**
  The result register must be r31 (return address register).

4.4 Execute Functions

The execute functions are gathered in supercalc.c and supersim.c and they can be separated into two groups:

- those which control multiple instruction issuing from the instruction windows.
- those which manage memory accesses.

4.4.1 Instruction Issue

Instruction issue involve several procedures, the main ones are:

- **Execute**, which implements the issue logic, traverses each instruction window, and selects ready instructions for issue;

- **ExecuteInteger** and **ExecuteFloat**, which are called by Execute to effectively compute each ready operation, based on its opcode, and to write the result in the appropriate reorder buffer ;
- **BranchExecute**, which is called by ExecuteInteger and ExecuteFloat to handle the particular case of branch execution, and wrong prediction "repairs";
- **RemoveOperation**, which removes operations from instruction windows, once they have been issued by *Execute*. It also removes flushed operations.

4.4.2 Instruction Selection

Execute looks at each operation in the instruction window from bottom to top (so as to prioritize older instructions), and declare that an instruction is ready for issue if:

- its corresponding reorder buffer entry is not flushed: if it is flushed, the operation is removed from the instruction window;
- both its firstOperand and secondOperand valid bits are set;
- one functional unit of the type required by the operation is free;

If the operation is not flushed nor ready to issue, nothing happens: *Execute* simply looks at the next operation in the instruction window.

4.4.3 Instruction Execution

If the operation is ready for issue, its reorder buffer entry is linked to the list of pending operations (*listOfExec*). As all necessary operands are usable, the result of the operation is computed and placed in the reorder buffer entry. But it cannot be made available yet: the latency of the operation has to be modelled. The ready field is used for that purpose: it indicates the clock cycle when the result will be available.

The result of the instruction execution simply depends on the opcode:

- **Branches**
  The outcome of the branch is given by the value of the first operand. There are two cases:

- **if the branch prediction option is off**: the new PC to fetch is calculated, based on this outcome; the fetch process which was stopped when the branch was encountered, is restarted;
- **if the branch prediction option is on**: then we need to
  * refine the prediction information in the branch target buffer for this branch, based on this new branch outcome. For this, we access the branch entry in the branch target buffer;
  * repair any wrong prediction. For all entries following the branch in its own reorder buffer, we set the flush bit to one. For all loads and stores flushed, provision must be made also to flush the corresponding entries in the store/load buffers. Moreover, some flushed instructions may be currently in execution in some functional units: these functional units are freed.
    There may be also instructions "younger" than the branch in the reorder buffer of the other operational unit: we use the dummy entry that has been created for the branch in the other reorder buffer, and we flush all instructions following the branch in this buffer. The instruction queue is then emptied, and the new PC is calculated, based on the effective outcome of the branch. Then the fetch process is allowed to restart only one cycle later.

- **Direct Jumps**
  Only "jump and link" operations have to be taken care of, as "jump" instructions are entirely handled at fetch stage. In the instruction reorder buffer entry, we write the address of the instruction following the jump in program sequence ;

- **Indirect Jumps**
  A flag is raised to restart the fetch process which was stopped when the jump was fetched. The new PC is calculated, given by the operation first operand. Besides, for "jalr" instructions, the return address is written in the reorder buffer;

- **Loads and Stores**
  They enter their first phase of execution: the address calculation, in an address unit (INT_DATA, FP_DATA). The address is placed in the corresponding instruction entry in the store/load buffer. No result is written in the reorder buffer;

- **Others**
  It would be too long to detail each remaining case of opcode.

  - For compare instructions (FSET, SET), the comparison is made between the value fields of firstOperand and secondOperand and the result of this comparison (0 or 1) is placed in operation reorder buffer entry;

- For arithmetic instructions, the operation which is specified by the opcode and which uses both firstOperand and secondOperand is computed; then the result is written in the operation reorder buffer entry.

4.4.4 Memory Access

The reader may wonder why the memory access description is included in the execute process section: in fact memory access is a separate and independent process. However, we might want to consider memory as a functional unit, where instructions are sent and results come back. Then, describing "memory operation" executions in this part of the document can be considered appropriate: memory access is just a second phase of execution for loads and stores, after their memory addresses calculation.

Two procedures mainly handle memory accesses:

- **SendLoads**, which traverses the load buffer from bottom to top, to select and send to cache ready load operations (i.e load operations whose address calculation is complete);
- **SendStores**, which traverses the store buffer from bottom to top, to select and send to cache ready store operations (i.e store operations whose address calculation is complete and whose data to store has been computed).

Load access

SendLoads is called first: the number of memory accesses per clock cycle is limited and loads have a higher priority. When the procedure completes, it might well be that the memory access quota has been reached: in that case SendStores will not be called. In any cases, the number of load accesses performed determines how many stores can be released to the cache.

A load in the load buffer can be sent to cache if:

- it has not been flushed: if the *flush* bit is set, the load is simply removed from the buffer;
- its address calculation is complete: the *release* bit is set;
- it does not bypass a store that accesses the same memory location. To check this, the store buffer is traversed from bottom to top. Bypassing is forbidden when the load memory address conflicts with the memory address of an older store still in the store buffer. Memory conflicts between a store and a load can be of two kinds:

- explicit memory conflicts: the store and load have identical memory addresses;
- non explicit memory conflicts: the memory addresses differ by one word. This is the case when the store writes a double word from which the load accesses the upper part, or else when the store writes a word which is the upper part of a double word read by the load.

Loads do not have to proceed in order to the cache: so if a load does not fulfill the above criteria, the traversal of the load buffer continues. The process only stops when the number of memory accesses of the current clock cycle has reached its upper limit, or when there is no more loads to process in the load buffer.

If a load passes the above checks, its processing then depends on its opcode, which characterizes

- the size (byte, halfword, word, double) of the data to read;
- its sign (signed, unsigned);
- its type (integer, float).

The value at the load memory address is read from the memory array, and is then transformed according to the characteristics of the data to load (most of the read process is reused from DLXsim).

The loaded data is then placed in the reorder buffer.

Store Access

Each store examined by **SendStores** in the store buffer is sent to cache if:

- It has not been flushed: if the *flush* bit is set, the store is simply removed from the buffer;
- Its address calculation is complete, and the corresponding store entry has appeared at the head of the reorder buffer: the *release* bit is set.

Stores have to proceed strictly in order to the cache: so if one store does not fulfill the above criteria, the store process stops. It also stops when the number of memory accesses of the current clock cycle has reached its upper limit, or when there is no more stores in the store buffer.

If the checks are passed, the store handling depends on its opcode, which specifies the size of the data to store and its type. The DLXsim function *WriteMem* is used which writes the data at the memory location, in the specified number of bytes. For doubles it is called a second time, as it must access two words.

4.5 Write Back Functions

In the write back process, we include:

- the **validation of just completed results** in the reorder buffers (i.e results whose computation ends at the current clock cycle). This is done by the **WriteBack** function;
- the **forwarding of just validated results** to the non yet valid operands of the instruction window, and to the non yet valid data to store of the store buffer. This is done by the **ForwardOperand** function.

The write back process takes place in one function: WriteBack. The goal of the function is to validate the completed results in the reorder buffers.

It traverses the list of pending operations within each reorder buffer.

Each traversed entry is examined and is selected for validation if:

- its *ready* field value is equal to the current clock cycle count: this means the computation made by the functional unit is complete;
- its *flush* bit is not set;

In that case, the *valid* field of the reorder buffer entry is set to one, meaning that the *result* value is available and can be used by other instructions: then the entry is treated depending on the value of the *class* field:

- **Loads**
  The functional unit field of a load entry indicates in which processing phase (address calculation, memory access) the load is.
  - if the load has just completed its address calculation phase, it can enter its second processing step: memory access. Basically,

        * its *unit* field is changed to indicate that it is entering its memory phase;
        * its *ready* and *valid* field are reset;
        * the *release* field of its corresponding load buffer entry is set to one, so it can be sent to memory by SendLoads;
    - if the load is in its memory phase: its entry, which contain the read data, can be validated;

- **Stores**
  A store which is in the list of pending operations is in its address calculation phase. It will only enters its second processing step, memory access, when it arrives at the head of the reorder buffer;

- **others**
  The functional unit where the operation was executed is freed and the entry is validated.

4.6 Result Commit Functions

ResultCommit is the function responsible for committing instructions from the reorder buffers. It is called successively for the integer reorder buffer, then for the floating point reorder buffer.

In each buffer, the result commit mechanism examines the instruction at the head of the reorder buffer.

There are several conditions for this instruction to be allowed to commit:

- it must be **valid** (i.e its computation his over);
- it must not be **flushed** (i.e it is on a valid branch path);

Once these conditions are satisfied, the process depends again on the instruction class:

- **Stores**
  When a store arrives at the head of the reorder buffer, it is allowed to proceed to the cache: the *release* field of its corresponding store buffer entry is set to one. The store may then be sent to memory in a future execute stage. This way we ensure that the store executes after all instructions of the same operational unit have completed, and most of all, that the store is released only if it is on a correct branch path.

- **Traps**
  The traps are, like stores, instructions allowed to execute only when they arrive at the head of their reorder buffer (the integer reorder buffer). That is why the execution of traps is handled by the result commit process. The point is that traps should not bypass any of the integer stores: this is because the trap arguments are pushed onto the stack by integer stores, and those arguments are needed by the trap handler to be passed to the operating system. To insure correct handling of traps, their process has been cut into three phases:

  - a synchronization phase (in one or more cycles), which insures that the trap does not bypass any store in the store buffer, as stores which have been retired from the reorder buffer may still be waiting in the store buffer;
  - an execution phase (one cycle) where the trap is executed by the trap handler (Handle_Trap) . The result of the trap is placed in the reorder buffer. And it is validated;
  - a commit phase where the trap can safely commit: the result of the trap is placed in R1 by default.

- **other instructions**
  The other cases are either instructions producing no results in the reorder buffer (jumps, dummy branches...) or instructions that produce results to be written in the register file. In that case the result is written in the destination register.

Every time an instruction commits, its entry is cleared out from the reorder buffer: the head of the reorder buffer moves backwards in the buffer.

5 User Interface Description

5.1 Overall Functional Organization

Until now, we have described the core of superDLX: the simulated hardware mechanisms and hardware elements. The user interface needs to be introduced also, as it represents a non-negligible part of the simulator. Although less complex than the simulation core, it will likely receive more attention from the users. The interface presented here is an on-line interface with a DLXsim "look and feel": when the program is run, the simulator prompt appears and the user can enter interactive commands. A prototype for an X Window interface has also been developed as part of a course project (cs 308-573A) and some windows are displayed in appendix F.

User's input is of two kinds:

- **a number of simulation parameters**; they are gathered in the optional machine configuration file;
- **a number of commands**; they allow the user to interact with the simulator within a simulation session;

Figure 8 gives the general organization of the interface. The link to the simulation core is represented in with dotted lines. There are several functions involved:

- the **machine configuration functions**: their goal is to prepare the machine structure that model all the machine data structures.
  - **Sim_Init** prepare the machine structure, and initializes all parameterizable variables with default values;
  - **read** if a machine configuration file is used, it set all the machine parameters with the values given by the file;
  - **Sim_Create** it performs parameter dependent memory allocation to create the machine elements.
- the **command functions** they are invoked by user's commands during a simulation session. A command interpreter package is used (Tcl, the same as DLXsim) to interpret the commands and call the appropriate function;
- the **main procedure** of the simulator: it coordinates the above functions and parses the user's command lines (the commands parsing is reused from DLXsim);

5.2 User Parameters

The Machine Configuration File must respect a specific format, as shown in appendix C.

This file allows to set the following values:

- the maximum number of instructions to be fetched per clock cycle;

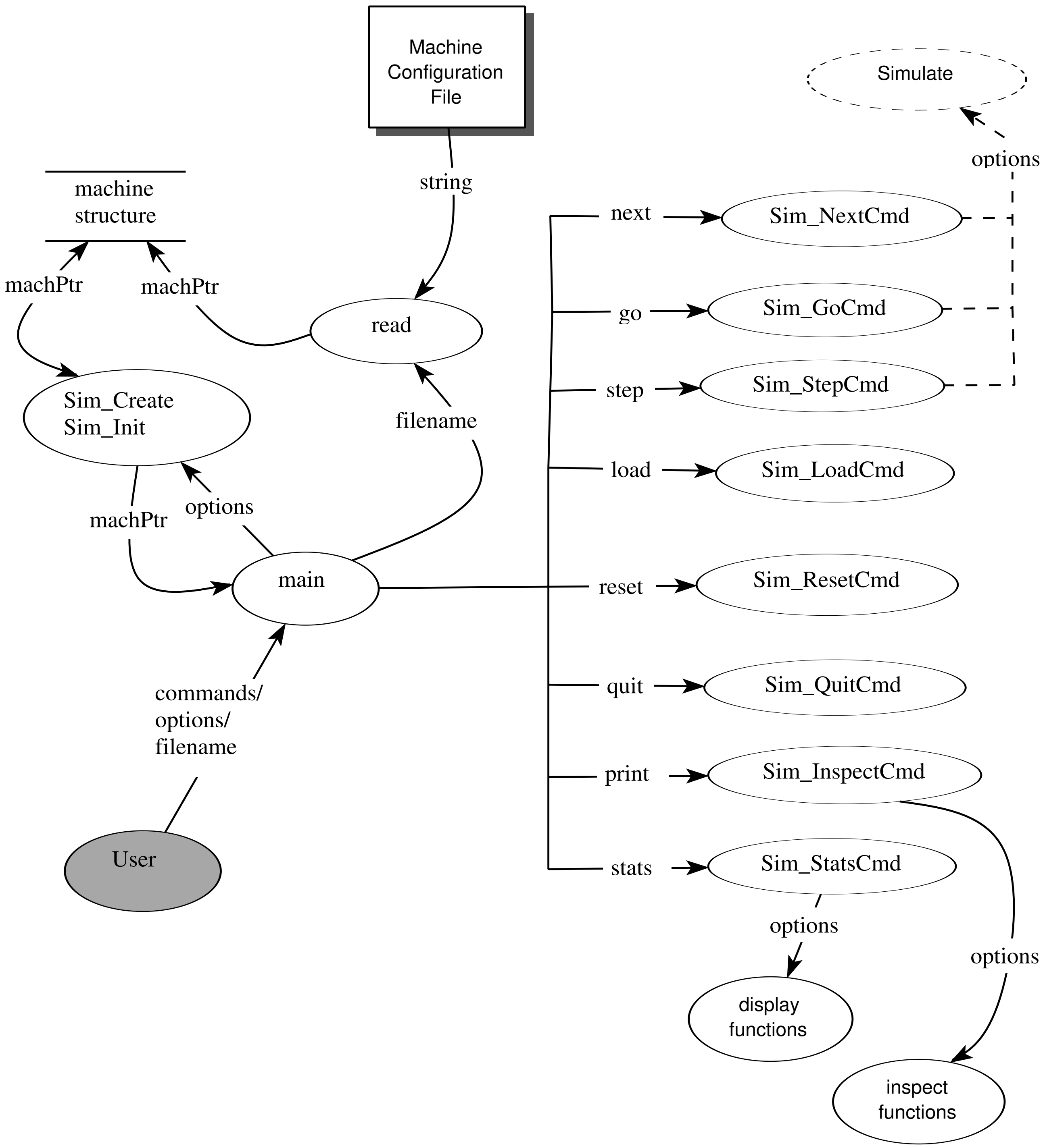

Figure 8: User Interface Functional Diagram

- the maximum number of instructions to be decoded per clock cycle;
- the maximum number of instructions to be committed per clock cycle in each operational unit.
- the memory characteristics:
  - size;
  - latency;
  - number of memory accesses per clock cycles;
- the various buffer and queues sizes:
  - the integer and floating point reorder buffers;
  - the integer and floating point central windows;
  - the load and store buffers;
  - the instruction queue;
  - the branch target buffer size;
- the integer and floating point functional unit characteristics:
  - number of available units;
  - latency of the units;

5.3 Processing User Commands

5.3.1 Load Command

This command invokes **Asm_LoadCmd** function in asm.c . It has been entirely reused from DLXsim. It loads assembly code and its data in specific sections of memory. For more information, please refer to [HB92].

5.3.2 Go Command

The go commands invokes the **Sim_GoCmd** of sim.c. This function has been partly reused from DLXsim. There are two uses of the command:

- An address can be given as argument of the command: then the Sim_GoCmd make the machine program counter equal to this address and calls **Simulate** to execute the program until the end. The address can be numeric, it can also be a label existing in the DLX assembly file. Generally, this use of the command is to start simulation of a loaded program; then the address should be the program starting address ("256" or "0x100" or "_main"). If the simulation of a program has already started when the user enters a "go address" commands, it is likely that several instructions will be "jumped" without being executed.
- If no address is given as argument, *Simulate* is called to execute the program until the end, starting at the current pc.

Clearly, there are erroneous situations that can arise; this is because, at the level of **Sim_GoCmd**, the checking of the address is limited to syntactic and bound checks. Hopefully, in cases of errors, the simulation should fail "gracefully", generating error messages and returning control to the user. Let's analyze the different erroneous situations:

- a DLX-assembly program has been loaded and part of it has been simulated. Then, the user enters a go command that makes the simulator "jump" instructions in the program sequence: then,
  - either the simulated program eventually finishes execution, without producing its correct output,
  - or the simulation eventually fails because the simulator ends up executing a store or load with a calculated data address that falls out of the data section of memory: an error message is then presented to the user.
- a DLX-assembly program has been loaded, but simulation has not started yet. Then the user enters a go command with an address which is syntactically correct, but is not the program starting address. Then there are several cases:
  - the entered starting address hits somewhere inside the program code, simulation starts but will eventually fails because the simulator ends up executing a store or load with a wrong data address: an error message is then presented to the user.
  - the entered starting address hits somewhere outside the program code in the code space, then the **FetchInstruction** functions eventually detects that too many NOPs are fetched (as memory is initialized with NOPs), then an error message is presented to the user.

- No assembly program has been loaded. The user then enters a go command with a numeric address. The simulation eventually stops and and error message is displayed.

Whenever a simulation fails, the machine structure is reset: memory is cleaned and has to be reloaded before another simulation can be started.

5.3.3 Step Command

The step command invokes the **Step_SimCmd** of sim.c. This function has been partly reused from DLXsim. In our simulator, there are two types of uses of this command:

- If an address is given as a command argument, Sim_StepCmd makes the machine program counter equal to this address and calls the **Simulate** to execute one cycle, starting at the given address. Generally, this use of the command is to start a program simulation: the address should then be the program starting address.
- If no address is given as argument, **Simulate** is called to simulate one cycle only, without changing the current pc.

Clearly also, there are several cases of abnormal uses of the step command that eventually causes a simulation to fail. They are basically the same as those exposed in the **Go Command** paragraph, as they generate similar situations: this is because a *step [address]* followed by a *go* is equivalent to a *go [address]*. So in those cases, the simulation should also fail gracefully and generate appropriate messages.

5.3.4 Next Command

It invokes the **Sim_NextCmd** function of sim.c. This function calls **Simulate** with directives that depend on the command arguments:

- if the argument specifies that a certain number of instructions should be executed, **Simulate** is called to simulate as many cycles as necessary for the execution of the specified number of instructions.;
- if the argument specifies a number of cycles to be executed, **Simulate** is called to simulate as many cycles as specified.

5.3.5 Print Command

This command is intended for the user to inspect the various buffers and queues of the superscalar simulator, during simulation.

It invokes the **Sim_InspectCmd** function of sim.c. This function calls other procedures of inspect.c that format and output the content of various data structures, on a given clock cycle:

- the instruction queue;
- the instruction windows ;
- the reorder buffers;
- the store and the load buffer;
- the number, latency and usage of each functional units;

The arguments of the user's command specify what display function should be called. If no argument is given all display functions are called.

An example of print command output is given in appendix E.

5.3.6 Stats Command

It invokes the **Sim_DumpStats** function that formats and displays the various statistics that are gathered during simulation. Again, not all the statistics are necessarily presented, the choice is made based on the command arguments. Statistics gathered during simulation are counters which are transformed for display into percentages, distribution tables.. Please refer to the user's manual for information on the output, and to appendix D for an example of output.

5.3.7 Reset Command

Sim_ResetCmd reinitializes all the elements of the processor. But user's parameters entered via the Machine Configuration file at the beginning of a simulation session are still valid.

6 Future Directions

SuperDLX was developed within the limited scope of a Master's project, so there is certainly a lot of room for improvement and modification of superDLX simulator. The goal of this section is to give some ideas for future enhancements of the simulator. This list of proposals does not intend to be exhaustive.

- **Improving or adding superscalar features**

  The implemented branch prediction mechanism permits to start execution of one conditional path of a branch. It could have been interesting to experiment branch speculation on both paths of the branch and compare the two branch processing techniques. This type of branch speculation is certainly very complex to implement: buffering of instructions has to be provided to support execution of two conditional streams of instructions at a time, which implies some kind of replication of the reorder buffer and instruction window . Duplication of hardware elements has to stay within realistic limits, and our feeling is that speculative execution will not expand across more than one branch at a time.

- **Improving the simulator interface and usability**

  The on-line interface allow the user to interact with the simulator through the use of some display and execution commands.There are others interesting commands that could be added to give to the simulator a full **debugging capability** :

  - **commands to inspect registers and memory locations**. The command(s) would return values in format specified by the user (decimal, hexadecimal, binary, floating point double, floating point simple, character, string..). Getting a register value in a context of a superscalar machine could mean getting the values of the current instances of this register in the processor;
  - **commands to modify registers or memory locations**. The user should be able to store numbers, like integers or floating points. Again, this may not work the same way as in a scalar simulator: storing a value in a register can affect all the current semantic instances of that register, present in the machine;
  - **commands to inspect machine characteristics**. The user could display parameters from the machine configuration file, and also changing values, like the current size of a window/buffer, the maximum size of a window/buffer so far ...
  - **commands to modify machine configuration parameters**;

- **commands for profiling.** Profiling commands could produce traces to be used as input to some plotting programs: for example, to study the evolution of buffer sizes over time ...
- **commands to set up break-points.** These break points would be specified by the user (by an address for example). The possibility to clear break points should be provided.

A prototype of a X Window interface has been developed as a course project (appendix F). But is does not provides all the commands that are available through superDLX on-line interface (*next, reset, stats*). If the X-window is to be fully integrated in the simulator, it should be expanded to include those missing commands.

We can think of other types of interfaces: for example, a graphical interface, displaying the flow of information and data in the different elements of the processor in real time.

- **Improving the simulator applicability** The simulators suffers from the same limitations as DLXsim, in terms of the variety of benchmarks that can be run. Many aspects should be enhanced:

  - **adding new traps.** C library functions are compiled by DLXcc into traps; during simulation, traps are processed as instructions; these instructions give control to the operating system which execute the required functions. But there is not enough traps available for DLXsim and superDLX, and a lot of benchmarks cannot be simulated because they use library functions for which no trap has been implemented (example: scanf, fscanf ...). Some traps have been added during this project (some mathematical functions), but they have to be entered "manually" in the assembly code, because they are not generated by DLXcc;
  - **adding a linking capability to DLXcc.** For the moment, DLXcc cannot link programs. So to simulate benchmarks consisting of different modules, one has to load the corresponding assembly files one by one in the simulator memory;
  - **adapting DLXcc to the simulator.** DLXcc generates code for a scalar machine. One major problem is that we could not use DLXcc optimized code, because DLXcc optimizer fills the delay slots of loads or branches with instructions from before the loads or branches. From a superscalar decoding prospective, this puts instructions out of order and is not an appropriate optimization. It would be interesting to study what compiler optimizations the superscalar simulator could exploit, and implement them in DLXcc.

A User Manual

NAME

superdlx - The DLX Superscalar Simulator

OPTIONS

[-bp] [-f *file name*] [-ms#]

-bp Set the branch prediction option.

-f Select the machine configuration file where user parameters are defined. If this option is not used, default parameters are taken.

-ms Select the memory size (in bytes). This argument overrides any memory size specification from the machine configuration file.

-X Run the X-Window interface (prototype).

example
superdlx -bp -f machine

DESCRIPTION

superDLX is an interactive program that simulates a superscalar machine, using the DLX instruction set, and part of the DLXsim simulation environment. When the program is invoked, this opens a simulation session: the simulator prompt - (superdlx) - is displayed, waiting for user's input: DLX assembly programs (compiled with DLXcc) can then be loaded, and simulation can be controlled by the user's command on standard input.

COMMANDS

go [*address*]

If followed by an address, it makes the program counter point to that address and starts simulating the loaded program from there. Otherwise, it continues execution from where it last stopped. Addresses are labels or numeric values (decimal or hexadecimal notation). Normally, they specify the beginning of the loaded program.
Syntactically wrong addresses, as well as addresses that are not in the code space are detected before simulation. But errors can be caused by bad, though syntactically correct, starting addresses: they are detected *after* simulation has started. In that case the simulation is interrupted and the machine is reset.
If no error occurs, simulation does not stop before the end of the loaded program. In any cases, a message is displayed which explains why simulation stopped.

examples
(superdlx) go _main
(superdlx) go 0x100
(superdlx) go 256
(superdlx) go

help [*command name*]
When followed by a *command name*, it gives information on the command and how to use it. Otherwise, it lists all the available commands.
examples
(superdlx) help
(superdlx) help go

load *file file file* ...
As in DLXsim, it reads each of the given *files*. It treats them as DLX assembly language files and loads memory as indicated in the files. Code (text) is normally loaded starting at address 0x100, but the **codeStart** variable may be used to set a different starting address. Data is normally loaded starting at address 0x1000, but a different starting address may be specified in the **dataStart** variable. The return value is either an empty string or an error message describing problems in reading the files. A list of directives that the loader understands is in a later section of this manual.
example
(superdlx) load fdata.s f.s

next [**#c**] [**#i**]
It executes a single cycle, a specified number of cycles, or the number of cycles necessary to fetch a number of instructions. One argument, if present, may specify how the command behaves:

#c gives the number of cycle to execute;

#i gives the number of instructions to execute;

If no argument is given , a single cycle is executed (like *step* command)
examples
(superdlx) next 20i
(superdlx) next 5c
(superdlx) next

print **[queue]** **[window]** **[reorder]** **[data]** **[unit]** **[all]** **[help]**
This command displays the content of the machine elements specified by the options,

at the clock cycle where simulation last stopped. Any combination of options may be selected. The options and their results are as follows:

queue display the instruction queue in a table, with the following column headers:

- #: entry number
- **icount**: dynamic instruction count
- **address**: instruction address in memory
- **opcode**: the instruction opcode
- **rs1**: register source 1 (I-type and R-type instructions)
- **rs2**: register source 2 (R-type instructions)
- **rd**: register destination (I-type and R-type instructions)
- **extra**: immediate or offset (I-type and J-type instructions)

window print both instruction windows (integer and floating point), with the following column headers:

- #: instruction entry number in the window
- **opcode**: instruction opcode
- **address**: instruction address (program counter)
- **rb_entry**: reorder buffer entry number of the instruction
- **operand1**: contains either the value of the first operand or a reorder buffer entry from where the value will be taken when computed
- **ok1**: the validity field of the first operand: it contains 0 if the operand refers to a reorder buffer entry and 1 if the operand value is available
- **typ1**: type of the first operand (INT: integer; FPS: floating point simple; FPD: floating point double). The type of the operand indicates from which reorder buffer the operand value can be taken.
- **operand2**: second operand; just like the first operand, it can contain either a value or a reorder buffer entry. However, for I-type or J-type instructions, it contains immediate or offset values.
- **ok2**: validity field of the second operand. (always 1 for immediates and offsets).
- **typ2**: type of the second operand (same as type1, plus IMM: immediate).
- **pred**: only for branch instructions, indicates what prediction has been made at fetch stage (PT: predict taken; PNT: predict not taken)

reorder display the integer and the floating point reorder buffer, in tables with the following column headers:

- #: instruction entry number

- **icount**
- **opcode**
- **reg#**: result register number
- **unit**: functional unit where the instruction executes
- **result**: instruction result when produced (some instructions such as stores produce no result)
- **ok** : validity of the result;
- **ready**: indicates at what clock cycle the computation of the result will be over;
- **flsh**: the flush bit; it is set to 1 when the instruction follows a wrong branch prediction. In that case, when the instruction arrives at the head of the reorder buffer, it is discarded.

data display the load and store buffers, in tables with the following headers:

- #
- **opcode**
- **icount**
- **address**: the accessed memory from where data is read or written.
- **data to store**: (only for stores) it contains the value to write in memory, or a reorder buffer entry.
- **type**: (only for stores) type of the data to store (INT, FPS, FPD)
- **ok**: (only for stores) validity bit of the data to store
- **rel**: for stores, this field indicates if the data is allowed to be written in memory.
- **flsh**: set to 1 if the instruction follows a wrong branch prediction.

unit show the latency, number, and current usage of the functional units.

help print help information on how to interpret the displayed data, for any of the above machine elements. The machine components for which information is requested must be passed as arguments to the commands. If none of the components is specified, information is provided for all of them.

example

(superdlx) print queue window unit
(superdlx) print
(superdlx) print queue help

quit

Exit the simulator.

reset

Reset the machine memory and empties its components: for another simulation to take place, the user needs to load again a DLX-program.

stats [**-p**#] [**-f** *file name*] [**-g**] [**-r**] [**-i**] [**-b**] [**-a**]

This command displays the various statistics that are gathered during simulation. Options allow to filter the statistics (there is a lot of figures!) or to redirect the output:

-f Redirect the output to a file. The file name is specified one space after *-f*.

-p Set the minimum value for percentages: percentages lower than this value will not be output: this is to limit the number of figures. The value is specified just after *-p*, without spacing: it can be a positive integer or real.

-g General information display:

- Number of fetched, decoded, issued instructions
- Number of committed instructions, writes to registers and useless writes. A useless write is a write to a register that could have been avoided because the writing instruction is followed in the reorder buffer by another instruction writing to the same register.
- Per cycle rates of fetch, decode, issue, commit.
- Branch information: correct and wrong prediction (if branch prediction is performed).
- Fetch stalls.

-r Information on renaming:

- Renamed Operands: a table gives the distribution of the number of operands renames per clock cycle.
- Operands searching information: a table gives the distribution of the number of operands searched (in the reorder buffer and register file) per clock cycle;

-i Instruction process information:

- Instruction issue distribution table, per clock cycle.
- Instruction delay distribution table: the distribution of the number of cycles that instructions have to wait in the central window, before issue.
- Instruction commit distribution table, per clock cycle.

-b Occupancy rate tables of the instruction windows and the reorder buffers, throughout simulation;

-a Display of all the above statistics: this is also the default when no filter option is given.

example
(superdlx) stats -i -p1.0 -f output
(superdlx) stats

step [*address*]

If no *address* is given, the **step** command executes a **single** cycle, continuing from wherever simulation had previously stopped. If *address* is given, then the program counter is changed to point to *address*, and a **single** cycle is executed from there. In either case, an information message after the step execution. If an error take place.

SIMULATION SPEED

The simulation run time varies depending on the simulation configuration parameters. The sizes of the buffers (instruction queue, reorder buffers, instruction windows) have a significant impact on the simulation time. For example, with buffer sizes of 60, and with branch prediction, a rate of about 5,000-6,000 committed instructions per seconds (wall clock) is achieved on a Sparc ELC workstation. But when the buffer sizes are set equal to 200, the rate drops to 3,000-4,000 instructions per seconds.

B The Opcode Description Table

Name	Oper.unit	Class	Func.unit	First oper.	Second oper.	Result
ADD	INTEGER	ALU	INT_ALU	INTEGER	INTEGER	INTEGER
ADDI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
ADDU	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
ADDUI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
AND	INTEGER	ALU	INT_ALU	INTEGER	INTEGER	INTEGER
ANDI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
BEQZ	INTEGER	BRANCH	INT_BRANCH	INTEGER	IMMED.	NONE
BNEZ	INTEGER	BRANCH	INT_BRANCH	INTEGER	IMMED.	NONE
J	INTEGER	DIR_JUMP	INT_BRANCH	NONE	NONE	NONE
JAL	INTEGER	DIR_JUMP	INT_BRANCH	NONE	NONE	INTEGER
JALR	INTEGER	INDIR_JUMP	INT_BRANCH	INTEGER	NONE	INTEGER
JR	INTEGER	INDIR_JUMP	INT_BRANCH	INTEGER	NONE	INTEGER
LB	INTEGER	LOAD	INT_DATA	INTEGER	IMMED.	INTEGER
LBU	INTEGER	LOAD	INT_DATA	INTEGER	IMMED.	INTEGER
LH	INTEGER	LOAD	INT_DATA	INTEGER	IMMED.	INTEGER
LHI	INTEGER	SHIFT	INT_SHIFT	INTEGER	IMMED.	INTEGER
LHU	INTEGER	LOAD	INT_DATA	INTEGER	IMMED.	INTEGER
LW	INTEGER	LOAD	INT_DATA	INTEGER	IMMED.	INTEGER
MOVFP2I	INTEGER	MOVE	INT_ALU	FP_SIMPLE	NONE	INTEGER
MOVI2S	INTEGER	MOVE	INT_ALU	INTEGER	NONE	INTEGER
MOVS2I	INTEGER	MOVE	INT_ALU	INTEGER	NONE	INTEGER
OR	INTEGER	ALU	INT_ALU	INTEGER	INTEGER	INTEGER
ORI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
RFE	OTHER	NONE	NONE	NONE	NONE	NONE
SB	INTEGER	STORE	INT_DATA	INTEGER	IMMED.	INTEGER
SEQ	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SEQI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SEQU	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SEQUI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER

Name	Oper.unit	Class	Func.unit	First oper.	Second oper.	Result
SGE	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SGEI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SGEU	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SGEUI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SGT	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SGTI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SGTU	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SGTUI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SH	INTEGER	STORE	INT_DATA	INTEGER	IMMED.	INTEGER
SLE	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SLEI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SLEU	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SLEUI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SLL	INTEGER	SHIFT	INT_SHIFT	INTEGER	INTEGER	INTEGER
SLLI	INTEGER	SHIFT	INT_SHIFT	INTEGER	IMMED.	INTEGER
SLT	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SLTI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SLTU	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SLTUI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SNE	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SNEI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SNEU	INTEGER	SET	INT_COMP	INTEGER	INTEGER	INTEGER
SNEUI	INTEGER	SET	INT_COMP	INTEGER	IMMED.	INTEGER
SRA	INTEGER	SHIFT	INT_SHIFT	INTEGER	INTEGER	INTEGER
SRAI	INTEGER	SHIFT	INT_SHIFT	INTEGER	IMMED.	INTEGER
SRL	INTEGER	SHIFT	INT_SHIFT	INTEGER	INTEGER	INTEGER
SRLI	INTEGER	SHIFT	INT_SHIFT	INTEGER	IMMED.	INTEGER
SUB	INTEGER	ALU	INT_ALU	INTEGER	INTEGER	INTEGER
SUBI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
SUBU	INTEGER	ALU	INT_ALU	INTEGER	INTEGER	INTEGER
SUBUI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
SW	INTEGER	STORE	INT_DATA	INTEGER	IMMED.	INTEGER
TRAP	INTEGER	TRAP	NONE	NONE	IMMED.	INTEGER
XOR	INTEGER	ALU	INT_ALU	INTEGER	INTEGER	INTEGER
XORI	INTEGER	ALU	INT_ALU	INTEGER	IMMED.	INTEGER
NOP	OTHER	NONE	NONE	NONE	NONE	NONE

Name	Oper.unit	class	Func.unit	First oper.	Second oper.	result
ADDD	FLOAT	FADD	FP_ADD	FP_DOUBLE	FP_DOUBLE	FP_DOUBLE
ADDF	FLOAT	FADD	FP_ADD	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
BFPF	FLOAT	FBRANCH	FP_BRANCH	FLOAT	IMMED.	FP_SIMPLE
BFPT	FLOAT	FBRANCH	FP_BRANCH	FLOAT	IMMED.	FP_SIMPLE
CVTD2F	FLOAT	FCONV	FP_CONV	FP_SIMPLE	NONE	FP_DOUBLE
CVTD2I	FLOAT	FCONV	FP_CONV	FP_DOUBLE	NONE	FP_SIMPLE
CVTF2D	FLOAT	FCONV	FP_CONV	FP_SIMPLE	NONE	FP_DOUBLE
CVTF2I	FLOAT	FCONV	FP_CONV	FP_SIMPLE	NONE	FP_SIMPLE
CVTI2D	FLOAT	FCONV	FP_CONV	FP_SIMPLE	NONE	FP_DOUBLE
CVTI2F	FLOAT	FCONV	FP_CONV	FP_SIMPLE	NONE	FP_SIMPLE
DIV	FLOAT	FDIV	FP_DIV	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
DIVD	FLOAT	FDIV	FP_DIV	FP_DOUBLE	FP_DOUBLE	FP_DOUBLE
DIVF	FLOAT	FDIV	FP_DIV	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
DIVU	FLOAT	FDIV	FP_DIV	FP_SIMPLE	IMMED.	FP_SIMPLE
EQD	FLOAT	FSET	FP_COMP	FP_DOUBLE	FP_DOUBLE	FP_SIMPLE
EQF	FLOAT	FSET	FP_COMP	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
GED	FLOAT	FSET	FP_COMP	FP_DOUBLE	FP_DOUBLE	FP_SIMPLE
GEF	FLOAT	FSET	FP_COMP	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
GTD	FLOAT	FSET	FP_COMP	FP_DOUBLE	FP_DOUBLE	FP_SIMPLE
GTF	FLOAT	FSET	FP_COMP	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
LED	FLOAT	FSET	FP_COMP	FP_DOUBLE	FP_DOUBLE	FP_SIMPLE
LEF	FLOAT	FSET	FP_COMP	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
LTD	FLOAT	FSET	FP_COMP	FP_DOUBLE	FP_DOUBLE	FP_SIMPLE
LTF	FLOAT	FSET	FP_COMP	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
LD	FLOAT	FLOAD	FP_DATA	INTEGER	IMMED.	FP_DOUBLE
LF	FLOAT	FLOAD	FP_DATA	INTEGER	IMMED.	FP_SIMPLE
MOVD	FLOAT	FMOVE	FP_ADD	FP_DOUBLE	NONE	FP_DOUBLE
MOVF	FLOAT	FMOVE	FP_ADD	FP_SIMPLE	NONE	FP_SIMPLE
MOVI2FP	FLOAT	FMOVE	FP_ADD	INTEGER	NONE	FP_SIMPLE
MULT	FLOAT	FMULT	FP_MUL	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
MULTD	FLOAT	FMULT	FP_MUL	FP_DOUBLE	FP_DOUBLE	FP_DOUBLE
MULTF	FLOAT	FMULT	FP_MUL	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
MULTU	FLOAT	FMULT	FP_MUL	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
NED	FLOAT	FSET	FP_COMP	FP_DOUBLE	FP_DOUBLE	FP_SIMPLE
NEF	FLOAT	FSET	FP_COMP	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE
SD	FLOAT	FSTORE	FP_DATA	INTEGER	IMMED.	FP_DOUBLE
SF	FLOAT	FSTORE	FP_DATA	INTEGER	IMMED.	FP_SIMPLE
SUBD	FLOAT	FADD	FP_ADD	FP_DOUBLE	FP_DOUBLE	FP_DOUBLE
SUBF	FLOAT	FADD	FP_ADD	FP_SIMPLE	FP_SIMPLE	FP_SIMPLE

C An Example of a Machine Configuration File

```
Instructions_process_per_cycle
        fetch            4
        decode           4
        commit           4



Memory
        size            65536
        latency         1
        accesses        4



Reorder_Buffer_sizes
        integer         20
        float           20



Instruction_Window_sizes
        integer         20
        float           20

Data_Buffer_sizes
        load            5
        store           5

Instruction_Queue_size  20
Branch_Buffer_size      111

Integer_Functional_Units
alu
        number          4
        latency         1

shift
        number          2
```

```
        latency         1
comp
        number          1
        latency         1
address
        number          4
        latency         1
branch
        number          1
        latency         1

Floating_Point_Units
add
        number          4
        latency         2
mult
         number         4
        latency         5
div
        number          2
        latency         10
convt
        number          2
        latency         2
comp
         number         1
        latency         1
address
         number         4
        latency         1
branch
        number          1
        latency         1
```

D An Example of Statistical Output

This is the print out of the statistics gathered during simulation of the tomcatv benchmark (a vectorized version of a mesh algorithm). The simulator was run with the branch prediction flag on, and the machine configuration file of appendix B. The *stats* command is run with the *-p1* option, meaning that lines where percentages are less than 1 are not displayed in the distribution tables

```
(superdlx) stats -p1
Number of cycles: 343946

Instructions fetched         1038812
Instructions decoded         1014576    (97.67% of total fetched)

Instructions issued          1009097    (97.14% of total fetched)
-> integers                   845844    (83.82% of total issued)
-> floating points            163253    (16.18% of total issued)

Instructions commited         994182    (95.70% of total fetched)
-> integers                   831085    (83.59% of total commited)
-> floating points            163097    (16.41% of total commited)
-> writes to registers        955513    (96.11% of total commited)
-> useless writes             954262    (99.87% of total writes)

Loads Blocked by Stores:      10647 ( 5.51% of total loads)

Per Cycle Rates
-> fetch        3.0 instructions/cycle
-> decode       2.9 instructions/cycle
-> issue        2.9 instructions/cycle
-> commit       2.9 instructions/cycle

Number of branches: 9397, taken 4574 (48.68%), untaken 4823 (51.32%)
Correct predictions             8280 (88.11% of total branches)
-> taken                        3725 (81.44% of total taken branches)
-> not taken                    4555 (94.44% of not taken branches)
Wrong predictions
```

```
-> lost cycles                   3972 ( 1.15% of total cycles)
-> flushed instructions         13447 ( 1.29% of total fetched)

Fetch Stalls: 23517 (6.84% of total cycle count)
Fetch stalls due to full buffers: 22242 (94.58% of total stalls)

 **Operands Renaming (flow/anti-dependencies)

Renamed operands:          80.52%  of total operands
-> integers                78.02% of integer operands
-> floats                  99.98% of floating point operands
```

NUM	BOTH UNITS total	BOTH UNITS %	INTEGER UNIT total	INTEGER UNIT %	FLOATING POINT UNIT total	FLOATING POINT UNIT %
0	91387	26.57%	92951	27.02%	300091	87.25%
1	78798	22.91%	81388	23.66%	768	0.22%
2	40117	11.66%	42503	12.36%	11631	3.38%
3	25339	7.37%	32943	9.58%	18403	5.35%
4	21538	6.26%	26188	7.61%	7751	2.25%
5	21152	6.15%	25828	7.51%	1937	0.56%
6	16152	4.70%	8153	2.37%	1325	0.39%
7	8492	2.47%	4171	1.21%	2040	0.59%
8	11510	3.35%	11057	3.21%	0	0.00%
9	9692	2.82%	8286	2.41%	0	0.00%
10	3965	1.15%	4088	1.19%	0	0.00%
11	4623	1.34%	1398	0.41%	0	0.00%
12	5776	1.68%	1777	0.52%	0	0.00%

```
("NUM": number of operands renamed ;"total": total number of clock cycles;
 "%": percentage of clock cycles)
```

**Operands Searching Information

NUM	BOTH UNITS total	BOTH UNITS %	INTEGER UNIT total	INTEGER UNIT %	FLOATING POINT UNIT total	FLOATING POINT UNIT %
0	75985	22.09%	76815	22.33%	300089	87.25%
1	89845	26.12%	91309	26.55%	769	0.22%
2	30696	8.92%	33412	9.71%	11622	3.38%
3	11138	3.24%	11978	3.48%	18413	5.35%
4	29627	8.61%	30997	9.01%	7751	2.25%
5	25581	7.44%	26308	7.65%	1928	0.56%
6	13094	3.81%	17143	4.98%	1334	0.39%
7	13488	3.92%	14857	4.32%	2040	0.59%
8	4251	1.24%	6625	1.93%	0	0.00%
9	7695	2.24%	6671	1.94%	0	0.00%
10	11961	3.48%	11120	3.23%	0	0.00%
11	7474	2.17%	3520	1.02%	0	0.00%
12	4027	1.17%	3398	0.99%	0	0.00%
13	8176	2.38%	4131	1.20%	0	0.00%

("NUM": number of operands searched;"total": total number of clock cycles;
"%" : percentage of clock cycles)

**Instruction Issue Distribution

NUM	BOTH UNITS total	BOTH UNITS %	INTEGER UNIT total	INTEGER UNIT %	FLOATING POINT UNIT total	FLOATING POINT UNIT %
0	14853	4.32%	26289	7.64%	234000	68.03%
1	34753	10.10%	38129	11.09%	68813	20.01%
2	85030	24.72%	107706	31.31%	33899	9.86%
3	94360	27.43%	110010	31.98%	4454	1.29%
4	72280	21.01%	51768	15.05%	620	0.18%
5	30280	8.80%	6624	1.93%	2160	0.63%
6	7566	2.20%	2579	0.75%	0	0.00%

("NUM": number of instructions issued:"total": total number of clock cycles;
"%" : percentage of clock cycles

**Issue Delay Distribution

	BOTH UNITS		INTEGER UNIT		FLOATING POINT UNIT	
NUM	total	%	total	%	total	%
1	277008	27.45%	231793	22.97%	45215	4.48%
2	243197	24.10%	230827	22.87%	12370	1.23%
3	151774	15.04%	128835	12.77%	22939	2.27%
4	74978	7.43%	61202	6.07%	13776	1.37%
5	61984	6.14%	49738	4.93%	12246	1.21%
6	52335	5.19%	43587	4.32%	8748	0.87%
7	48141	4.77%	37805	3.75%	10336	1.02%
8	44191	4.38%	32263	3.20%	11928	1.18%
9	25302	2.51%	15364	1.52%	9938	0.98%
10	13806	1.37%	7636	0.76%	6170	0.61%

("NUM": number of clock cycles;"total": total number of instructions;
"%": percentage of issued instructions)

**Instruction Commit Distribution

	BOTH UNITS		INTEGER UNIT		FLOATING POINT UNIT	
NUM	total	%	total	%	total	%
0	0	0.00%	17617	5.12%	208991	60.76%
1	112598	32.74%	132771	38.60%	40535	11.79%
2	23698	6.89%	10856	3.16%	30924	8.99%
3	43367	12.61%	46116	13.41%	5815	1.69%
4	24143	7.02%	12531	3.64%	3662	1.06%
5	42199	12.27%	39406	11.46%	308	0.09%
6	13211	3.84%	10769	3.13%	8	0.00%
8	10756	3.13%	8556	2.49%	1506	0.44%

("NUM": number of clock cycles;"total": total number of instructions;
"%": percentage of commited instructions)

**Occupancy of the Integer Buffers

%OCC	INSTRUCTION WINDOW total cc	INSTRUCTION WINDOW %cc	REORDER BUFFER total cc	REORDER BUFFER %cc
+ 0%	13512	3.93%	4322	1.26%
+10%	26463	7.69%	750	0.22%
+20%	44573	12.96%	3997	1.16%
+30%	70198	20.41%	3536	1.03%
+40%	86503	25.15%	2044	0.59%
+50%	48609	14.13%	2126	0.62%
+60%	37625	10.94%	3716	1.08%
+70%	13176	3.83%	8997	2.62%
+80%	2686	0.78%	6275	1.82%
+90%	601	0.17%	13827	4.02%
100%	0	0.00%	294356	85.58%

**Occupancy of the Floating Point Buffers

%OCC	INSTRUCTION WINDOW total cc	INSTRUCTION WINDOW %cc	REORDER BUFFER total cc	REORDER BUFFER %cc
+ 0%	163283	47.47%	75778	22.03%
+10%	101546	29.52%	72405	21.05%
+20%	40568	11.79%	71042	20.65%
+30%	24349	7.08%	54074	15.72%
+40%	11027	3.21%	24121	7.01%
+50%	3173	0.92%	8264	2.40%
+60%	0	0.00%	7330	2.13%
+70%	0	0.00%	10052	2.92%
+90%	0	0.00%	5040	1.47%
100%	0	0.00%	13680	3.98%

E A Running Example

In this example of simulation, we use the following little programs, that are loaded in the machine memory:

- fdata.s

```
.data   0
.global a
a:      .double 3.14159265358979
.global x
x:      .double 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
        .double 17,18,19,20,21,22,23,24,25,26,27
.global xtop
xtop:   .double 28
```

- f.s

```
.text   256
start:  ld      f2,a
        add     r1,r0,xtop
loop:   ld      f0,0(r1)
        multd   f4,f0,f2
        sd      0(r1),f4
        ld      f6,-8(r1)
        multd   f8,f6,f2
        sd      -8(r1),f8
        ld      f10,-16(r1)
        multd   f12,f10,f2
        sd      -16(r1),f12
        ld      f14,-24(r1)
        multd   f16,f14,f2
        sd      -24(r1),f16
        sub     r1,r1,#32
        bnez    r1,loop
        nop
        trap    #0
```

The simulator is run with the machine configuration file of appendix C, and the branch prediction option is set. The two assembly files are loaded in the simulator memory. Simulation is started for the 6 first clock cycles.

```
%superdlx -bp -f 605/machine

 ** Superscalar Version of the DLX Simulator

(C) copyright 1990, University of California
(C) copyright 1993, McGill University

For a list of the available commands: type "help"
(superdlx) load tests/fdata.s tests/f.s
Heap (for malloc) begins at 0x148
(superdlx) step 256
--> Clock Cycle 1
  stopped at pc = loop+0x8: sd a(r1),f4
(superdlx) next 5c
--> Clock Cycle 6
  stopped at pc = 0xfff0: sd 0xfff0(r1),f12
```

At the end of the sixth simulated cycle, the print commands is used to display various elements of the machine. To understand the notations used for the output, please refer to appendix A (an on-line help is also available, which provides the same information)

```
(superdlx) print
 ---> Clock Cycle 6
 ** Instruction Fetch Queue
Number of entries: 4
Maximum number of entries: 20
 -------------------------------------------------------------
| #      icount  address    opcode    rs1  rs2   rd     extra |
|-------------------------------------------------------------|
| 4          24  loop+0x1c   MULTD     10    2   12         0 |
| 3          23  loop+0x18   LD         1   10   10       -16 |
| 2          22  loop+0x14   SD         1    8    8        -8 |
| 1          21  loop+0x10   MULTD      6    2    8         0 |
+-------------------------------------------------------------+
```

** Integer Instruction Window
Number of entries: 1
Maximum number of entries: 20

#	opcode	address	rb_entry	operand1	ok1	typ1	operand2	ok2	typ2	pred
1	BNEZ	loop+0x34	1	0	0	INT	-56	1	IMM	PT

** Floating Point Instruction Window
Number of entries: 6
Maximum number of entries: 20

#	opcode	address	rb_entry	operand1	ok1	typ1	operand2	ok2	typ2	pred
6	LD	loop+0xc	17	0	0	INT	-8	1	IMM	-
5	SD	loop+0x8	16	0	0	INT	0	1	IMM	-
4	MULTD	loop+0x4	15	14	0	FPD	3.142	1	FPD	-
3	LD	loop	14	0	0	INT	0	1	IMM	-
2	MULTD	loop+0x28	11	10	0	FPD	3.142	1	FPD	-
1	MULTD	loop+0x1c	8	7	0	FPD	3.142	1	FPD	-

** Integer Reorder Buffer
Number of entries: 2
Maximum number of entries: 20
Head: 0 Tail: 1

#	opcode	icount	reg#	unit	result	ok	ready	flsh
0	SUBI	15	1	INT_ALU	192	0	7	0
1	BNEZ	16	0	INT_BRANCH	-	0	0	0

** Floating Point Reorder Buffer
Number of entries: 17
Maximum number of entries: 20
Head: 1 Tail: 17

#	opcode	icount	reg#	unit	result	ok	ready	flsh
1	LD	3	0	FP_MEM	28.000	1	6	0
2	MULTD	4	4	FP_MULT	87.965	0	11	0
3	SD	5	0	FP_MEM	-	1	5	0
4	LD	6	6	FP_MEM	27.000	1	6	0
5	MULTD	7	8	FP_MULT	84.823	0	11	0
6	SD	8	0	FP_MEM	-	1	5	0
7	LD	9	10	FP_MEM	26.000	0	7	0
8	MULTD	10	12	FP_MULT	-	0	0	0
9	SD	11	0	FP_MEM	-	1	6	0
10	LD	12	14	FP_MEM	25.000	0	7	0
11	MULTD	13	16	FP_MULT	-	0	0	0
12	SD	14	0	FP_DATA	-	0	7	0
13	BNEZ	16	0	NO_UNIT	-	0	0	0
14	LD	17	0	FP_DATA	-	0	0	0
15	MULTD	18	4	FP_MULT	-	0	0	0
16	SD	19	0	FP_DATA	-	0	0	0
17	LD	20	6	FP_DATA	-	0	0	0

** Load Buffer
Number of entries: 2
Maximum number of entries: 5

#	opcode	icount	address	rel	flsh
2	LD	20	0	0	0
1	LD	17	0	0	0

** Store Buffer
Number of entries: 5

```
Maximum number of entries: 5
 ------------------------------------------------------------------
|  #   opcode    icount   address  data to store  type ok rel  flsh |
|------------------------------------------------------------------|
|   5     SD         19        0              15    FPD  0   0     0 |
|   4     SD         14        c8             11    FPD  0   0     0 |
|   3     SD         11        d0              8    FPD  0   0     0 |
|   2     SD          8        d8              5    FPD  0   0     0 |
|   1     SD          5        e0              2    FPD  0   0     0 |
+------------------------------------------------------------------+

 ** Integer Functional Units
 --------------------------------------------
|   Unit         Number    Latency     Used   |
|--------------------------------------------|
| INT_ALU          4          1          1    |
| INT_SHIFT        2          1          0    |
| INT_DATA         4          1          0    |
| INT_COMP         1          1          0    |
| INT_BRANCH       1          1          0    |
+--------------------------------------------+

 ** Floating Point Functional Units
 --------------------------------------------
|   Unit         Number    Latency     Used   |
|--------------------------------------------|
| FP_ADD           4          2          0    |
| FP_DIV           2         10          0    |
| FP_MULT          4          5          2    |
| FP_CONV          2          2          0    |
| FP_DATA          4          1          1    |
| FP_COMP          1          1          0    |
| FP_BRANCH        1          1          0    |
+--------------------------------------------+
```

Let us examine the **reorder buffers** to find out information about the instructions that are currently in execution: in each of those instruction entries, the *ready* field is greater than the current clock cycle and the *validity* field (*ok* column) is equal to 0. Each of the **results** has already been calculated by the simulator, and its associated *ready* field indicates when the **result** is available, taking into account the latency of the operation. When the machine cycle count reaches the ready field value sometimes in the "future", the result will be validated. For example, the *multd* operation, at entry # 2 in the floating point reorder buffer will complete at clock cycle 11. Then its *validity* bit will be set to 1.

The unit tables give the current usage of the functional units: one *alu* is used (for the *subi*), two *floating point multiply* units (the two *multd* in execution), and one store address is being calculated in the address unit of the floating point operational unit.

Let us examine the **instruction windows** for information about the non yet issued instructions.

- **In the integer operational unit**
  There is only one entry in the integer instruction window, allocated to the first branch operation of the program. One operand is not ready: it refers to an entry in the integer reorder buffer (entry #0), where its value is going to be computed, by the **subi** operation. As we can infer from entry #0 of the integer reorder buffer, the **subi** result will be ready at the next clock cycle: the branch operation will then be issued. The branch is predicted *TAKEN* (PT) and the instructions that have been fetched in the fetch queue are from the predicted path. Since the result of the **subi** is going to be non zero, the outcome of the branch will correspond to its prediction and no instruction will have to be flushed. The prediction will then remain *take n* for the next time the branch is encountered: from the f.s program, we can see that the branch will always be taken, except once, at the end of the program. The prediction will thus always be correct except that one time. This little loop is a typical example for which the two bit branch prediction scheme is quite appropriate. But unfortunately, it is not always the case.

- **In the floating point operational unit**
  None of the instructions in the floating point window are ready to issue. They all wait for one of their operands to be computed. But we can try to deduct their respective issuing times from looking at both the instruction window and the reorder buffer.

  - entry #1, at the bottom of the instruction window: the **multd** operation needs one operand that will be forwarded from entry #7 of the floating point reorder buffer. Entry #7 is a **ld** entry that will return from memory at clock cycle 7: then the **multd** operation will issue. There will be no problem of structural hazards since only two multiply functional units are in use.

- entry #2: similarly, this **multd** needs an operand that is computed in reorder entry #10, and will be ready to issue next cycle.
- entries #3 #5 #6 need their first operands (register r1) from the integer reorder buffer, entry #0, which will be ready next cycle.
- entry #4: the **multd** depends on the result of the **ld** of entry #14 in the floating point reorder buffer: but this **ld** has not been issued yet (it is still in the instruction window, entry #3). The **ld** will need two cycles to complete, one for address calculation, one for memory access (without structural hazards), so the **multd** will not issue before clock cycle 9. But then, at clock cycle 9, there will be a functional unit conflict (four multiply units used). So it will wait till clock cycle 11, when two multiply units are freed.

Finally, let's now look at the **load and store buffers**. Loads and stores are particular cases of instructions as their process requires two phases: an address calculation phase (in the INT_DATA or FP_DATA functional units), and a memory access phase (their functional unit field is then set to INT_MEM or FP_MEM). Loads that are in their memory phase are not in the load buffer any more because they have been released to the data-cache: some have already got a value from memory, and all of them have bypassed older stores from the store buffer. All the stores that are present in the reorder buffer are still in the store buffer also, even though they may be in their memory access phase: this is because stores can only be released to the data-cache when they arrive at the head of the reorder buffer.

The following *print* output now illustrates the effect of the last branch prediction of the simulated program. When the **bnez** is issued this time, the effective calculated outcome reveals a wrong prediction: as a result, all instructions that were fetched after the branch have their entries flushed in both reorder buffers, and in the store/load buffers (**flsh** bit set to one).

```
(superdlx) next 40c
--> Clock Cycle 47
  stopped at pc = loop+0x38: nop

(superdlx)print reorder
---> Clock Cycle 47

 ** Integer Reorder Buffer
Number of entries: 2
Maximum number of entries: 20
```

```
Head: 6  Tail: 7
 ---------------------------------------------------------------------------
| #  opcode     icount  reg#    unit        result    ok       ready  flsh |
|--------------------------------------------------------------------------|
| 6  SUBI           99     1  INT_ALU            0     1          47     0 |
| 7  BNEZ          100     0  INT_BRANCH     -         0          48     0 |
+--------------------------------------------------------------------------+

 ** Floating Point Reorder Buffer
Number of entries: 17
Maximum number of entries: 20
Head: 0 Tail: 16
 ---------------------------------------------------------------------------
| #  opcode     icount  reg#    unit        result    ok       ready  flsh |
|--------------------------------------------------------------------------|
| 0  MULTD          88     4  FP_MULT       12.566     0          49     0 |
| 1  SD             89     0  FP_MEM         -         1          43     0 |
| 2  LD             90     6  FP_MEM         3.000     1          44     0 |
| 3  MULTD          91     8  FP_MULT        9.425     0          49     0 |
| 4  SD             92     0  FP_MEM         -         1          44     0 |
| 5  LD             93    10  FP_MEM         2.000     1          45     0 |
| 6  MULTD          94    12  FP_MULT        6.283     0          50     0 |
| 7  SD             95     0  FP_MEM         -         1          44     0 |
| 8  LD             96    14  FP_MEM         1.000     1          45     0 |
| 9  MULTD          97    16  FP_MULT        3.142     0          50     0 |
| 10 SD             98     0  FP_MEM         -         1          47     0 |
| 11 BNEZ          100     0  NO_UNIT        -         0           0     0 |
| 12 LD            101     0  FP_DATA        -         0           0     1 |
| 13 MULTD         102     4  FP_MULT        -         0           0     1 |
| 14 SD            103     0  FP_DATA        -         0           0     1 |
| 15 LD            104     6  FP_DATA        -         0           0     1 |
| 16 MULTD         105     8  FP_MULT        -         0           0     1 |
+--------------------------------------------------------------------------+

For information on the notation used: "print help" or "print <buffername> help"
```

```
(superdlx) print data

---> Clock Cycle 47

 ** Load Buffer
Number of entries: 2
Maximum number of entries: 5
 --------------------------------------------
|   #  opcode    icount   address  rel  flsh |
|--------------------------------------------|
|   2     LD        104        0     0     1 |
|   1     LD        101        0     0     1 |
+--------------------------------------------+


 ** Store Buffer
Number of entries: 5
Maximum number of entries: 5
 -------------------------------------------------------------------
|  #   opcode    icount   address  data to store  type ok rel  flsh |
|-------------------------------------------------------------------|
|   5     SD        103        0              13   FPD  0   0     1 |
|   4     SD         98        8               9   FPD  0   0     0 |
|   3     SD         95       10               6   FPD  0   0     0 |
|   2     SD         92       18               3   FPD  0   0     0 |
|   1     SD         89       20               0   FPD  0   0     0 |
+-------------------------------------------------------------------+

For information on the notation used: "print help" or "print <buffername> help"
(superdlx)
```

F Prototype of a SuperDLX X Window Interface

A prototype for a X Window interface has been developped as a course project (cs308-573A). The X Window environment can be used to set up the machine configuration parameters and to study the simulation at run time. It consists of command panels and display panels. Each panel is provided with a help button, permitting the user to pop up a window displaying the content of an help file. Some panels are shown here as examples. They have been printed using the OpenLook Snapshot program.

1. **The Front Menu**

   This is the first window appearing when the simulator is run. It consists of a label widget and four command widgets (figure 9):

   - **START**: to start up the simulation by popping up the control panel;
   - **EXIT**: to exit the simulator: before exiting a confirmation is asked by the exit window;
   - **MACHINE**: to makes the machine configuration panel appear and set up the parameters of the machine.
   - **HELP**: to display the help window;

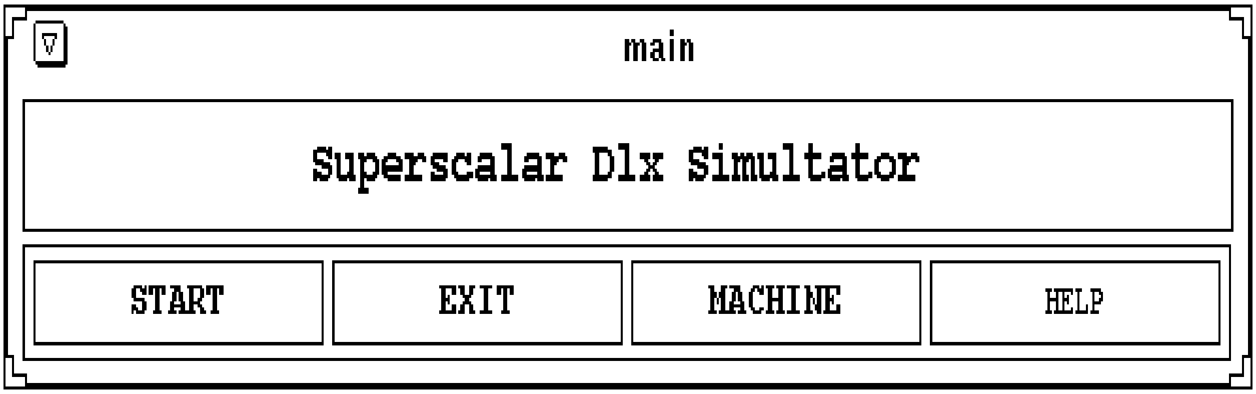

Figure 9: X-Interface Front Panel

2. **The Control Panel**

   Two kinds of commands are available:

   - the **simulation control commands** to run the simulator;

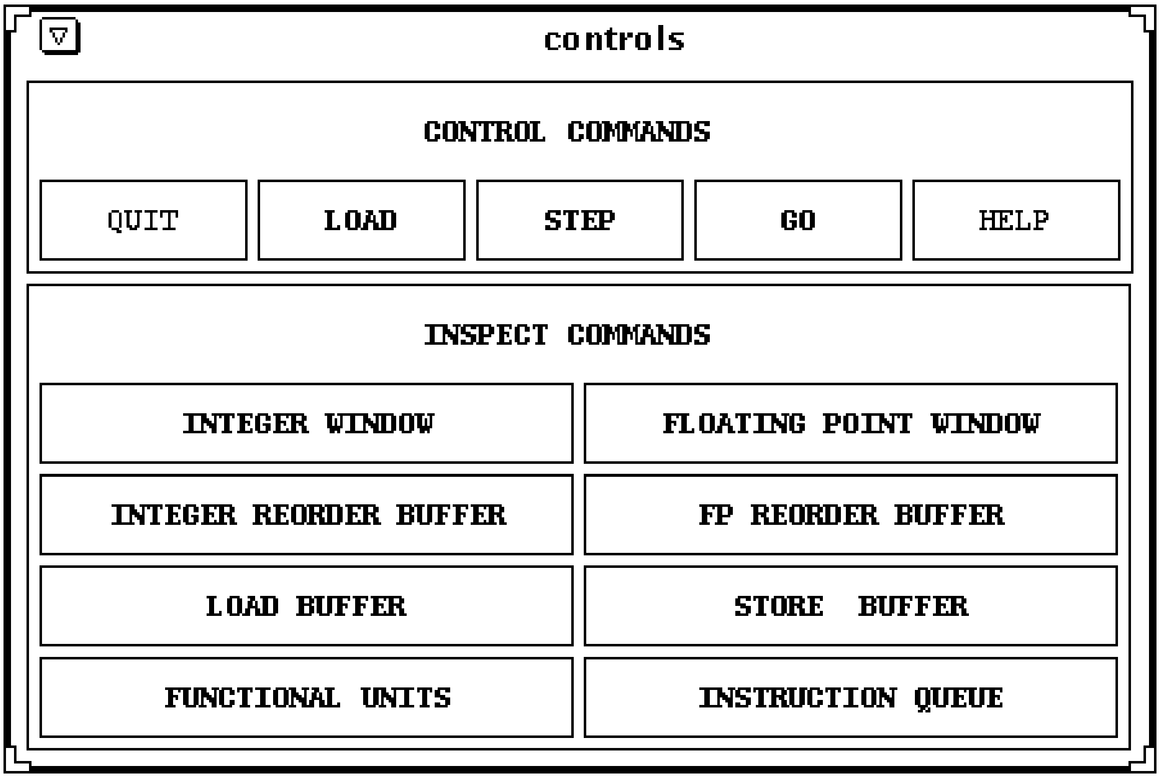

Figure 10: Control Menu

- the **inspect commands** to look at the contents of different machine data structures during simulation.

3. **Example of simulation command panel**
   The step and load command panels are shown in figure 11 and figure 12.

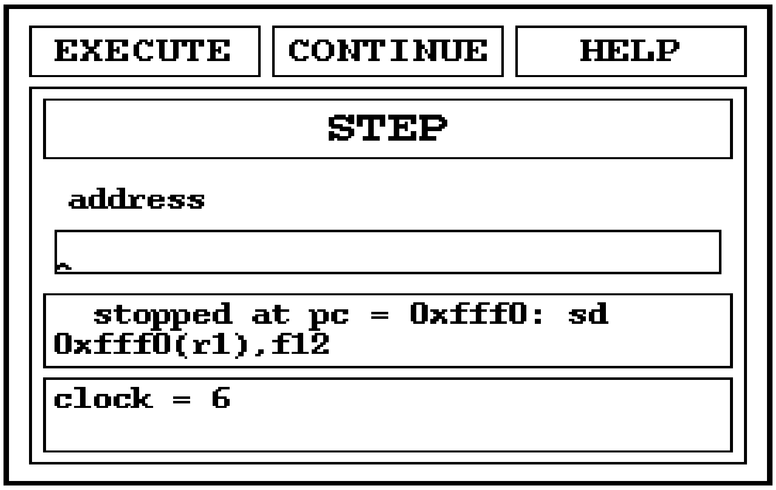

Figure 11: Step Command Panel

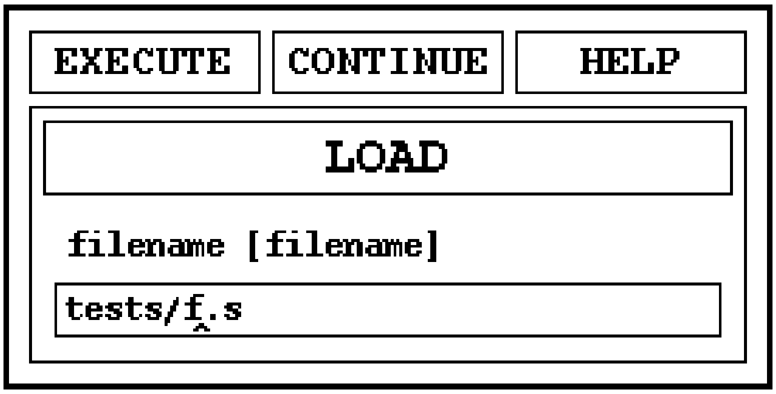

Figure 12: Load Command Panel

4. **Examples of inspect panel** The floating point instruction window and the floating point reorder buffer are shown in figure 13 and figure 14

5. **The Machine configuration panel**: figure 15

fpWindowShell

CONTINUE | UPDATE | HELP

Floating Point Instruction Window

#	opcode	address	rb_entry	operand1	ok	typ	operand2	ok	typ	pred
7	LD	loop+0xc	17	0	0	INT	-8	1	IMM	-
6	SD	loop+0x8	16	0	0	INT	0	1	IMM	-
5	MULTD	loop+0x4	15	14	0	FPD	3.142	1	FPD	-
4	LD	loop	14	0	0	INT	0	1	IMM	-
3	MULTD	loop+0x28	11	10	0	FPD	3.142	1	FPD	-
2	MULTD	loop+0x1c	8	7	0	FPD	3.142	1	FPD	-
1	MULTD	loop+0x10	5	4	0	FPD	3.142	1	FPD	-

Number of Elements: 7

Figure 13: Floating Point Instruction Window Display

fpBufferShell

CONTINUE | UPDATE | HELP

Floating Point Reorder Buffer

#	opcode	icount	reg#	unit	result	ok	ready	flsh
1	LD	3	0	FP_MEM	28.000	1	6	0
2	MULTD	4	4	FP_MULT	87.965	0	11	0
3	SD	5	0	FP_MEM	-	1	5	0
4	LD	6	6	FP_MEM	27.000	0	7	0
5	MULTD	7	8	FP_MULT	-	0	0	0
6	SD	8	0	FP_MEM	-	1	5	0
7	LD	9	10	FP_MEM	-	0	0	0
8	MULTD	10	12	FP_MULT	-	0	0	0
9	SD	11	0	FP_MEM	-	1	6	0
10	LD	12	14	FP_MEM	-	0	0	0
11	MULTD	13	16	FP_MULT	-	0	0	0
12	SD	14	0	FP_DATA	-	0	7	0
13	BNEZ	16	0	NO_UNIT	-	0	0	0
14	LD	17	0	FP_DATA	-	0	0	0

Figure 14: Floating Point Reorder Buffer Display

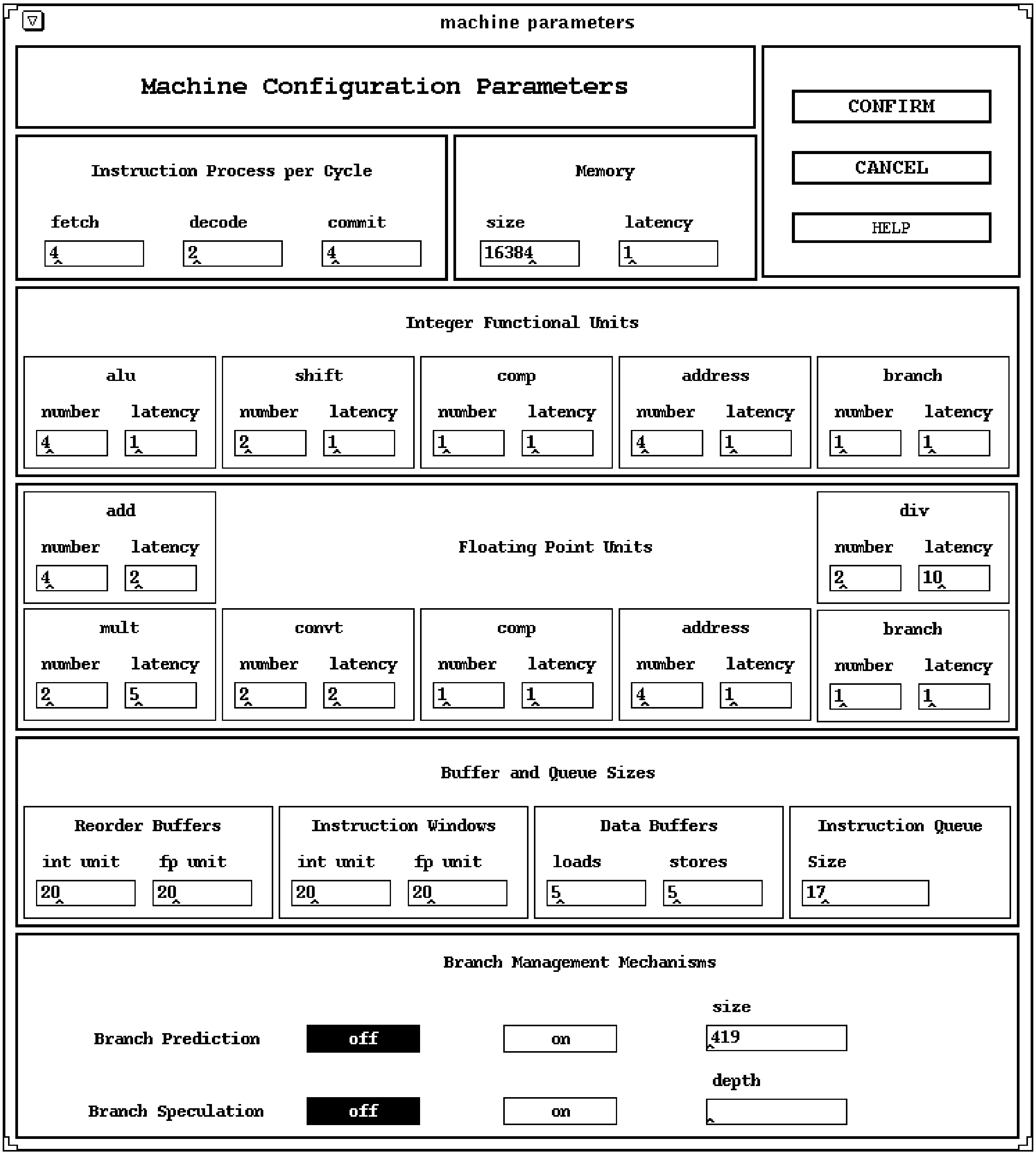

Figure 15: Machine Configuration Parameters

References

[AD92] M. Allan and K. Diefendorff. Organization of the motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[BD91] V.Popescu M.Shultz J.Spracklen B.Lightner and D.Isaman. The metaflow architecture. *IEEE Micro*, pages 10–73, June 1991.

[GKTM90] G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore. Branch and fixed-point instruction execution units. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 154–161. International Business Machines Corporation, 1990. Order No. SA23-2619.

[Gro90] G.F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Research and Development*, 34(1):1–136, January 1990.

[HB92] L.B Hostetler and B.Mirtich. DLXsim - A Simulator for DLX, Handout#4 Course 308-605B. January 1992.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[Kel75] R. M. Keller. Look-ahead processors. *Computing Surveys*, 7(4), 1975.

[Lai92] M. Laird. A comparison of the three current superscalar designs. *ACM SIGARCH NEWS*, 20(3):14–31, June 1992.

[LS84] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, January 1984.

[McG90] S. McGeady. Inside Intel's i960CA superscalar processor. *Microprocessors and Microsystems*, pages 385–396, July 1990.

[PGH+87] A. R. Pleszkun, A. J. Goodman, W. C. Hsu, R. Joersz, G. Bier, P. Woest, and P. Schecter. WISQ: A restartable architecture using queues. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 15(2), June 1987.

[SLH90] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 18(2), June 1990.

[SP85] J.E. Smith and A.R. Pleszkun. Implementation of precise interrupts in pipelined processors. *Proceedings of the 12th Annual International Synposium on Computer Architecture*, pages 36–44, June 1985.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[WS84] S. Weiss and J.E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, C-33:1013–1022, November 1984.