Duper scalar Processor:
The hardware approach to instruction level parallelism

A Thesis

Submitted to the Department of Computer Science and Engineering

of

BRAC University

by

Intekhab Sadekin

Student ID: 05341002

In Partial Fulfillment of the

Requirements for the Degree

of

Bachelor of Science in Computer Science

August 2007

DECLARATION

I hereby declare that this thesis is based on whatever that I thought of and is very much a unique idea and not copied or plagiarized from anyone or any paper. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Signature of
Supervisor

Signature of
Author

ACKNOWLEDGMENTS

Special thanks to Risat Mahmud Pathan, senior lecturer of BRAC University, who has taught me the basics of computer architecture and helped me with the diagrammatic representation of the proposed architecture. He has, moreover, helped me with writing the algorithm that I have proposed to complement the proposed architecture. I would also like to thank Dr. Mumit Khan, associate professor of BRAC University and director of CRBLP who has given me advice on various issued regarding this thesis. Dr. Mark Hill of the University of Wisconsin, Madison, has also helped me in providing the simulator that I was supposed to use for the purpose of this thesis. Last but by no means the least I would like to thank Dr. Per Stenstrom of Chalmers University who has provided the initial motivation for the thesis.

ABSTRACT

Multi-core processors are being widely used nowadays and the numbers of cores are increasing in the commercial arena with great speed with the gigahertz race between the two stalwarts, Intel and AMD. Usually the cores are symmetric, which means that all the cores are functionally identical.

This paper proposes an architecture that brings in a new dimension to instruction level parallelism. The operating system in today's machines does all the decision making as to how the instructions in a task can be parallelized by deciding which task gets assigned to which core. The hardware support for exploiting instruction level parallelism is very small and has very little decision making power. Most recently dynamic scheduling of the instructions paved the pathway for major hardware changes and hence the decision making power shared. But the problem still persists. The operating has no direct help from the hardware and has to do most of the work at software level and hence the operating system has to be modified as the number of cores increase and the type of cores change. So a hardware support is a necessity in order to keep the operating system unchanged so that it doesn't have to worry about the cores.

This hardware support greatly simplifies the design of the OS, which is trying to make the maximum benefits of multi-core processors without needing changes as the number of cores change.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Introduction:

But this sort of arrangement also means that we are unable to take the benefits of specialized cores- cores that are designed to execute certain types of instruction in the least possible clock cycles. They will be able to execute the other types of instructions as well, but maybe not in the least possible clock cycles.

In addition when dealing with asymmetric cores, the OS must be specifically designed with the multiple cores in mind. We plan to release the OS designer from this burden by introducing a **Hardware Virtual Processor** that will appear as a single processor to the OS, but inside it will be multiple specialized cores executing instructions in parallel. It might sound like Superscalar processors at first. But the difference of our design is that each core is capable of executing all types of instructions. But for some types they will be specialized and will be able to execute those instructions with maximum speed (minimum clock cycles).

Now let us try to see what problem arises if we try to make the OS aware of the presence of multiple specialized cores .The OS has more to do than just assigning cores to tasks and maintaining a task queue. It has to keep track of how many cores (and of what specialization) are there so that it can schedule the tasks optimally to all the cores. This will require it to have the knowledge of the instruction distribution of the task at hand. Because if a task that has 80% arithmetic operations are assigned to a core that is specialized for Load/Store operations, performance will suffer. It also has to balance load across cores, i.e. assign tasks to each core in such a way that no single core is idle or no single core is being over-burdened. This ultimately means that with the change of number of specialized cores, the OS has to be changed.

We are suggesting a hardware support for both the OS and the cores so that the OS does not have to know how many cores are there and hence the OS can

treat the multiprocessors as a single core ordinary processor. The following diagram will clarify the statement just made. We call this hardware support a Virtual Processor (VP). With this VP the OS will no longer have to worry about the number of cores and will therefore treat this VP as the only single core on the motherboard and pass the instructions to this VP and not the cores directly. This will also free the OS from the burden of maintaining multiple task queues and load balancing. So any type of OS can reap the benefits of multi-core processors even if it is not designed to do that.

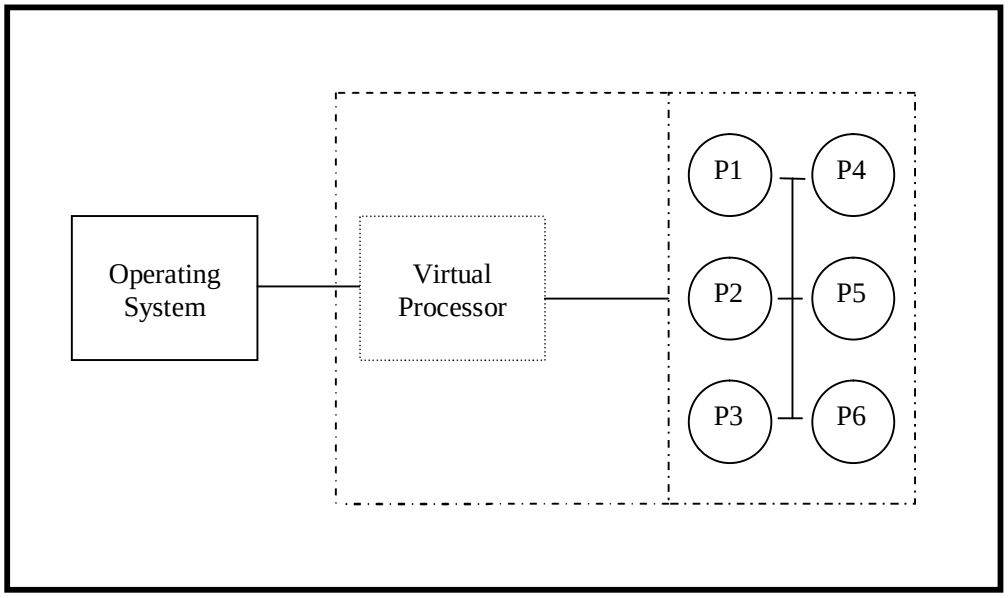

Fig-1 The overall structure

CHAPTER I

1. Operation of VP

The VP will encapsulate the operations of the OS and the cores from each other and act as an interface for both.

What the OS will do is de-queue a task from the task queue that it maintains and forward it to the VP. The VP, from this point on, will try to parallelize the instruction and not the task. It will decide which particular instruction should be assigned to which core i.e. it will act as an instruction dispatcher rather than a task dispatcher like the OS.

Among the information that the VP designer needs is determining what are the percentages of instructions that is, what percentage of instructions are arithmetic instructions, logical instructions, jump instructions, branch instructions and so on. Once the VP designer has this information, he can how decide many cores of which type are required. Which means core 1 could be specialized to perform arithmetic operations and core 2 could be specialized to perform logical operations and core 3 could be specialized in performing memory operations and so on.

Each of the cores will be designed in such a way that it will be able to perform a specific type of instructions optimally. For example core 1 can be specialized in performing arithmetic operations, i.e. this core will take lesser cycles to perform arithmetic operations than the other cores but the other cores will be able to perform the same operation but not quite as efficiently as core1. Similarly core 2 can be specialized in performing branch instructions, core 3 logical instructions, and core 4 floating-point instructions and so on. There are no limitations as to a

single core performing specialized operations, which means that there may be multiple cores specialized for the same type of instruction.

The next issue is: how many cores and how many of what type? Here we have to consider the system in question. If the system is designed to execute tasks that have an average estimate of 50% Arithmetic/Logic instructions, 25% Load/Store and 25% Branch instructions, then we have 2 specialized cores for Arithmetic/Logic, 1 for Load/Store and 1 for Branch. One thing should be always kept in mind and that is, all the cores are capable of executing all types of instructions but each one is specialized for one particular category.

CHAPTER II

2. The Objective

The main aim of this thesis will be to devise an algorithm that will optimally dispatch instruction to the cores, i.e. it will dispatch instructions so that they are executed in the least possible clock cycles and make the most efficient use of the cores. Issues like inter-core dependency, hazards, re-ordering instructions, etc. will also have to be taken into consideration. Or in other words we can say that the main aim is to parallelize instructions given by the Operating System efficiently to all the cores; hence an algorithm has to be devised.

CHAPTER III

3. Instruction Level Parallelism-concept and challenges

All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP) since the instructions can be evaluated in parallel. In this section we look at a wide range of techniques for extending the pipelining ideas by increasing the amount of parallelism exploited among instructions. We start this section by looking at the limitation imposed by data and control hazards and then slowly turn to the topic of increasing the ability to exploit parallelism.

There are two largely separable approaches to exploiting ILP. This section covers techniques that are largely dynamic and depend on the hardware to locate the parallelism. The next section focuses on techniques that are static and rely much more on software. In practice, this partitioning between dynamic and static and between hardware-intensive and software-intensive is not clean, and techniques from one camp are often used by the other. Nonetheless, for exposition purposes, we have separated the two approaches and tried to indicate where an approach is transferable.

The dynamic, hardware intensive approaches dominate the desktop and server markets and are used in a wide range of processors, including: the Pentium III and 4, the Althon, the MIPS R10000/12000, the Sun ultraSPARC III, the Power-PC 603, G3, and G4, and the Alpha 21264. The static, compiler-intensive approaches, which we focus on in the next section, have seen broader adoption in the embedded market than the desktop or server markets, although the new IA-64 architecture and Intel’s Itanium use this more static approach. In this section, we discuss features of both programs and processors that limit the amount of parallelism that can be exploited among instructions, as well as the

critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances. Recall that the value of the CPI (Cycles per Instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

The ideal pipeline CPI is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI and thus increase the IPC (Instructions per Clock). In this section we will see that the techniques we introduce to increase the ideal IPC, can increase the importance of dealing with structural, data hazard, and control stalls. The equation above allows us to characterize the various techniques we examine in this section by what component of the overall CPI a technique reduces.

Before we examine these techniques in detail, we need to define the concepts on which these techniques are built. These concepts, in the end, determine the limits on how much parallelism can be exploited.

3.1 Instruction-Level Parallelism

All the techniques in this section and the next exploit parallelism among instructions.

As we stated above, this type of parallelism is called instruction-level parallelism or ILP. The amount of parallelism available within a basic block–a straight- line code sequence with no branches in except to the entry and no branches out except at the exit–is quite small. For typical MIPS programs the average dynamic

branch frequency often between 15% and 25%, meaning that between four and seven instructions execute between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks. The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap. There are a number of techniques we will examine for converting such loop level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (an approach we explore in the next section) or dynamically by the hardware (the subject of this section).

An important alternative method for exploiting loop-level parallelism is the use of vector instructions. Essentially, a vector instruction operates on a sequence of data items. For example, the above code sequence could execute in four instructions on some vector processors [1]: two instructions to load the vectors x and y from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped. Although the development of the vector ideas preceded many of the techniques we examine in these two sections for exploiting ILP, processors that exploit ILP have almost completely replaced vector-based processors. Vector instruction sets,

however, may see a renaissance, at least for use in graphics, digital signal processing, and multimedia applications.

3.2 Data Dependence and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism we must determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist). If two instructions are dependent they are not parallel and must be executed in order, though they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

3.2.1 Data Dependences

There are three different types of dependences: data dependences (also called true data dependences) [1], name dependences, and control dependences. An instruction j is data dependent on instruction i if either of the following holds:

-Instruction i produces a result that may be used by instruction j, or
-Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program. For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
Loop: L.D F0, 0(R1); F0=array element
      ADD.D F4, F0, F2; add scalar in F2
      S.D F4, 0(R1); store result
      DADDUI R1, R1, #-8; decrement pointer 8 bytes (/e
      BNE R1, R2, LOOP; branch R1! =zero
```

The data dependences in this code sequence involve both floating point data:

```
Loop: L.D F0, 0(R1); F0=array element
      ADD.D F4, F0, F2; add scalar in F2
      S.D F4, 0(R1); store result
```

and integer data:

```
DADDIU R1, R1,-8; decrement pointer
                ; 8 bytes (per DW)
BNE       R1, R2, Loop; branch R1! =zero
```

Both of the above dependent sequences, as shown by the arrows, with each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap, since the program will not execute correctly. The presence of data

dependence in an instruction sequence reflects data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of programs. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the pipeline organization. This difference is critical to understanding how instruction-level parallelism can be exploited. In our example, there is data dependence between the DADDIU and the BNE; this dependence causes a stall because we moved the branch test for the MIPS pipeline to the ID stage. Had the branch test stayed in EX, this dependence would not cause a stall. Of course, the branch delay would then still be 2 cycles, rather than 1.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. The importance of the data dependences is that dependence (1) indicates the possibility of a hazard, (2) determines the order in which results must be calculated, and (3) sets an upper bound on how much parallelism can possibly be exploited [1]. Since data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this section and the next is overcoming these limitations. Dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering dependence. In this section, we consider hardware schemes for scheduling code dynamically as it is executed. As we will see, some types of dependences can be eliminated, primarily by software, and in some cases by hardware techniques. A data value may flow between instructions either through registers or through memory locations. When the data flow occurs in a register, detecting the dependence is reasonably straightforward since the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns

cause a compiler or hardware to be conservative. Dependences that flow through memory locations are more difficult to detect since two addresses may refer to the same location, but look different: For example, 100(R4) and 20(R6) may be identical. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that 20(R4) and 20(R4) will be different), further complicating the detection of a dependence. In this section, we examine hardware for detecting data dependences that involve memory locations, but we shall see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism, as we shall see in the next section.

3.2.2 Name Dependences

The second type of dependence is name dependence [1]. Name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction i that precede instruction j in program order:

1. An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i read the correct value.

2. An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Both antidependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be

reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict. This renaming can be more easily done for register operands, where it is called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

3.3 Data Hazards

A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called program order that is the order that the instructions would execute in, if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order only where it affects the outcome of the program. Detecting and avoiding hazards ensures that necessary program order is reserved. Data hazards [1] may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i occurring before j in program order. The possible data hazards are:

3.3.1 RAW (read after write) —j tries to read a source before i write it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to true data dependence. Program order must be preserved to ensure that j receives the value from i. In the simple common five-stage static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

3.3.2 WAW (write after write) — j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The classic five-stage integer pipeline writes a register only in the WB stage and avoids this class of hazards, but this section explores pipelines that allow instructions to be reordered, creating the possibility of WAW hazards. WAW hazards can also between a short integer pipeline and a longer floating-point pipeline. For example, a floating point multiply instruction that writes F4, shortly followed by a load of F4 could yield a WAW hazard, since the load could complete before the multiply completed.

3.3.3 WAR (write after read) — j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard arises from antidependence. WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) [2] and all writes are late (in WB) [2]. A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline or when instructions are reordered, as we will see in this section.

Note that the RAR (read after read) case is not a hazard.

3.4 Control Dependences

The last type of dependence is control dependence [1]. Control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these

control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the “then” part of an ‘if’ statement on the branch. For example, in the code segment:

```
If p1
{
        S1;
};
If p2
{
        S2;
}
```

S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the if-statement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

Control dependence is preserved by two properties in a simple pipeline. First, instructions execute in program order. This ordering ensures that an instruction that occurs before a branch is executed before the branch. Second, the detection of control or branch hazards ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known. Although preserving control dependence is a useful and simple way to help preserve

program order, the control dependence in itself is not the fundamental performance limit. We may be willing to execute instructions that should not have been executed, thereby violating the control dependences, if we can do so without affecting the correctness of the program. Control dependence is not the critical property that must be preserved. Instead, the two properties critical to program correctness–and normally preserved by maintaining both data and control dependence–are the exception behavior and the data flow. Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program. A simple example shows how maintaining the control and data dependences can prevent such situations. Consider this code sequence:

```
        DADDU R2, R3, R4
        BEQZ R2, L1
        LW R1, 0 (R2)
L1:
```

In this case, it is easy to see that if we do not maintain the data dependence involving R2, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that no data dependence prevents us from interchanging the BEQZ and the LW; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we would like to just ignore the exception when the branch is taken. The second property preserved by maintenance of data dependences and control dependences is the data flow. The data flow is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points. Put another way, it is not sufficient to just maintain data dependences because an instruction may

be data dependent on more than one predecessor. Program order is what determines which predecessor will actually deliver a data value to an instruction. Program order is ensured by maintaining the control dependences.

For example, consider the following code fragment:

```
        DADDU R1, R2, R3
        BEQZ R4, L
        DSUBU R1, R5, R6
L: ...

        OR R7, R1, R8
```

In this example, the value of R1 used by the OR instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The OR instruction is data dependent on both the DAAU and DSUBU instructions, but preserving this order alone is insufficient for correct execution. Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken then the value of R1 computed by the DSUBU should be used by the OR, and if the branch is taken the value of R1 computed by the DADDU should be used by the OR. By preserving the control dependence of the OR on the branch, we prevent an illegal change to the data flow. For similar reasons, the DSUBU instruction cannot be moved above the branch. Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow, as we will see in section 3.5. Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow.

Consider the following code sequence:

```
            DADDU    R1, R2, R3
            BEQZ     R12, skipnext
            DSUBU    R4, R5, R6
            DADDU    R5, R4, R9
skipnext:   OR       R7, R8, R9
```

Suppose we knew that the register destination of the DSUBU instruction (R4) was unused after the instruction labeled skipnext. (The property of whether a value will be used by an upcoming instruction is called liveness.) If R4 were unused, then changing the value of R4 just before the branch would not affect the data flow since R4 would be dead (rather than live) in the code region after skipnext. Thus, if R4 were dead and the existing DSUBU instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the DSUBU instruction before the branch, since the data flow cannot be affected by this change. If the branch is taken, the DSUBU instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is sometimes called speculation, since the compiler is betting on the branch outcome; in this case, the bet is that the branch is usually not taken. Control dependence is preserved by implementing control hazard detection that causes control stalls. Control stalls can be eliminated or reduced by a variety of hardware and software techniques. Delayed branches [1], can reduce the stalls arising from control hazards; scheduling a delayed branch requires that the compiler preserve the data flow. The key focus of the rest of this section is on techniques that exploit instruction level parallelism using hardware. The data dependences in a compiled program act as a limit on how much ILP can be exploited. The challenge is to approach that limit by trying to minimize the actual hazards and associated stalls that arise. The techniques we examine become ever more sophisticated in an attempt to exploit all the available parallelism while maintaining the necessary true data dependences in the code.

CHAPTER IV

4. Overcoming Data Hazards with Dynamic Scheduling

A simple satirically scheduled pipeline fetches an instruction and issues it, unless there was data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. (Forwarding logic reduces the effective pipeline latency so that the certain dependences do not result in hazards). If there is a data dependence that cannot be hidden, then the hazard detection hardware stalls the pipeline (starting with the instruction that uses the result). No new instructions are fetched or issued until the dependence is cleared. In this section, we explore an important technique, called dynamic scheduling, in which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior. Dynamic scheduling offers several advantages: It enables handling some cases when dependences are unknown at compile time (e.g., because they may involve a memory reference), and it simplifies the compiler. Perhaps most importantly, it also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. Later, we will explore hardware speculation, a technique with significant performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at a cost of a significant increase in hardware complexity. Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences, which could generate hazards, are
present. In contrast, static pipeline scheduling by the compiler (covered in the next section) tries to minimize stalls by separating dependent instructions so that they will not lead to hazards. Of course, compiler pipeline scheduling can also be used on code destined to run on a processor with a dynamically scheduled pipeline.

4.1 Dynamic Scheduling: The Idea [1]

A major limitation of the simple pipelining techniques is that they all use in-order instruction issue and execution: Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle. If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute. For example, consider this code:

```
DIV.D  F0, F2, F4
ADD.D F10, F0, F8
SUB.D  F12, F8, F14
```

The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet SUB.D is not data dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order. In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved. To allow us to begin executing the SUB.D in the above example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in-order instruction issue (i.e., instructions issue in program order), but we want an instruction to begin execution as soon as its data operand is available. Thus, this pipeline does out-of-order execution, which implies out-of-order completion. Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline.

Consider the following MIPS floating-point code sequence:

```
DIV.D F0, F2, F4
ADD.D F6, F0, F8
SUB.D F8, F10, F14
MULT.D F6, F10, F8
```

There is an antidependence between the ADD.D and the SUB.D, and if the pipeline executes the SUB.D before the ADD.D (which is waiting for the DIV.D), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of F6 by MULT.D, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming. Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise. Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed; we will see shortly how this property can be guaranteed. Although exception behavior must be preserved, dynamically scheduled processors may generate imprecise exceptions. An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

1. The pipeline may have already completed instructions that are later in program order than the instruction causing the exception, and
2. The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.

Imprecise exceptions make it difficult to restart execution after an exception. To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

1. Issue—Decode instructions, check for structural hazards.
2. Read operands—Wait until no data hazards, then read operands.

An instruction fetch stage precedes the issue stage and may fetch either into an instruction register or into a queue of pending instructions; instructions are then issued from the register or queue. The EX stage follows the read operands stage, just as in the five-stage pipeline. Execution may take multiple cycles, depending on the operation. We will distinguish when an instruction begins execution and when it completes execution; between the two times, the instruction is in execution. Our pipeline allows multiple instructions to be in execution at the same time, and without this capability, a major advantage of dynamic scheduling is lost. Having multiple instructions in execution at once requires multiple functional units, pipelined functional units, or both. Since these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units. In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. Scoreboarding [1]; is a technique for allowing instructions to execute out-of-order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability. We focus on a more sophisticated technique, called Tomasulo's algorithm [1] that has several major enhancements over scoreboarding.

4.2 Dynamic Scheduling Using Tomasulo's Approach

In this approach we find a way to execute instructions out-of-order that is schedule the instructions dynamically and Tomasulo's algorithm proposes just that. The algorithm that we will be proposing is quite different from that of Tomasulo's algorithm but the following is a review of how dynamic scheduling works with the algorithm that Tomasulo proposed.

As we will see RAW hazards are avoided by executing an instruction only when its operands are available. WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. Register renaming eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand. To better understand how register renaming eliminates WAR and WAW hazards consider the following example code sequence that includes both a potential WAR and WAW hazard:

```
DIV.D F0, F2, F4
ADD.D F6, F0, F8
S.D F6, 0 (R1)
SUB.D F8, F10, F14
MULT.D F6, F10, F8
```

There is antidependence between the ADD.D and the SUB.D and an output dependence between the ADD.D and the MULT.D leading to three possible hazards: a WAR hazard on the use of F8 by ADD.D and on the use of F8 by the MULT.D, and a WAW hazard since the ADD.D may finish later than the MULT.D. There are also three true data dependences between the DIV.D and the ADD.D, between the SUB.D and the MULT.D, and between the ADD.D and the S.D. These name dependences can both be eliminated by register renaming. For

simplicity, assume the existence of two temporary registers, S and T. Using S and T, the sequence can be rewritten without any dependences as:

```
DIV.D F0, F2, F4
ADD.D S, F0, F8
S.D S, 0 (R1)
SUB.D T, F10, F14
MULT.D F6, F10, T
```

In addition, any subsequent uses of F8 must be replaced by the register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of F8 that are later in the code requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the above code segment and a later use of F8. As we will see Tomasulo's algorithm can handle renaming across branches. In Tomasulo's scheme, register renaming is provided by the reservation stations, which buffer the operands of instructions waiting to issue, and by the issue logic. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming. Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler. As we explore the components of Tomasulo's scheme, we will return to the topic of register renaming and see exactly how the renaming occurs and how it eliminates WAR and WAW hazards. The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: The information held in the reservation stations at each functional unit determine when an instruction can

begin execution at that unit. Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91 this is called the common data bus, or CDB). In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

There are three components of the Tomasulo's algorithm that is dramatically different from the five-stage pipelining that we know and they are:

1. Issue—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. This step renames registers, eliminating WAR and WAW hazards.

2. Execute—If one or more of the operands is not yet available, monitor the common data bus (CDB) while waiting for it to be computed. When an operand becomes available, it is placed into the corresponding reservation station. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available RAW, hazards are avoided. Notice that several instructions could become ready in the same clock cycle for the same functional unit. Although independent functional units could begin execution in the same clock cycle for different instructions, if more than one instruction is ready for a single functional unit, the unit will have to choose among them. For the floating point reservation stations, this choice may be made arbitrarily; loads and stores, however, present an additional complication.

3. Write result—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.

CHAPTER V

5. Basic Compiler Techniques for Exposing ILP [3]

This chapter starts by examining the use of compiler technology to improve the performance of pipelines and simple multiple-issue processors. These techniques are key even for processors that make dynamic issue decisions but use static scheduling and are crucial for processors that use static issue or static scheduling. After applying these concepts to reducing stalls from data hazards in single issue pipelines, we examine the use of compiler-based techniques for branch prediction. Armed with this more powerful compiler technology, we examine the design and performance of multiple-issue processors using static issuing or scheduling. Putting It All Together examines the IA-64 architecture [3] and its first implementation, Itanium. Two different static, VLIW-style processors [3] are covered in Another View.

5.1 Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Throughout this section we will assume the FP unit latencies [3], unless different latencies are explicitly stated. We assume the standard 5-stage integer pipeline, so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth), so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

In this subsection, we look at how the compiler can increase the amount of available ILP by unrolling loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described later in this chapter. We will rely on an example similar to the one we used in the last chapter, adding a scalar to a vector:

```
for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. We will formalize this notion later in this chapter and describe how we can test whether loop iterations are independent at compile-time. First, let's look at the performance of this loop, showing how we can use the parallelism to improve its performance for a MIPS pipeline with the latencies shown above. The first step is to translate the above segment to MIPS assembly language. In the following code segment, R1 is initially the address of the element in the array with the highest address, and F2 contains the scalar value, s. Register R2 is precomputed, so that 8(R2) is the last element to operate on. The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop: L.D F0, 0 (R1); F0=array element
      ADD.D F4, F0, F2; add scalar in F2
      S.D F4, 0 (R1); store result
      DADDUI R1, R1, #-8; decrement pointer
                         ; 8 bytes (per DW)
      BNE R1, R2, Loop; branch R1! = zero
```

Without any scheduling the loop will execute as follows:

	Clock cycle issued
Loop: L.D F0,0(R1)	1
stall	2
ADD.D F4,F0,F2	3
stall|	4
stall	5
S.D F4,0(R1)	6
DADDUI R1,R1,#-8	7
stall	8
BNE R1,R2,Loop	9
stall	10

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop: L.D F0,0(R1)
DADDUI R1,R1,#-8
ADD.D F4,F0,F2
stall
BNE R1,R2,Loop ;delayed branch
S.D F4,8(R1) ;altered & interchanged
             with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6. The stall after ADD.D is for the use by the S.D.

5.2 Static Multiple Issue: the VLIW Approach

Superscalar processors decide on the fly how many instructions to issue. A statically scheduled superscalar must check for any dependence between instructions in the issue packet as well as between any issue candidate and any

instruction already in the pipeline. A statically scheduled superscalar requires significant compiler assistance to achieve good performance. In contrast, a dynamically-scheduled superscalar requires less compiler assistance, but has significant hardware costs. An alternative to the superscalar approach is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences. The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when dependence may be present. Such an approach offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization. The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences used wide instructions with multiple operations per instruction. For this reason, this architectural approach was named VLIW, standing for Very Long Instruction Word, and denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more). The basic architectural concepts and compiler technology are the same whether multiple operations are organized into a single instruction, or whether a set of instructions in an issue packet is preconfigured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware. To reduce this inflexibility and enhance performance of the approach, several innovations have been incorporated into more recent architectures of this type, while still requiring the compiler to do most of the work of finding and scheduling instructions for parallel execution. This second generation of VLIW architectures is the approach being pursued for desktop and server markets. In the remainder of this section, we look at the basic concepts in a VLIW architecture.

5.2.1 The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. Since there is not fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded. Since this advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider-issue processor. Indeed, for simple two issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four issue processor has manageable overhead, but as we saw in the last chapter, this overhead grows with issue width. Because VLIW approaches make sense for wider processors, we choose to focus our example on such an architecture. For example, a VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit— perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straighline code, then local scheduling techniques, which operate on a single basic block can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex global scheduling algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they must deal with significantly more complicated tradeoffs in optimization, since moving code across branches is

expensive. In the next section, we will discuss trace scheduling, one of these global scheduling techniques developed specifically for VLIWs.

For now, let's assume we have a technique to generate long, straight-line code sequences, so that we can use local scheduling to build up VLIW instructions and instead focus on how well these processors operate. For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of lock-step operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code
fragment requires ambitiously unrolling loops (as earlier examples) thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Figure 2 below, we saw that only about 60% of the functional units were used, so almost half of each instruction was empty. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled. To combat this code size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D -24(R1),F16			
S.D F20,-32(R1)	S.D -40(R1),F24			DADDUI R1,R1,#-56
S.D F28,8(R1)				BNE R1,R2,Loop

Table-1 Functional units in VLIW approach

Early VLIWs operated in lock-step; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause

the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause all the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued. Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus, different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between successive implementations, or between implementations with different issue widths, more difficult than it is for a superscalar design. Of course, obtaining improved performance from a new superscalar design may require recompilation. Nonetheless, the ability to run old binary files is a practical advantage for the superscalar approach. One possible solution to this migration problem and the problem of binary code compatibility in general, is object-code translation or emulation. This technology is developing quickly and could play a significant role in future migration schemes. Another approach is to temper the strictness of the approach so that binary compatibility is still feasible. This later approach is used in the IA-64 architecture [3]. The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP programs, the original loop probably could have been run efficiently on a vector processor [4]. It is not clear that a multiple-issue processor is preferred over a vector processor for such applications; the costs are similar, and the vector processor is typically the same speed or faster. The potential advantages of a multiple-issue processor versus a vector processor are twofold. First, a multiple-issue processor has the potential to

extract some amount of parallelism from less regularly structured code, and, second, it has the ability to use a more conventional, and typically less expensive, cache-based memory system.

For these reasons multiple-issue approaches have become the primary method for taking advantage of instruction-level parallelism, and vectors have become primarily an extension to these processors.

In the above two sections we have seen how instructions are being parallelized using both hardware and software support. The area that we will be concentrating is the hardware approach to exploiting parallelism. Our main objective is to provide a design that will help exploit parallelism in a much more efficient way and also give an algorithm of its functionalities.

CHAPTER VI

6. The Virtual Processor

The virtual processor (VP) as mentioned earlier will dispatch instructions rather than the tasks to the different cores. This requires a great deal of decision making by the VP. But before getting into details about how the VP will work, let us have a look at the architecture of VP. The VP will have its private data storage that is the instruction queue, where the instructions are brought in from the main memory for dispatching.

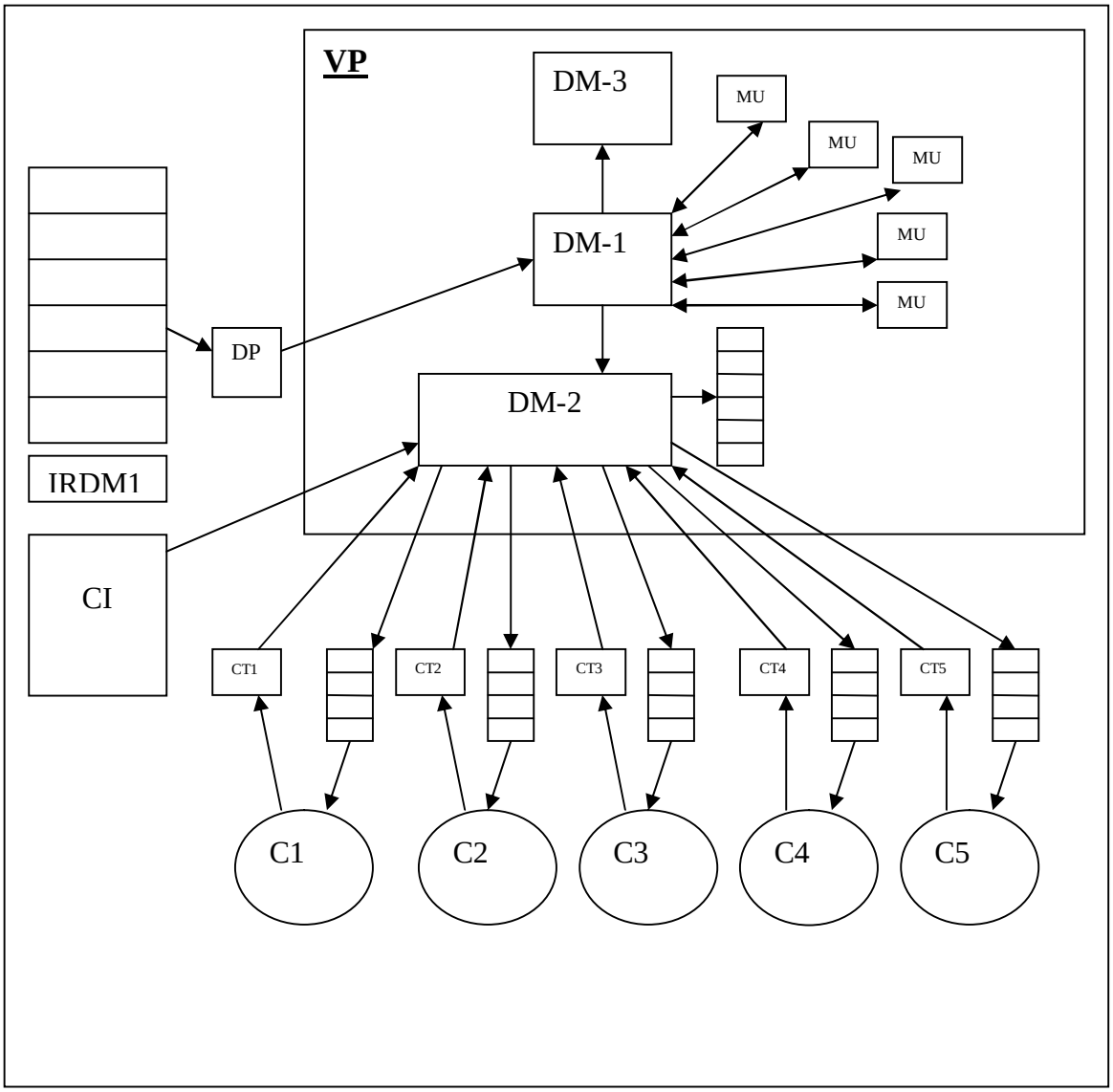

Fig-3 Decision Maker – 2 and the overall architecture

The overall architecture of the virtual processor looks quite like the one drawn above. First and foremost the VP will itself fetch the instruction from the main memory and keep it a temporary queue called the IRDM1 (Instruction Register for Decision Maker - 1) which will hold the instructions until they are fetched by DM-1.

6.1 DM-1 (Decision Maker -1)

One by one instruction will be dispatched by the DP module to another module called the DM-1 (Decision Maker-1). DM-1 is a decision maker that will essentially take the decision whether the instruction just came from the dispatcher is a trivial instruction or a complex instruction. DM-1 will take this decision looking at the instruction's operands and opcode. The incoming instructions will be judged primarily on the operands that it requires to be adjudged trivial or complex instructions. Trivial instructions are instructions whose required registers are not being used by some other instructions currently running or are being issued for running, which means that the issued instructions are waiting in the instruction queue of a core. Complex instructions are instructions whose required registers are being used currently by some other instructions. We can also say that these complex instructions are dependant on other instructions whose registers are being used currently. For example instructions such as ADD R1, R2, R3 and OUT R3, 3 might be trivial instructions because R1 and R2 might not be currently used by any other instructions running or waiting in the instruction queue. The other type of instruction is the complex instruction that is dependant on other instructions currently running or is waiting in the instruction queue. Or in other words we can say that these instructions use registers that are being used by instructions that are currently running or waiting in the instruction queue. After DM-1 has decided upon what sort of instruction it is from the operands it is using, it will dispatch the instruction to the desired DM. If the instruction is trivial then it will be given to DM-2 (Decision Maker-2) else it will be given to DM-3 (Decision Maker-3).

We have discussed above that DM-1 will take the decision whether a fetched instruction is trivial or complex from the operands used by the instruction. But how will it know which operands are being used by some other instruction? DM-1 will get this information from the management unit (MU) that is present in the chip

of the VP which is shared by both DM-1 and DM-3 because both of the modules require information about the operands of an instruction.

6.2 Management Unit

These MUs are located in the VP and used by both DM-1 and DM-3. DM-1 uses the information from the MU in order to know whether the instruction in question is a trivial instruction or a complex instruction.

Whenever an instruction is dispatched to the instruction queue of a core, the information of which registers the instruction will use are kept in that management unit. These management units are few in number, more precisely exactly each for one core. So if there are 5 cores then there will be 5 MUs. Each of the MUs will carry information of the instructions that are situated in the instruction queue of each of the cores. For example if there are 5 instructions I1, I2, I3, I4 and I5 in the instruction queue of C5 then which registers are being used by these instructions will be written in the MU for C5, which is MU5. So DM-3 will know which instruction is using which registers from MU5 and hence any other instruction that needs registers that are in conflict with the ones currently in use will have to be stalled.

For example if there is an instruction such as ADD R1, R2, R3 and it is fetched by DM-1, it will first see whether the operands R1, R2 or R3 are used by any other instruction running or waiting and this information will be found in the MU. If DM-1 decides this to be a complex instruction it will be send to DM-3. Now DM-3 will use information from MU for a different purpose and that is to decide which instruction, currently running or waiting in the instruction queue, is using the registers R1, R2 or R3 so it can decide which instruction ADD R1, R2, R3 depends on.

DM-2 will decide which core the instruction should be dispatched to since it is a vital decision to make for this paper particularly. Each of the cores is modified to perform specialized operation much like the super scalar processor. Each of the cores will be specialized to perform a single type of operation. For example C1 might be specialized to perform arithmetic operation so every time an arithmetic operation comes along C1 will have the first priority provided that it is free at that moment. For example C2 might be specialized to perform load/store operation and C3 might be specialized to perform branch instruction. So far it looks exactly like a super scalar processor but there is a huge difference between the architecture that we are proposing to that of super scalar processor. The difference is that each of the cores will be able to perform specialized operation but it can perform other operations as well. It means, for example, if C1 is specialized to perform arithmetic operation then it will take lesser cycles to perform this type of operation than other cores; the other cores can perform the same type of operation but not as efficiently as C1. Hence, we can say that each of the cores will be able to perform specialized operation but they can perform other operations as well.

6.3 Response Time calculation

So information such as which core is specialized in what instruction is kept in the CI (Core Information) module so that DM-2 can use the information from CI to decide which instruction will be dispatched to which core. After DM-2 has decided upon which core to dispatch the instruction to it will insert the instruction to the private instruction queue of the designated core. For example if C1 is destined for an instruction then DM-2 will dispatch the instruction to the instruction queue of C1. This decision of dispatching the instruction to C1 came after making a calculation about which core will be more efficient which means which core can perform the operation in lesser clock cycles. This calculation is called the “Response Time” calculation which is nothing but the total time taken for the execution of a particular instruction. So this “Response Time” calculation

is done on all the cores including the specialized one for that particular instruction. The core that results in the minimum “Response Time”, that core will be chosen to execute the instruction. For example, if there is an instruction such as LD R3, 0 (R1) and it has been deduced that it is not a dependent instruction, DM-2 will make a “Response Time” calculation on all the cores including the one that is specialized to perform the LD/SW instruction. Generally the specialized core should win but there might be cases where another core will return a minimum “Response Time” result. So that particular core will be selected over the specialized one because the time required or the clock cycles required to perform the LD instruction would be less in that core as opposed to the specialized one.

6.4 DM-3 (Decision Maker - 3)

However DM-3 will act differently as opposed to DM-2 where it only deals with trivial instructions as mentioned before. DM-3, on the other hand, will deal with complex instructions where the instructions either use registers or they are dependant on other instructions. So the operations or the decision making process for DM-3 is a bit critical and it quite complex in comparison to DM-2.

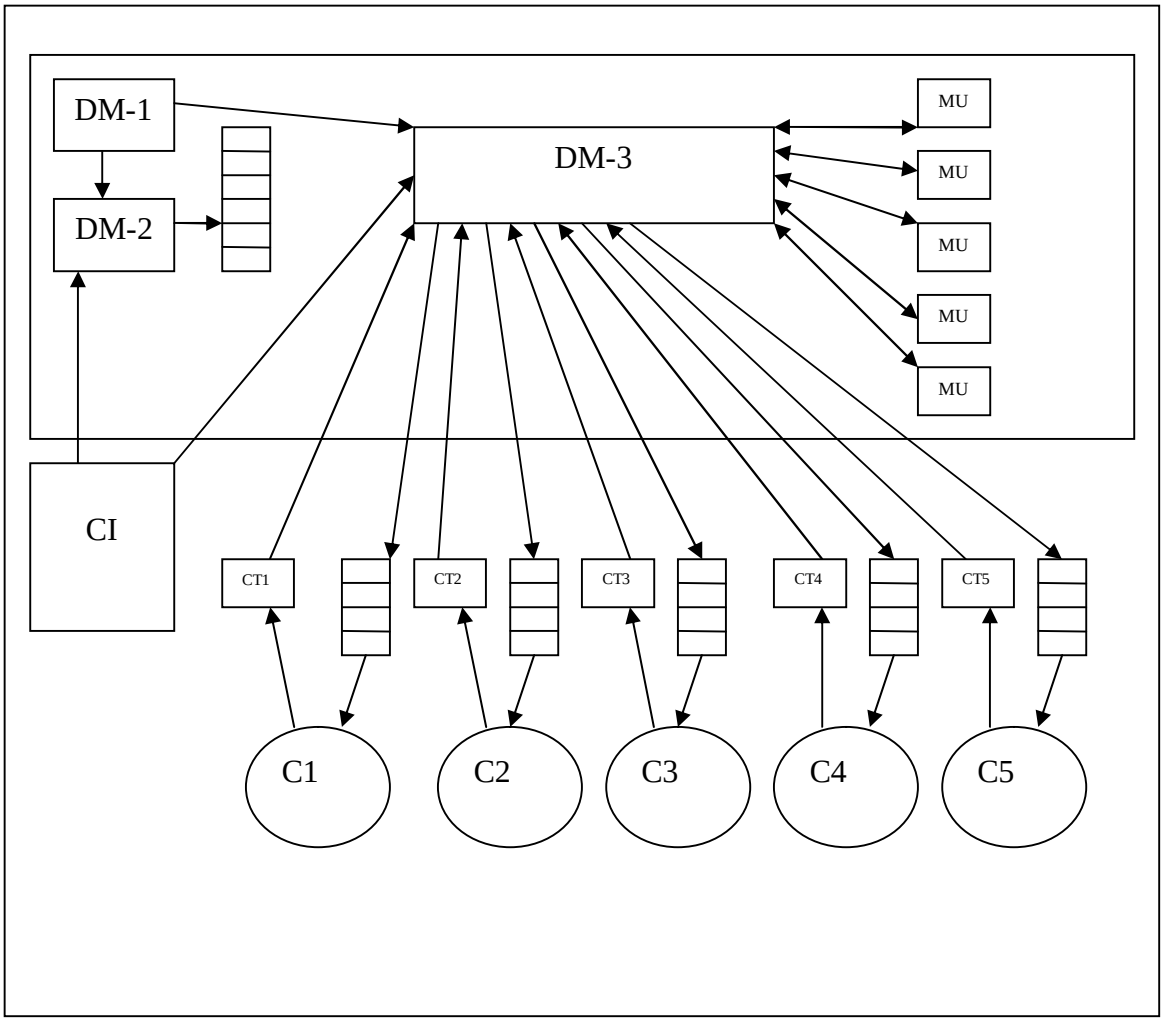

Fig-4 Decision Maker - 3

When DM-1 has decided what type of instruction it is, it either offloads them to DM-2 or DM-3. After DM-3 gets the instructions the first and foremost thing that it has to see is what the registers that the instruction uses are. After that it will look for instructions that are currently running in the cores so that DM-3 will know the registers that are currently being occupied or being used. If the registers that the incoming instruction needs are being used by the instructions currently in the cores then DM-3 will try to stall the instruction or take some other decision in order to eliminate hazards due to register use by two or more instructions. For example if there is an instruction ADD R1, R2, R3 then DM-3 will have to see whether the registers R2 and R3 are being used by some other instructions. If yes, then it will have to stall that instruction until those registers are free again. It

means that the ADD instruction is waiting for write on R2 and R3 to finish. The information of which instruction, which is currently running, uses which registers are kept in a management unit (MU). There are many MUs like MU1, MU2 and so on depending on the number of cores there are in the machine. Each of the MUs are associated with each core, meaning, for core-1 (C1) there is MU1, for C2 there is MU2 and so on and so forth. When an instruction is issued to the instruction queue of any core, the operands of that instruction makes a reservation in the core's corresponding MU.

Every time an instruction is issued to a core, information about that instruction is kept in the MU that is associated with the core the instruction has been assigned to. For example, if there is an instruction ADD R1, R2, R3 and while being issued to, let's say, core-1, information about what registers (operands) this instruction is using will be kept in MU1 which is the management unit associated with this core.

Now the question is how long the instruction should wait before it is issued to the core. Since the MU has information on which instruction has been assigned to which core, DM-3 can easily use this information to decide how long the execution time of a particular instruction that is currently running is. It will do a very simple mathematical calculation in order to decide how long the instruction must be stalled before it is being issued to a core. The minimum time an instruction must wait is the time until which the instruction it depends on finishes. Let's consider the following example:

```
ADD R1, R2, R3
SUB R4, R5, R1
```

We can see that SUB instruction depends on the ADD instruction. SUB instruction would need the value of R1 after it is written by the ADD instruction. We can also see that the number of registers used by ADD instruction that SUB

instruction needs is one. Let's assume that the ADD instruction is running on C1 which is the core specialized for arithmetic operation. If we use any other algorithm it will invariably assign the SUB instruction to C1 both because it is yet another arithmetic operation and also because the instruction SUB depends on is in C1. But the algorithm we are proposing would consider other cores before assigning an instruction to a core. We know that the minimum time the SUB instruction must wait is the time required for the completion of the ADD instruction. Beyond that the same "Response Time" calculation has to be made in order to decide which core would be the best, in terms of efficiency, to dispatch the instruction to. Nevertheless there could be a complex situation where the calculation of the minimum time an instruction must wait could be quite tricky. Let's consider such a case:

```
LD R1, 0 (A0)
ADD R4, R2, R3
SUB R5, R6, R7
MUL R5, R1, R4
```

We can see that LD, ADD and the SUB instruction are all independent instructions but MUL depends on the all three because it requires registers from all the instructions prior to it. Let's also assume that all the three independent instructions are running on three different cores. So, now, how long should the MUL instruction wait before being issued to any core? Well, the answer is that it must wait for as long as the maximum time required by either of the instruction. DM-3 will calculate the time required by each of the independent instruction and then stall the MUL instruction for the maximum of the three times that were calculated. Beyond this point the "Response Time" calculation will be made in order to decide which core should the instruction be dispatched to making the best use of the clock cycles. This "Response Time" calculation in DM-3 is discussed in the following paragraph.

6.4.1 Response Time calculation for DM-3

Even after these decisions of how long to stall an instruction, another decision has to be made and that is the optimization criteria for dispatching the instructions. We have mentioned earlier that each of the cores is specialized to perform single operations but it can also perform other type of operations but not efficiently. So this assumption is helpful for making certain decision making calculations that will help the DM-3 dispatch instructions in a more efficient way. For example, an instruction ADD R1, R2, R3 is destined for C2 which is specialized to perform arithmetic operation. But the instruction queue of C2 has only one slot free so it is obvious from the algorithm designed so far that it will be assigned to C2 because C2 is more efficient in performing arithmetic operation. But this might not always be efficient due to latencies in the instruction queue of the destined core. If ADD R1, R2, R3 requires 2 clock cycles to finish in C2 and there are other instructions in the queue that will take a total of 5 clock cycles then the total number of clock cycles will that is required for the ADD instruction to complete is (5+2) 7 clock cycles. But, now, if, lets say, C3 (not specialized to perform arithmetic operation but it can still do so but in greater clock cycles) has 3 slots free and the instructions in its queue takes 2 clock cycles to finish and the ADD instruction, in this core, takes 4 clock cycles then the total amount of clock cycles required for the completion is 6 clock cycles which is still 1 cycle less than the core it was supposedly destined to earlier. So DM-3 has to do calculations such as these to optimize the decision making criteria before it dispatched the instructions to the cores.

6.4.2 Optimization for DM-3 (Dependency related calculation)

The other calculation that DM-3 has to make is the dependency related calculation. Let us consider the following code segment:

```
LD    R3, 0 (R0)
ADD R1, R2, R3
SUB R4, R1, R5
SW   R4
```

In the above code segment ADD instruction depends on LD instruction, SUB depends on ADD and subsequently SW depends on SUB. Here LD produces a result that is needed by the ADD instruction and so on and so forth. This is a classic example of data dependence. We know from the algorithms given by Robert Tomasulo for dynamic scheduling [2] that these sorts of dependant instructions has to be placed in the same core, otherwise it will create data hazards such as RAW, WAW and WAR. But the algorithm that we are proposing will not require the dependant instructions to be executed in the same core and this can be accomplished with a calculation that DM-3 will have to do before dispatching.

Let's assume the LD instruction is in the instruction queue of C3 which is specialized for load/store instructions and it has other instructions in the queue before it. So the total clock cycles required for this instruction is, let's say, 5 clock cycles, which is the clock cycles required for the completion of the LD instructions. The ADD instruction will have to be stalled for at least 5 clock cycles if not more. There is a possibility that it might stall for more than 5 clock cycles if there are other instructions in the queue before the ADD instruction and this phenomenon will take place if we follow any algorithm for dynamic scheduling. But the algorithm that we are proposing will try to avoid such a phenomenon as much as possible.

If we follow any algorithm of dynamic scheduling it suggests that all the dependant instructions have to be placed in the same core in order to avoid any sort of hazards. So if ADD instructions take 2 clock cycles in addition to the 5 clock cycles taken by the LD instruction and moreover the clock cycles required

for the other instructions before ADD which takes, let's say, 4 clock cycles, there will be a stall of 11 clock cycles before ADD instruction gets executed. But we can minimize the clock cycles required for the ADD instruction by making it stall or wait for a shorter period of time. Let's see how.

Let's take the example used above. We know that LD instruction takes 5 cycles to complete so rather than issuing the ADD instruction in the same core as the LD instruction we can right away issue the ADD instruction to another core after the LD instruction has completed and the value in R3 is available. The algorithm used by DM-3 will check which core will take the least possible clock cycle in addition to the 5 cycles used by the LD instruction. For example, if there is a core that is totally free and to perform the ADD instruction it will take 4 cycles, then the total stall time for ADD instruction to complete is 9 clock cycles, so we have saved 2 cycles. For example, if the ADD instruction is issued to the core specialized for arithmetic operation and it has 2 more instructions before it but takes only 3 clock cycles to finish then we can save up to 3 cycles.

Let's illustrate the above example with diagrams.

……	DIV A4, A2, A3	SUB A0, A0, A1
……	MUL A14, A15, A16	SW 16 (R0), R3
LD R3, 0 (R0)	ADD A12, A13, A14	ADD R8, R9, R10
…….....	SUB T3, T4, T5	JMP L3
……	SUB T0, T1, T2	CALL L2
SW 12 (T0), R6	MUL A3, A2, A1	BNE L1, 0
Ins Q of destined core	**Ins Q of specialized core**	**Ins Q of any other core**

Fig-4 Three different types of instruction queues

Since we are, for the time being, dealing with the dependency between the LD instruction and ADD instruction we will be concerned with the dispatching of the ADD instruction while calculating which core the instruction should be dispatched to. We can see from the diagram above that instruction queue for the destined core has the LD instruction which is also the core specialized for LD/SW instructions. If we follow any other algorithm other than the one we are proposing it will be straight away dispatched to the core where the LD instruction is because ADD instruction depends on the LD instruction. But the algorithm that we are proposing will try to find out the exact core to where the instruction can be dispatched in order to execute it in the least possible clock cycles.

First of all it will calculate the clock cycles required for the execution of ADD instruction in its specialized core then it will calculate the clock cycles required for its execution in the core where the LD instruction is and finally it will calculate the clock cycles of the rest of the cores. After this calculation has taken place it will see which among the clock cycles calculated has the least response time, which means which core will take the least possible time to execute the ADD instruction.

It is not always true that the core specialized to perform arithmetic operation will always perform such instructions efficiently than the others. It is possible for another core to perform the arithmetic instruction more efficiently (taking lesser cycles) than the core specialized for it. For example, in the core specialized for arithmetic operation the ADD instruction might take 6 clock cycles whereas it was supposed to take 2 clock cycles had this instruction be the only instruction in the queue of the core. Since it is taking 6 clock cycles there must be other instructions before the ADD instruction that is already running or waiting in the instruction queue of that core. So dispatching the ADD instruction in another core which is not specialized for arithmetic operation could be a better option.

Once the instruction has completed, the information in the MU about that instruction has to be freed so that some other instruction can reserve information for it. Just to recap on what MU does, there is one MU for each core and any information about any instruction left in the MU means that the instruction has been assigned to that core. For example if MU1 has some information about any instruction then that instruction is sure to be in C1.

Since the instructions are executed out of order, the results of those instructions have to be placed in program order. Reorder buffer (ROB) is one such device that tags each an every instruction that is dispatched to the cores out of order and later brings the result in the original program order. This tagging is done according to the indices in the ROB [2].

CHAPTER VII

7. The algorithm

So far what we have discussed is basically the algorithm that the VP will follow in order to execute its functionalities. The algorithm was mainly conceptualized with examples in all of the sections above. Following is the algorithm in the form of a pseudo code.

First and foremost the algorithm for the DM-1 where it will make a small decision as to whether the instruction it has fetched is a trivial one or a complex one.

```
fetchDispatch()//done by DM-1
{
   Fetch instruction in IRDM1;
If(isDependent(ins) or !isOut(ins))
{
        dispatchDM3();
}
Else
        dispatchDM2();
}
```

Next comes the detailed structure for the function called in the conditional statement called isDependent() that basically says whether the instruction is dependent or not and decides this by using the information from the management units. We have also seen another function called isOut() that basically says whether the instruction is an output statement. By output statement I mean whether the instruction just prints a value in the console or not. These sort of output statement doesn't write any value into the memory; it just prints a value in the console.

```
isDependent(ins)//done by DM-1
{
        for each MU
        {
                for i=1 to 48
                {
                    if (ins.operands == i)//ins.op1 || ins.op2 || ins.op3
                    {
                        flag=1
                        break;
                    }
                }
                if (flag==1)
                    break;
        }
        if(flag==1)
                return true;
        else
                return false;
}
```

Next comes the functionalities of DM-2 where it mainly takes decision as to which core should the trivial instruction be dispatched to by making a number of decisions aided by certain calculations.

```
DM2(ins)//DM-2 has the trivial instruction
{
        s=findCore(ins.opcode);//return core that is specialized for
                                        //ins.opcode
        Mset = findAll(s);//find cores except the specialized one, s
        X = coreResponse(s, Mset);//return core with minimum response
                                        //time
        dispatchToX(ins);//dispatch instruction to the core
                                //with minimum response time
        Manager(ins.op1, ins.op2, ins.op3);//input information in the MU
}
```

The information about the specialized core is kept in a table where the core that is specialized for a type of instruction is kept against the type of instruction. And this information is kept in a module called the core information (CI). The CI has four fields and they are cycles, opcode, core and number as the following:

Cycle	opcode	Core	No
3	ADD/ SUB	C1	1
4	FP	C2	2
2	LD/SW	C3	3
2	JMP	C4	4

Table-2: Two dimensional structure of Core Information (CI) module

```
findCore(ins.opcode)
{
        For i = 2 to 4
                If (opcode[i] == ins.opcode)
                        Return core;

}
```

CHAPTER VIII

8. Conclusion

The main aim of this paper is to provide yet another approach to instruction level parallelism and to aid the operating system in making some vital decisions in order to parallelize instruction and dispatch them dynamically to an asymmetric set of cores in the chip multi-processor. This dynamic scheduling of instruction to exploit parallelism is done by a hardware that works as an interface for both the operating system and the chip multi-processor. That hardware is the Virtual Processor.

The architecture that I have proposed in this paper takes care of two problems and they are; the operating system has a lot to do in order to dispatch the instructions to the cores and hence as the number of cores change so does the operations of the OS. The architecture that is proposed in this thesis takes care of that problem. The operating system doesn't need to know the number of cores there are in the machine, all it needs to know is that it has to provide the VP with instructions and then its work is done. The operating system will only do so much. The other problem is that the numbers of cores are symmetric in today's chip multiprocessors (CMP) which means the number is two, four, eight, and sixteen and so on. The design that has been proposed in this paper does not require the number of cores to increase in that progression. The number can be any is it three, four, five or more.

The deciding factor is whether it is going to be efficient or not. This paper does not have any results from any simulations in order to justify the claim of a hardware support trying to solve all the problems. But nevertheless this design is sure to work and it most definitely is a force to reckon with in today's world where the number of cores in the commercial industry is growing very rapidly. Soon a

time will come when the complexity of the operating system reaches its utmost height and have to resort to an architecture that takes some loads off the OS.

CHAPTER IX

9. List of References

[1] Patterson & Hennessy, “Computer Architecture, a quantitative approach” *Instruction level parallelism and its dynamic exploitation,* 3rd ed. USA, Elsevier Science, 1990, 1996, 2003, p.172-253.

[2] Patterson & Hennessy, “Computer Organization and Design: The Hardware/Software Interface” *Pipelining,* 3rd ed. USA, Elsevier Science, 1997, p.370-432.

[3] Patterson & Hennessy, “Computer Architecture, a quantitative approach” *Exploiting Instruction level parallelism with software approach,* 3rd ed. USA, Elsevier Science, 1990, 1996, 2003, p.304-350.

[4] Patterson & Hennessy, “Computer Architecture, a quantitative approach” Appendix G*,* 3rd ed. USA, Elsevier Science, 1990, 1996, 2003, p.172-253.

[5] Julia Chen, Philo Juan, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh and Margaret Martonosi, “Hardware-Modulated Parallelism in Chip Multiprocessors”, in *International Symposium on Computer Architecture,* June 2006

http://www.kroening.com/diploma/main002.html