TRADEOFFS IN INSTRUCTION FORMAT DESIGN FOR HORIZONTAL ARCHITECTURES

by

Gurindar S. Sohi and Sriram Vajapeyam

Computer Sciences Technical Report #810

December 1988

TRADEOFFS IN INSTRUCTION FORMAT DESIGN FOR HORIZONTAL ARCHITECTURES

Gurindar S. Sohi and Sriram Vajapeyam

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706.

Abstract

With recent improvements in software techniques and the enhanced level of fine grain parallelism made available by such techniques, there has been an increased interest in horizontal architectures and large instruction words that are capable of issuing more that one operation per instruction. This paper investigates some issues in the design of such instruction formats. We study how the choice of an instruction format is influenced by factors such as the degree of pipelining and the instruction's view of the register file. Our results suggest that very large instruction words capable of issuing one operation to each functional unit resource in a horizontal architecture may be overkill. Restricted instruction formats with limited operation issuing capabilities are capable of providing similar performance (measured by the total number of time steps) with significantly less hardware in many cases.

1. INTRODUCTION

To exploit fine grain parallelism, a high-performance processor must provide a set of functional units that comprise an underlying *resource architecture* and an operation issuing mechanism to provide work to the functional units. Fine grain parallelism amongst operations of different types can be exploited by using multiple functional units, each of which performs a different arithmetic/logic operation and by issuing operations to these functional units in parallel. Fine grain parallelism amongst operations of the same type can be exploited by pipelining the functional unit (used by the operations) to a greater extent or by providing multiple copies of the same functional unit.

The resource architecture is driven by an operation issuing mechanism that reads source operands from a set of registers (*register file*) and routes them to the functional units via a datapath (*input interconnect*). Once the results of the operations are available at the output of the functional units, they are routed back to the register file via another datapath (*output interconnect*).

Traditionally, the operation issuing mechanism in most machines has been limited to issuing at most one operation[1] (a single instruction) in a single clock cycle in spite of the presence of multiple functional units that could accept more than one operation per clock cycle [1, 14]. This is sometimes referred to as the *Flynn bottleneck* [6]. For many years this bottleneck was not considered to be very significant because of the limited amounts of fine-grain parallelism that could be detected in most applications [13, 16]. In the absence of sophisticated compiler techniques to detect and enhance the available fine-grain parallelism and the enormous hardware required to do the same, a peak operation issue rate of 1 operation per cycle seemed adequate.

However, recent work, has suggested that the amount of fine-grain parallelism that is available in an application could be enhanced considerably by the use of software techniques, especially for computation-intensive scientific programs [9]. With an enhanced level of fine-grain parallelism in the application programs, operation issuing mechanisms that allow for the initiation of several operations simultaneously may be in order.

Architectures with enhanced operation issuing capabilities have aroused considerable interest in the computer architecture community and several have appeared recently. These include VLIW machines such as the ELI-512[5] and its follow-on TRACE family of machines [3], the ESL Polycyclic processor[11] and its follow-on Cydrome Cydra 5 Departmental Supercomputer [12], and decoupled architectures such as the ZS-1 [15]. Even some single-chip microarchitectures attempt to issue more than one operation in a clock cycle [10].

VLIWs and the Cydra 5 provide instruction formats where each instruction is capable of issuing several operations in a single clock cycle. Such architectures are also called *horizontal architectures* and their instructions are called *horizontal instructions*. Proposed horizontal architectures make extensive use of software support to detect parallelism and pack several independent operations into a horizontal instruction. A horizontal instruction provides the basic multiple operation issuing mechanism. Architectures such as the ZS-1 use more traditional instruction formats that issue only one operation per instruction; multiple operation issue is achieved by issuing more than one operation per cycle at run time.

[1] In this paper, we distinguish between "operations" and "instructions". An instruction consists of one or more operations.

In this paper, we study horizontal architectures whose instruction formats are capable of issuing multiple operations with a single instruction. In designing such architectures, one of the first questions that a computer architect must answer concerns the number of operations that each instruction should be capable of issuing. There are several tradeoffs to consider. Let us illustrate this with the help of an example.

Suppose that a machine has to be designed for a hypothetical task that consists of 3 independent floating-point ADD operations and 2 independent floating-point MUL operations. Assume that the floating-point ADD and MUL operations take 5 and 6 clock cycles to execute, respectively. Using a resource architecture of 3 adders and 2 multipliers and an instruction format capable of issuing 5 operations simultaneously, one could achieve a best-case execution time of 6 clock cycles. To support this execution, the register file and the input interconnect would have to be capable of supplying 10 operands per cycle to the functional units. However, an execution time of 6 clock cycles could be achieved even with 2 multipliers, 2 adders, an instruction format capable of issuing 4 operations simultaneously and a register file/input interconnect capable of supplying 8 operands per cycle if the adders are pipelined and are able to accept a new request on every clock cycle. Continuing further, an execution time of 7 (10) clock cycles could be achieved with only one pipelined multiplier, one pipelined adder, an instruction format capable of issuing 2 (1) operations and a register file capable of supplying 4 (2) operands per cycle.

The above example serves to illustrate one major tradeoff in the design of a horizontal instruction format, namely the capabilities of an instruction format (and the associated hardware needed to support it) versus the number of time steps taken to execute a task using the instruction format. As we shall see, the tradeoff is complicated further by several factors. While the choice of an appropriate instruction format is clearly a very important one, it has not been studied widely in the literature. Previous studies have concentrated on full-fledged instruction formats that are capable of issuing one operation to every functional unit in the resource architecture in a single instruction [4]. This may be due to the fact that the machines capable of issuing more than one operation per instruction that have been built so far use full-fledged horizontal instruction formats [2, 3, 5, 12].

In this paper, we are interested in determining if full-fledged horizontal instruction formats (and the enormous amount of hardware needed to support them) are indeed worthwhile or might restricted instruction formats that can issue only a few operations at once be adequate. To do so, we study various horizontal instruction formats and see how they are influenced by and how they impact other architectural factors and decisions. Our studies use a hypothetical horizontal architecture that resembles (though not exactly) some recently announced horizontal architectures. The factors that we consider include the degree of pipelining of each resource, the view of the registers presented to the operations, and the parallelism available in the programs.

The outline of this paper is as follows. Section 2 discusses our machine model, the benchmarks that we use and our evaluation methodology. Section 3 discusses the maximum performance potential. Section 4 presents a detailed study of several horizontal instruction formats. Finally, section 5 presents a summary and some concluding remarks.

2. MACHINE MODEL AND EVALUATION METHODOLOGY

2.1. Machine Architecture and Instruction Format Considerations

The basic operation set of most horizontal architectures consists of "simple" load/store, register-register operations. Simple operations are used because more complex operations can be broken into a sequence of simpler operations that allow the software more flexibility in optimization. Each operation is simple in that it can be examined, decoded and issued in a single clock cycle. However, it may take several clock cycles to actually complete execution. We use a similar operation set in our machine model. Arithmetic operations use a 3-address format; 2 source operand addresses and 1 destination operand address. Load and store operations use a 2-address format. The operation set includes floating point operations as well as integer operations. All operations have scalar operands; no vector operations are used.

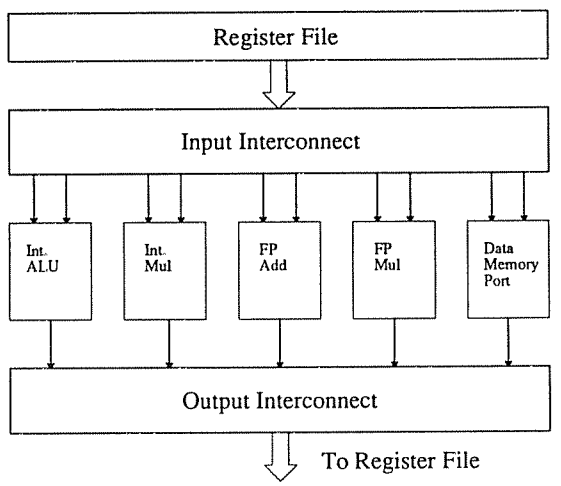


Figure 1: The Model Horizontal Architecture

The resource architecture that we consider in this paper consists of an integer adder/logical unit, an integer multiplier, a floating point adder, a floating point multiplier and a port to the data memory (Figure 1). We study two degrees of pipelining for each resource: (i) a modest degree of pipelining which we shall refer to as *modest pipelining* and (ii) a greater degree of pipelining which we shall refer to as *deep pipelining*. The number of pipeline stages in each of the functional units for the two cases is given below.

Functional Unit	Modest Pipelining	Deep Pipelining
Load	6	12
Int. Add	1	2
Int. Mul.	4	8
FP Add	3	6
FP Mul.	4	8

The number of pipeline stages for a system with modest pipelining are taken from [4] and are representative of moderately pipelined machines such as the TRACE family; the number of pipeline stages for deep pipelining are twice that of modest pipelining and are representative of highly pipelined machines such as the CRAY class of machines.

The computational units in Figure 1 are used in the obvious manner. Only memory data references proceed through the memory port; instruction references access an instruction cache and, to simplify matters, we assume that all instruction references hit in the instruction cache (not shown).

Fine-grain parallelism amongst operations that use distinct functional units can be exploited easily with the resource architecture of Figure 1. Fine-grain parallelism amongst operations that use the same functional unit can be exploited by providing more copies of the functional units, by increasing the number of pipeline stages in each unit, or by both. We do not consider multiple copies of each functional unit for two reasons. First, as we shall see in Section 3, the best-case performance of the resource architecture of Figure 1 is not substantially different from the best-case performance of a machine with unlimited resources *for our benchmark programs*, especially if the degree of pipelining is high. Second, the resource architecture of Figure 1 provides an excellent starting point for any study of horizontal architectures. Unless we understand the issues involved in exploiting the resource architecture of Figure 1, the study of more complex resource architectures can be quite frustrating. Finally, the resource architecture of Figure 1 is of significant current interest (notice the similarities between it and the resource architectures of the TRACE 7/200 and Cydra 5).

To provide operands to the operations in a horizontal instruction, the register file must be designed accordingly. Important factors include: (i) the view of the registers presented to the operations/instructions, and (ii) the number of read and write *ports* that must be provided.

The registers could be organized so that the instruction set views them either as a set of *shared* registers that are used both for integer and floating-point operations or as a *split* register file with a distinct set of registers for integer and floating-point operands. A shared register file is used in the Cydra 5 whereas a split register file is used in the TRACE machines. Apart from the splitting of registers, another issue that must be considered is the *partitioning* of registers of a particular type. Partitioning is needed to reduce the number of read and write ports for very long instruction formats [3]. However, the partitioning of a set of registers is important only if there is more than one functional unit of each type and we shall not consider it in this paper.

A shared register file has three distinct advantages over a split register file. First, a shared register file provides the best opportunity for a compiler to schedule operations in parallel since operations can be scheduled without being constrained by their type. More on this in section 4. Second, a shared register file is able to achieve a better utilization of registers. Third, a compiler's job is easier if it is presented with a unified view of the registers [17].

Unfortunately, the number of ports that must be provided (and consequently the complexity of the interconnects) is increased if a shared register file is used in conjunction with a horizontal instruction format. The number of register read and write ports that must be provided is dictated by the horizontal instruction format. If P independent operations are packed into a horizontal instruction, the register file must have $2P$ read ports to provide the operands and P write ports[2] to accept results from the functional units (note that loads and stores require fewer ports). Once the design of the register file and the horizontal instruction format has been fixed, the design of the input and output interconnects is determined automatically since the interconnects must be capable of routing as many operations from and to the register file(s) as is specified by the instruction format.

We use 2 metrics to measure the "goodness" of a horizontal instruction format. They are: (i) the number of time steps (clock cycles) taken to execute a program, and (ii) the code density of the program using the horizontal instruction format. The code density (a static measure) is the average number of useful (non-NOP) operations that are present in a horizontal instruction. NOPs must be used if no useful operation can be found to fill an operation slot in a horizontal instruction. Less dense code implies wasted memory space, unless an encoding of the horizontal instructions is used (as is the case in the TRACE machines). More importantly, the code density is an indicator of how well the capabilities of the instruction format are actually being used. More on this in section 4.

2.2. Experimental Methodology, Benchmark Programs and Performance Metrics

To evaluate the different performance tradeoffs, we carried out several experiments using a set of benchmark programs. The benchmark programs that we use are the original 14 Lawrence Livermore loops [8]. (Also see[7] for related experiments using some more benchmarks). These benchmarks are easily understood and have been used widely in the evaluation of high-performance numeric processing machines. We assume that the reader is familiar with these benchmarks. The programs are hand-compiled into our 3-address operation set assuming that only one operation can be issued per instruction and assuming an unlimited number of registers[3]. An unlimited number of registers is assumed because we did not want our results to be influenced by the spilling operations introduced due to a limited number of registers. Furthermore, the number of architectural registers that one provides are always subject to debate.

Loop unrolling is used to enhance the available fine-grain parallelism. More loop unrolling implies more fine-grain parallelism. After unrolling, the benchmarks are fed into a scheduler (discussed in the next section). The scheduler compacts the operations into a horizontal instruction format subject to resource and instruction format constraints.

Using the resulting code we calculate (i) the *per iteration* execution time, (ii) the *code density* of the resulting code and (iii) the *total execution time* which is the sum of the individual execution times of each of the benchmarks. We assume that an instruction can be issued in a single clock cycle, irrespective of its width. All our results for the execution times are presented in clock cycles. We do not attempt to convert clock cycles into actual times since many of the factors that influence the clock

[2]Providing additional read ports can be accomplished by a simple replication of the registers. Providing additional write ports is more complex. We shall not discuss ways of providing additional write ports in this paper; suffice it to say that providing additional write ports can be quite cumbersome in most technologies.

[3]This assumption has also been made in [4].

are highly dependent upon the technology and the implementation. Keep in mind, however, that the clock cycle for a deeply pipelined system is smaller than the clock cycle for a modestly pipelined system.

2.3. The Scheduler

All our experiments are driven by a *scheduler* that schedules operations in the benchmark subject to the constraints of the horizontal instruction format and compacts the operations into horizontal instructions. The scheduler uses algorithms similar to those used in the Bulldog compiler [4]. It attempts to generate an optimal compaction of operations into horizontal instructions; however, optimality is not guaranteed.

The scheduler first builds a DAG representing the dependencies in the basic block. The nodes of the graphs are the individual operations and the edges are the dependencies. Each node is then assigned a *depth*. The depth of a node is the longest path to the node from any entry point of the DAG, plus the time taken to execute the operation in the node itself. The depths are used as priorities in scheduling the instructions.

After the depths have been assigned, operations are scheduled bottom-up, i.e., starting from the exit points in the DAG, and propagating up to the entry points. Scheduling is done on a first-come first-served basis, with the depths used to assign priorities to operations that are ready to be scheduled at the same time. Constraints are placed on the scheduler by the number of operations of each type that can be issued simultaneously (this is dictated by the number of read ports in the register file that supplies operand values and the number of functional units of each type) and also by the number of operations that can complete execution simultaneously (number of write ports in the register file).

3. MAXIMUM PERFORMANCE POTENTIAL

Before evaluating various horizontal instruction formats, let us consider the best-case performance that can be achieved with the parallelism available in the benchmarks. Table 1 presents the best-case per iteration execution time (in time steps or clock cycles) for the benchmarks for various degrees of unrolling both for modest (Mod.) and deep (Deep) pipelining. An unrolling of 1 implies no loop unrolling. Unrollings greater than 10 have not been shown because they did not result in significant additional improvement in the *total execution time* (though individual loops with none or very few loop-carried dependencies did benefit from additional unrolling with unlimited resources in some cases).

The results of Table 1 assume an *unlimited* number of functional unit resources, unlimited number of read and write ports, and a horizontal instruction format capable of issuing an unlimited number of operations. Hence the best-case execution time is determined solely by the length of the critical path. Since the length of the critical path (in terms of clock cycles) is doubled in the case of deep pipelining with unlimited resources, the execution times for deep pipelining are exactly twice the execution times for modest pipelining. The total execution time is the sum of the time taken to execute all iterations of all the loops (implicit in this calculation is a multiplication of the per-iteration execution times by the number of iterations in each loop).

Table 2 presents the best-case execution times for the horizontal architecture of Figure 1 for the cases of modest and deep pipelining using a full-fledged horizontal instruction format capable of issuing one operation to each functional unit in a single instruction (a total of 5 operations). Figures 2(a) and 2(b) compare the execution times of the model architecture of Figure 1 and an ideal horizontal architecture capable of issuing an unlimited number of operations to an unlimited number of functional units in a single instruction. Bar charts are used in Figure 2 because of their visual appeal. Unless shown otherwise, subsequent results in this paper will use bar charts to present the results as a total or an average for all the benchmarks.

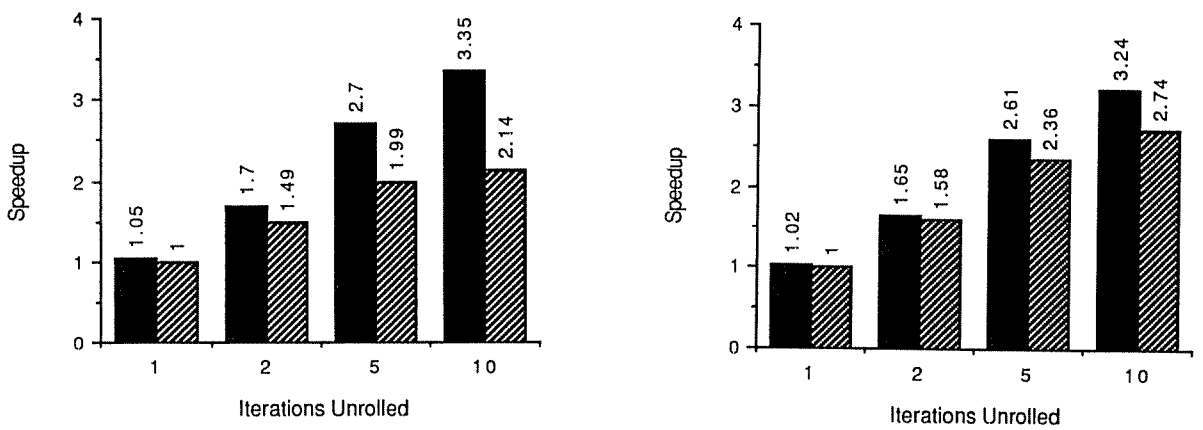
As we can see from Table 1, Table 2 and Figure 2, the limited resource architecture of Figure 1 is not a major impediment to the exploitation of the parallelism available in the benchmark programs (even with a high degree of loop unrolling), especially if the degree of pipelining is high. In some cases, additional performance could be obtained by providing

Table 1: Execution Times with an Unlimited Resource Architecture and Unlimited Operation Issue

Loop	Iterations Unrolled							
	1		2		5		10	
	Mod.	Deep	Mod.	Deep	Mod.	Deep	Mod.	Deep
1	22	44	11.5	23.0	5.2	10.4	3.1	6.2
2	21	42	11.0	22.0	5.0	10.0	3.0	6.0
3	14	28	8.5	17.0	5.2	10.4	4.1	8.2
4	14	28	7.5	15.0	3.6	7.2	2.3	4.6
5	28	56	24.5	49.0	22.4	44.8	21.7	43.4
6	29	58	25.0	50.0	22.6	45.2	21.8	43.6
7	32	64	16.5	33.0	7.2	14.4	4.1	8.2
8	34	68	17.5	35.0	7.6	15.2	4.3	8.6
9	26	52	13.5	27.0	6.0	12.0	3.5	7.0
10	35	70	17.5	35.0	7.6	15.2	4.3	8.6
11	10	20	6.5	13.0	4.4	8.8	3.7	7.4
12	11	22	6.0	12.0	3.0	6.0	2.0	4.0
13	51	102	26.0	52.0	11.0	2.0	6.0	12.0
14	50	100	25.5	51.0	10.8	21.6	5.9	11.8
Total (x1000)	94	188	58.1	116.2	36.7	73.3	29.5	59.0

Table 2: Execution Times with the Resource Architecture of Figure 1 and Full-Fledged Instruction Formats

Loop	Iterations Unrolled							
	1		2		5		10	
	Mod.	Deep	Mod.	Deep	Mod.	Deep	Mod.	Deep
1	24	45	14.5	25.0	8.8	13.0	7.2	9.0
2	30	50	21.0	31.0	17.0	20.2	16.2	17.6
3	15	29	9.0	17.5	5.4	10.6	4.2	8.3
4	15	29	9.5	16.5	6.4	9.0	6.0	6.7
5	28	56	24.5	49.0	22.4	44.8	21.7	43.4
6	29	58	25.0	50.0	22.6	45.2	21.8	43.6
7	33	64	23.0	34.0	19.6	22.0	18.5	19.6
8	48	71	45.0	48.5	41.0	43.0	37.7	38.7
9	30	53	21.0	30.5	15.6	18.8	13.5	14.8
10	36	70	24.0	37.0	22.0	23.6	22.0	22.1
11	10	20	6.5	13.0	4.4	8.8	4.0	7.4
12	11	22	7.0	12.5	5.0	6.8	5.0	5.1
13	51	102	26.5	52.0	20.0	22.2	20.0	20.1
14	51	101	27.0	52.0	16.0	23.2	14.8	15.7
Total (x1000)	98.8	191.4	66.2	121.2	49.7	81.0	46.1	69.8



(a) Modest Pipelining **(b) Deep Pipelining**

Solid Bars : Ideal Architecture
Hatched Bars : Model Architecture

Figure 2: Comparative Performance of an Ideal and the Model Architectures

more resources (for example loops 2 and 7). Overall, the performance could be improved by a factor of 1.56 by providing additional resources if the resources are moderately pipelined and each loop is unrolled 10 times to enhance the available parallelism (it is interesting to compare this number with a factor of 1.48 for the TRACE 14/200 versus the TRACE 7/200 for all 24 Livermore loops[3]). If the resources are deeply pipelined, this factor is only 1.18.

To support execution of a full-fledged instruction format for the architecture of Figure 1, a shared register file needs 10 read ports and 5 write ports. If the register file was split into separate floating-point and integer register files, the floating point register file would need 5 read ports (4 for arithmetic operations and 1 for a store) and 3 write ports (2 for arithmetic operation results and 1 for the destination of a store). The integer register file would need 6 read ports (4 for arithmetic operations, 1 for the address of a memory operation and 1 for the store of an integer register) and 3 write ports (2 arithmetic operations plus 1 integer load).

A full-fledged horizontal instruction format will work very well in executing a program that has *precisely the same number of independent operations of each type*. However, such a format could be quite wasteful of instruction space and interconnection datapaths if the operations are unbalanced

(more operations of one type) and if similar performance could be achieved with more restricted formats. Therefore, we consider restricted instruction formats for the resource architecture of Figure 1.

4. RESTRICTED HORIZONTAL INSTRUCTION FORMATS

4.1. One Operation Per Instruction

A first step would be to issue only a single operation per clock cycle as in any conventional machine. By doing so, we are exploiting the available fine-grain parallelism by pipelining alone. However, by studying the performance with a single operation issue per clock, we can establish a good lower bound on the performance of the resource architecture of Figure 1.

Shared Register File

To support the issue of one operation per cycle, a register file needs to provide 2 operands per cycle to the functional units and be able to accept 1 result from them, i.e., have 2 read ports and 1 write port. The input interconnect needs to be capable of routing two operands from the register file to the inputs of an arbitrary functional unit in every clock cycle and the output interconnect needs to be capable of routing a single result from the output of the functional units back to the register file. This configuration of the interconnects is standard in most conventional processor designs. Figure 3 presents the total execution time for the benchmarks for varying degrees of loop unrolling both for modest and deep pipelining.

Split Register File

Now consider a register file partitioned into distinct integer and floating point registers. If the input and output interconnects are similar to the case of a shared register file, i.e., a set of buses for the input operands of all the functional units and a single result bus for the result of all the functional units, a split register file is no different from a shared register file and, with no constraints on the size of either set of registers, the performance of the two architectural organizations will be the same.

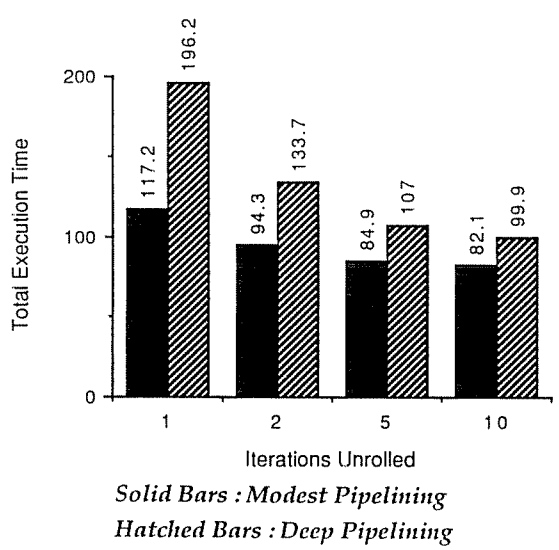


Figure 3: Total Execution Time (in thousands of clock cycles) with a Shared Register File and 1 Operation Per Instruction

However, partitioning the register file allows for a modification of the interconnect. The input interconnect can be partitioned into two input interconnects - one that connects the integer registers to the inputs of the integer functional units and another that connects the inputs of the floating point registers to the inputs of the floating point functional units (with appropriate connections to the memory port). Likewise, the output interconnect can be partitioned into separate interconnects for the integer and floating-point components. If the output interconnect is partitioned, an integer and floating-point operation can complete in the same clock cycle (even though only one operation is issued in a clock cycle). This flexibility allows for a slightly better performance than what could be achieved with a shared register file.

Figure 4 presents the results for a split register file with separate input and output interconnect for the integer and floating-point components. Comparing with Figure 3, we see that we can achieve slightly better performance by partitioning the interconnect mainly due to the scheduling flexibility provided by the additional data path. However, there is room for improvement both for split and shared register files (compare the results of Figures 3 and 4 with the results of Table 2).

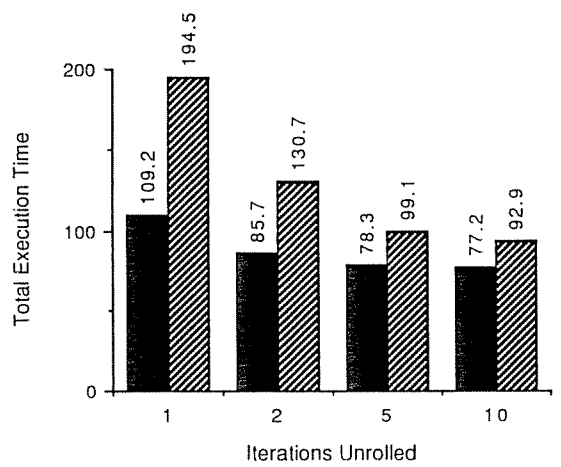


Figure 4: Total Execution Time (in thousands of clock cycles) with a Split Register File and 1 Operation Per Instruction

For the reader interested in comparing our results with the results presented in [4], the results of Figures 3 and 4 correspond to the "sequential ELI" (but with a different resource architecture), those of Table 1 correspond to an "ideal ELI" and those of Table 2 correspond to a "single cluster ELI". Our results follow trends similar to the results of [4]. In some cases, speedups greater than 20 over a simple pipelined execution[4] can be achieved with an enhanced level of fine grain parallelism and multiple operation issue (loop 8 is one such case) whereas in other cases the speedup is not so significant (loops 5 and 6 show this behavior).

[4]Note that a simple pipelined execution with limited fine grain parallelism already allows for some overlap. The results for a purely serial machine have not been presented because that would not provide a realistic lower bound for comparing horizontal architectures.

From Table 2 and Figures 3 and 4 we see that, for the resource architecture of Figure 1, speedups of 2.54 and 2.81 can be achieved for modest and deep pipelining, respectively, by making use of loop unrolling to enhance fine grain parallelism and by using pipelining and multiple operation issue to exploit this parallelism (going from rolled loops and the issue of one operation per instruction in Figure 3 to unrolled loops and the issue of 5 operations per instruction in Table 2). However, in the case of deep pipelining, most of this performance improvement can be obtained simply by enhancing the fine grain parallelism (loop unrolling) and by using pipelining to exploit it, without the need to issue more than one operation per instruction. This can be seen from Figure 3 where, with a loop unrolling of 10 and pipeline scheduling, a speedup of 1.96 can be achieved even with the issue of a single operation per instruction. This leaves only a factor of 1.44 to be gained by the issue of more than 1 operation per instruction. For modest pipelining where pipelining alone is not sufficient to exploit the available fine grain parallelism, multiple operation issue must also be used. Therefore, while the use of instruction formats capable of issuing more than one operation simultaneously will clearly improve the number of time steps required to execute a program, the need for such instruction formats is less compelling if the degree of pipelining in the machine is greater.

4.2. Two Operations Per Instruction

Now we consider a horizontal instruction format that allows two operations to be issued simultaneously.

Shared Register File

To allow two operations to issue per cycle with a shared register file, the input interconnect must be capable of delivering 4 operands from the register file to the functional units in every cycle. This requires the register file to have 4 read ports. Similarly, since 2 results can be generated in each cycle, the register file needs to have 2 write ports and the output interconnect needs to be able to deliver the results from the outputs of the functional units to the register file write ports. With this configuration, operations can be submitted to any two functional units with a single instruction.

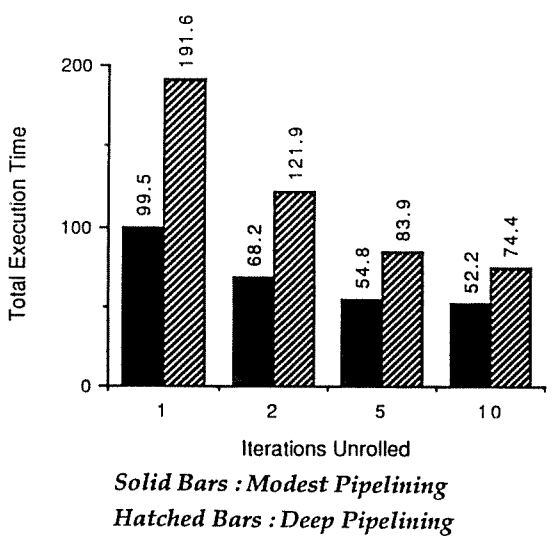


Figure 5: Total Execution Time (in thousands of clock cycles) with a Shared Register File and 2 Operations Per Instruction

Figures 5 and 6 present the total execution times and the average code densities for the benchmark programs for the cases of modest and deep pipelining. The code density is the average of the code densities of all the benchmarks (the code densities were not presented for the results of Figures 3 and 4 because the code density is 1 if there is only 1 operation per instruction). Since the code density is an indicator of a static phenomenon, namely the efficiency of a horizontal instruction format, the average has not been weighted by the number of iterations in the benchmarks.

By issuing two operations per cycle in a horizontal instruction, we can obtain a significant performance improvement especially if the level of pipelining is modest (compare the results of Figures 5 and 6 with the results of Figure 3). However, the code density suffers. This is because several horizontal instructions can not issue 2 operations. When two useful operations cannot be packed into a horizontal instruction, a NOP must be inserted. Notice that the code density improves as more loop unrolling is used. This is because by increasing the parallelism available (more loop unrolling) the same number of operations can be compacted into a fewer horizontal instructions.

Comparing the results of Figures 5 and 6 with the results of Table 2, we see that the *total* execution time that can be achieved by issuing 2 operations per instruction is quite close (a 13% difference for modest pipelining and a 6% difference for deep pipelining) to the best-case execution time that can be achieved by using a full-fledged instruction format, but the former requires significantly smaller amount of hardware to support it. We note again that for some individual loops (for example loops 3 and 8) the execution time could be improved by issuing more operations per instruction (see Table 3).

In Table 3, we show the code densities (CD) and per-iteration execution times (T) for each of the individual loops for a loop unrolling of 10 both for modest and deep pipelining. We notice that loops which could benefit from issuing more operations per instruction have comparatively high code densities for this restricted horizontal format. That is, the restricted format is being utilized well and a wider instruction format may be more appropriate. But loops with code densities near

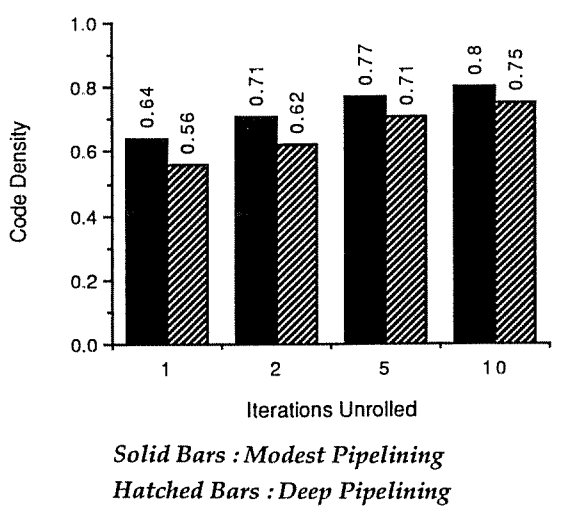


Figure 6: Average Code Density with a Shared Register File and 2 Operations Per Instruction

0.5 are able to achieve close to the best-case performance with this restricted instruction format. Note that code densities can not be less than 0.5 for this instruction format since that would imply the presence of instructions with no operations at all.

Table 3: Performance of the Individual Loops with a Shared Register File and Two Operations Per Instruction

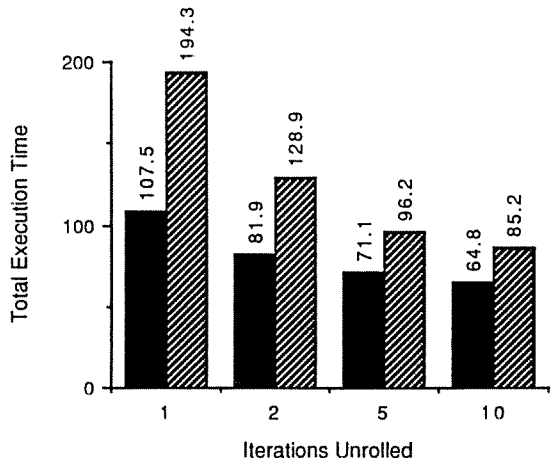
Loop	Modest		Deep	
	T	CD	T	CD
1	10.0	0.76	10.4	0.74
2	21.3	0.82	21.5	0.82
3	4.9	0.80	8.3	0.56
4	7.1	0.79	7.5	0.76
5	21.7	0.66	43.4	0.62
6	21.8	0.68	43.6	0.64
7	26.4	0.81	26.2	0.82
8	50.1	0.93	50.9	0.92
9	19.2	0.76	19.7	0.76
10	27.4	0.75	28.2	0.73
11	4.0	0.75	7.4	0.58
12	5.0	0.80	5.3	0.77
13	22.6	0.95	23.3	0.93
14	17.6	0.94	18.6	0.89

Split Register File

If the register file is partitioned, then we have a choice of two horizontal instruction formats to support the issue of more than one operation per instruction. The first format (free format) allows the issue of two operations to arbitrary functional units in a single instruction. As with a shared file, this would require the use of 4 read ports and 2 write ports in each of the register files. The second format (constrained format) is constrained to allow at most one integer and one floating point operation per instruction[5]. Because the constrained horizontal format places additional restrictions on the scheduler, programs compiled into a constrained format horizontal instruction set will have a greater execution time than programs compiled into a free format instruction set. But supporting the constrained horizontal instruction format poses fewer demands on the register files and the interconnects. To support the constrained horizontal format, each register file needs to have only 2 read ports and 1 write port and the input and output interconnects have the same complexity as in the case of the single operation instruction format of Section 4.1.

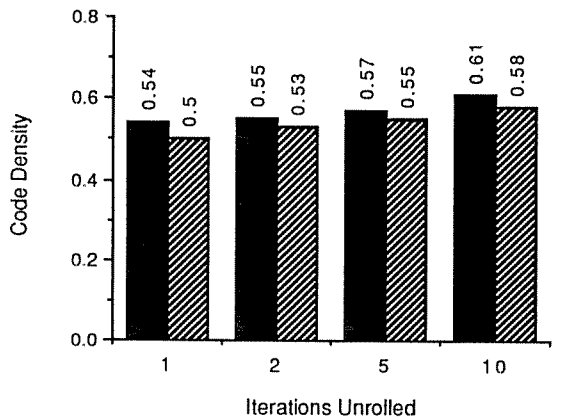
The total execution times and the average code densities for the benchmark programs using split register files and a constrained horizontal instruction format are presented in Figures 7 and 8. For these results, our scheduler was constrained to use a split register file with 2 read ports and 1 write port on either file. Comparing the results of Figures 7 and 8 with the results of Figure 4, we see that the performance improvement by issuing 1 integer and 1 floating-point operation per instruction in a constrained format is significant, though not spectacular (20.2% and 12.9%, respectively, for modest and deep pipelining). Furthermore, as expected, the constrained format is not able to accomplish the same performance as the free format

[5]Load and store operations are classified specially. Both need to access an integer register read port to obtain an address when they are issued. For a store, the other source operand accessed when it is issued could either be a floating point or an integer register. Likewise, loads would require a write port to the appropriate register file when they complete. Special case situations for loads and stores are automatically handled by our scheduler.



Solid Bars : Modest Pipelining
Hatched Bars : Deep Pipelining

Figure 7: Total Execution Time (in thousands of clock cycles) with a Split Register File and 2 Operations Per Instruction



Solid Bars : Modest Pipelining
Hatched Bars : Deep Pipelining

Figure 8: Average Code Density with a Split Register File and 2 Operations Per Instruction

(compare Figures 5 and 7) but it requires less hardware (fewer register ports and less complex interconnect) to support it. Also, the code density is worse with a split register file because of the additional constraints placed on the scheduler (compare Figures 6 and 8).

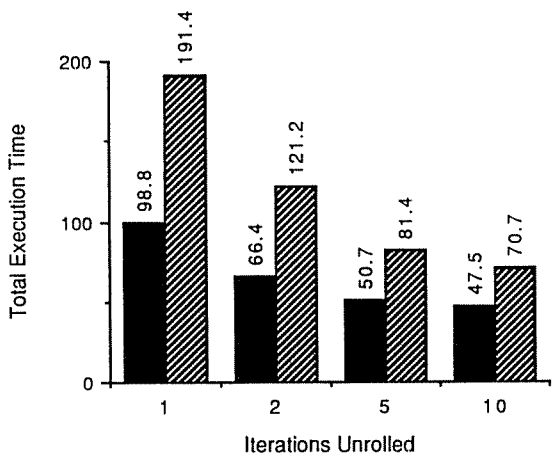
4.3. Three Operations Per Instruction

Continuing further, we now consider horizontal instruction formats capable of issuing 3 operations per instruction.

Shared Register File

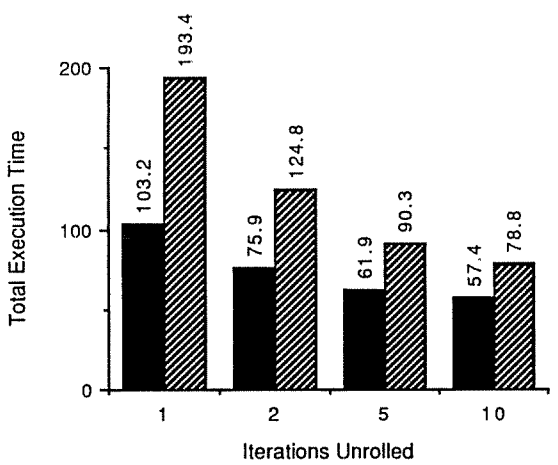
To support the issue of 3 operations per instruction cycle, the register file needs to have 6 read ports and 3 write ports. Also, the sizes of the input and output interconnects increase in proportion to the increased number of register file ports.

Figures 9 and 10 present the total execution times and the average code densities for the cases of modest and deep pipelining, respectively. As expected, the code density is lower than in the case of a 2-operation horizontal instruction format because of the large number of NOPs. Furthermore, little performance improvement over a 2-operation instruction format can be achieved since there is little room for improvement. For modest levels of pipelining and a loop unrolling of 10, the capability to issue 3 operations per instruction results in only a 9-10% improvement over the capability to issue 2 operations per instruction. For deep pipelining, this figure is about 5%. Therefore, if the hardware needed to provide the additional register ports and the additional interconnect degrades the clock cycle by as little as 5% in the deep pipelining case, the utility of the wider horizontal instruction format for the given functional unit resources (and benchmarks) is questionable.



Solid Bars : Modest Pipelining
Hatched Bars : Deep Pipelining

Figure 9: Total Execution Time (in thousands of clock cycles) with a Shared Register File and 3 Operations Per Instruction



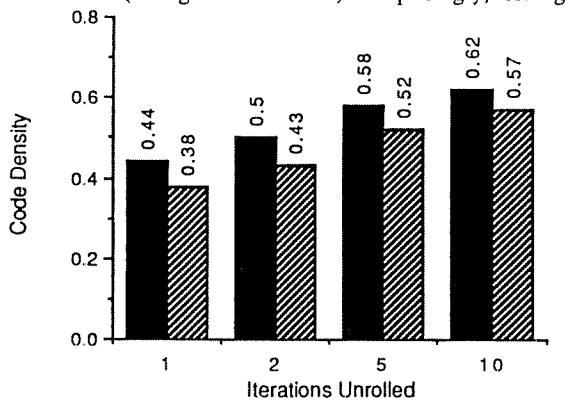
Solid Bars : Modest Pipelining
Hatched Bars : Deep Pipelining

Figure 11: Total Execution Time (in thousands of clock cycles) with a Split Register File and 3 Operations Per Instruction

Split Register File

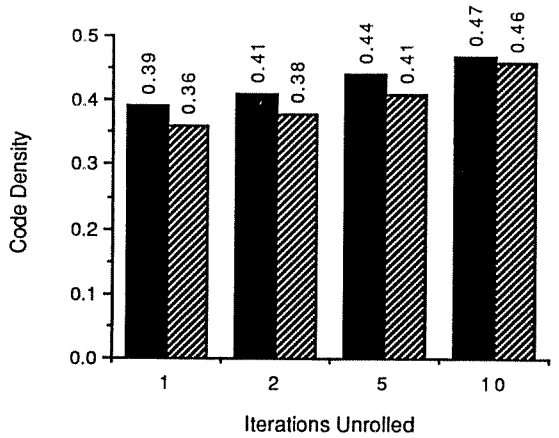
With a split register file, we can have three different horizontal instruction formats that can support the issue of 3 operations: a free format that allows the issue of up to 3 operations of either type, a constrained format that allows the issue of two floating point and one integer operation (CONS1) and a constrained format that allows the issue of two integer and one floating point operation (CONS2) per instruction. As before, loads and stores are classified accordingly.

We do not consider a free format for the same reasons as Section 4.2. If an application has more floating point operations than integer operations, the CONS1 format is more suitable. On the other hand, if the application has more integer operations than floating point operations, the CONS2 format is more suitable. We evaluated both restricted formats and both had a similar overall performance for our benchmark programs, though one was somewhat better than the other in individual cases. For brevity, we present only the results for the CONS1 format (in Figures 11 and 12). Surprisingly, issuing 3



Solid Bars : Modest Pipelining
Hatched Bars : Deep Pipelining

Figure 10: Average Code Density with a Shared Register File and 3 Operations Per Instruction



Solid Bars : Modest Pipelining
Hatched Bars : Deep Pipelining

Figure 12: Average Code Density with a Split Register File and 3 Operations Per Instruction

operations in a restricted format with a split register file has worse performance than a 2-operation instruction format with a shared register file even though the former requires more hardware. Remember, however, that the latter format file is able to support 2 operations of *either type* in a single instruction. This is not possible with the former instruction format.

The somewhat disappointing total execution time results of Figure 11 suggest that a shared register file may be preferable to a split register file if a restricted instruction format is to be used. For a full-fledged instruction format capable of issuing one operation to every functional unit, splitting the registers clearly saves on the number of register ports and reduces the complexity of the interconnects. However, if restricted instruction formats (and simpler interconnects) are to be used, splitting the register file may pose unnecessary constraints on the scheduling of operations, thereby resulting in worse performance as compared to a shared register file.

Since the performance of restricted instruction formats capable of issuing 2-3 operations per instruction is quite close to the performance of a full-fledged instruction format especially with a shared register file, we do not consider other restricted instruction formats, for example, formats capable of issuing 4 operations.

4.4. Comparative Results

Figures 13(a) and 13(b) summarize our results by comparing the performance of different horizontal instruction formats for our model architecture, both with modest and deep pipelining. Maximum performance (minimum number of clock cycles) is achieved when we can issue an operation to each functional unit in a single clock cycle with a wide instruction. Issuing only one operation per instruction with a shared register file results in minimum performance.

The figures also show the performance when issuing one operation per instruction with a split register file, and two and three operations per instruction with a shared register file. (The register file organizations chosen here are the ones which give the better performance for the corresponding instruction formats). We notice that for both modest and deep pipelining, issuing 3 operations per instruction results in very little additional speedup as compared to issuing 2 operations per instruction. Issuing 2 operations per instruction has a significant advantage over issuing just one operation per instruction if the degree of pipelining is modest and has a modest performance advantage if the degree of pipelining is high.

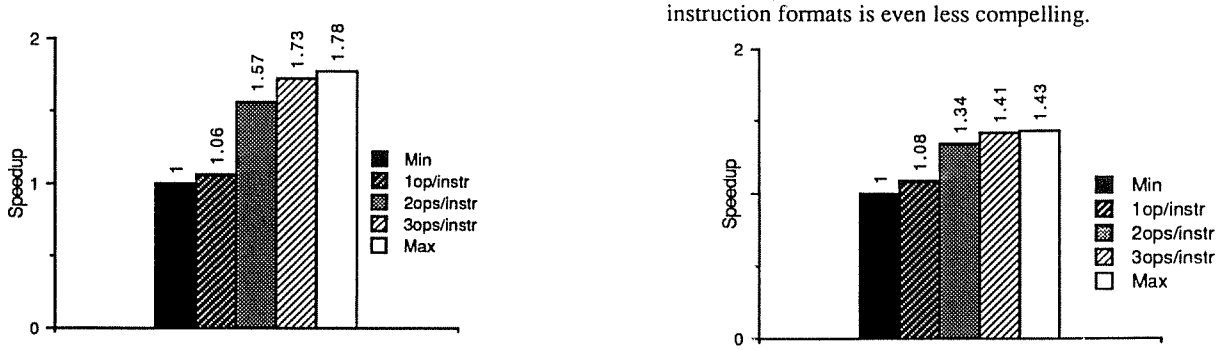
5. SUMMARY AND CONCLUDING REMARKS

The choice of an appropriate horizontal instruction format is crucial to the design of horizontal architectures. Horizontal architectures that have been proposed in the literature have full-fledged instruction formats capable of issuing an operation to each functional in the resource architecture unit with a single instruction. We felt that a horizontal architecture with a full-fledged instruction format (or to coin a term, a "complex horizontal architecture") is overkill since it is appropriate only for some programs and also requires an enormous amount of register file and interconnect hardware. Keeping this in mind, we studied horizontal architectures with restricted instruction formats (or "reduced horizontal architectures") and saw how this choice was influenced by factors such as the degree of pipelining and an instruction's view of the register file.

Our results suggest that, if the degree of pipelining is high, there is less need to issue more operations in parallel. Even with moderate levels of pipelining, restricted horizontal instruction formats capable of issuing operations only to a few functional units are competitive with full-fledged instruction formats capable of issuing one operation to each functional unit. While restricted instruction formats may perform worse than full-fledged formats on some programs, overall they are able to execute programs in a comparable number of time steps with much better code density and much less hardware.

We also considered the choice of a register file organization. A register file split into integer and floating point registers is clearly superior to a shared register file when a full-fledged horizontal instruction format is supported. However, for restricted horizontal instruction formats, a shared register file may be better and deserves attention.

An issue that we have ignored in this paper is the impact of the additional hardware needed to support a wider horizontal instruction format on the clock cycle. We assumed that an instruction is issued in a single clock cycle and that the clock cycle is the same for the various instruction formats, irrespective of the amount of hardware needed to support it. This is not true and machines with wider instruction formats typically have larger clock periods. As an example, compare the ZS-1 whose instruction formats can issue only one operation per instruction and the TRACE 7/200 whose instruction formats can issue several operations per instruction. Both machines are implemented in similar TTL technology but the clock period of the ZS-1 is 45ns and that of the TRACE 7/200 is 130ns. When the clock period is taken into account, the need to use wider instruction formats is even less compelling.



(a) Modest Pipelining

(b) Deep Pipelining

Figure 13: Comparative Speedups for various Horizontal Instruction Formats, using the Model Architecture of Figure 1 (10 Iterations Unrolled for each Loop)

In conclusion, horizontal architectures provide an excellent paradigm for exploiting fine-grain parallelism. However, a good architecture for exploiting fine-grain parallelism must make use of both pipelining and parallel operation issue. Unless the computer architect pays adequate attention to pipelining also, the architect is bound to get caught up in a vicious circle. To issue more operations simultaneously, additional register file and interconnect hardware must be provided. This additional hardware will penalize the clock cycle which in turn will reduce the number of pipeline stages in the functional units. With a reduced number of pipeline stages, more operations must be issued in parallel to exploit the available parallelism. A good horizontal architecture design must attempt to use both pipelining and parallel operation issue and must provide adequate instruction formats that enhance the ability of the architecture to exploit both pipelining and multiple operation issue without penalizing either.

Acknowledgments

This work has benefited from discussions with Jim Goodman and Jim Smith and was supported in part by NSF Grant CCR-8706722.

References

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, pp. 8-24, January 1967.

[2] A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *Computer*, vol. 14, September 1981.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, vol. 37, pp. 967-979, August 1988.

[4] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures," Research Report YALE/DCS/RR-364, Department of Computer Science, Yale University, Seattle, WA 98195, February 1985.

[5] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 140-150, June 1983.

[6] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, pp. 1901-1909, December 1966.

[7] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *Proc. ASPLOS III*, Boston, MA, April 1989.

[8] F. H. McMahon, *FORTRAN CPU Performance Analysis*. Lawrence Livermore Laboratories, 1972.

[9] A. Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Work Architectures," *IEEE Transactions on Computers*, vol. C-33, pp. 968-976, November 1984.

[10] Y. N. Patt, W.-M. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," in *Proc. 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, pp. 103-108, December 1985.

[11] B. R. Rau, C. D. Glaeser, and R. L. Picard, "Efficient Code Generation For Horizontal Architectures: Compiler Techniques and Architectural Support," *Proc. 9th Annual Symposium on Computer Architecture*, pp. 131-139, April 1982.

[12] B. R. Rau, "Cydra 5 Directed Dataflow Architecture," *Digest of Papers, COMPCON Spring 1988*, pp. 106-113, February 1988.

[13] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, pp. 1405-1411, December 1972.

[14] R. M. Russel, "The CRAY-1 Computer System," *CACM*, vol. 21, pp. 63-72, January 1978.

[15] J. E. Smith, et al, "The ZS-1 Central Processor," *Proc. ASPLOS II*, pp. 199-204, October 1987.

[16] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, vol. C-19, pp. 889-895, October 1970.

[17] W. A. Wulf, "Compilers and Computer Architecture," *IEEE Computer*, vol. 14, pp. 41-47, July 1981.