Hierarchical Registers for Scientific Computers

John A. Swensen

Yale N. Patt

University of California at Berkeley

Abstract

Simulations of scientific programs running on traditional scientific computer architectures show that execution with hundreds of registers can be more than twice as fast as execution with only eight registers. In addition, execution with a small number of fast registers and hundreds of slower registers can be as fast as execution with hundreds of fast registers. A hierarchical organization of fast and slow registers is presented, register-allocation strategies are discussed, and a novel, indirect, register-addressing mechanism is described.

1. Introduction

Early scalar, scientific computers had relatively few registers for temporary results. For example, the CDC 7600 [1] had only eight floating-point registers. More recent scientific computers have tended to have many more registers. The ETA-10 [6,7] has 256 general-purpose registers, the HEP-1 [5] had several thousand general-purpose registers, and the Cray-1 and its descendants [2-4] have 512 vector-register elements in addition to several scalar registers. One is compelled to ask if the availability of these large register sets improves performance, or whether they have been included simply to provide programming convenience?

It is well known that the access time for a large register set is greater than the access time for a smaller register set. Kuck [8] showed, for example, that worst-case fan-in grows as $log(k)$, where k is the number of data elements multiplexed. Under what circumstances then do the large sets of registers, with their longer access times, enhance performance?

This paper attempts to deal with these two questions. Execution speed of a set of programs is simulated with varying numbers of registers, varying speeds of registers, and varying degrees of parallel execution. Analyses of these simulations reveal the register requirements for fast execution of various kinds of programs. A hierarchical register organization that provides the necessary register characteristics is then presented.

In section 2 the set of programs and the simulator are described, the analyses of the simulation results are presented, and the registers requirements are discussed. In section 3 the hierarchical register organization is presented. In section 4 several issues, including the issues of register allocation and addressing are discussed. In section 5 we offer a few concluding remarks.

2. Execution Simulation

2.1. Simulated Code

Livermore Kernels 1-14 [9] are chosen as program fragments representative of scientific programs. Although they do not represent all important scientific programs, they are attractive because of their conciseness and widespread availability. The 1984 versions of the kernels that are used for the simulations are listed in Appendix A.

Programs that are simulated are specified by their dependency graphs. This eliminates anomalies introduced by specific programming languages and by specific compilations for specific machines. Each node in a dependency graph represents a single operation such as a multiply, a subtract, or a memory store, and a directed edge between two nodes represents a result produced by one node and consumed by the other. Programs are translated to dependency graphs by hand, with no exotic parallelizing techniques, such as factoring of expressions, performed on programs or dependency graphs.

2.2. Simulator Description

All simulations assume a load-store architecture with multiple, overlapped, pipelined functional units and interleaved memory. Operation-execution times and, hence, pipeline lengths are taken from the Cray-1S [2]; for exam-

ple, logical operations execute in one clock tick, floating-point multiplies execute in seven ticks, and memory loads execute in eleven ticks (results of simulations using other reasonable operation-execution times would not differ substantially from the simulation results reported below). Each operation's execution time is the sum of operand read time, pipeline latency, and result write time. For architectures with few registers the sum of the register-operand read time and the register-result write time is assumed to be less than one clock tick.

The time required to load and decode an instruction is not included in the execution time of the corresponding operation(s), because for scientific programs it can usually be overlapped with the execution of other operations. When it can not be overlapped (for example, following a conditional branch) the instruction overhead is included in the branch latency.

The time required to check hardware and result reservations is ignored in this paper, because for the programs simulated, all reservation checking can be performed at compile time. However, this is not true in general, and we are currently investigating this issue.

Parameters of the simulator that are varied are the number of registers available for temporary results, the speeds of the registers, and the number of operations that can start each tick. The number of registers available ranges from four to an unlimited number. An operation with no result register must either wait for one before it starts, or it must spill its result to main memory, increasing its execution time. An operation using a spilled result must first load it from memory, adding a memory load time to its execution time. Fast registers can be read or written in less than one tick, so that a logical operation can execute in one tick. Slow registers can be written in one clock tick, and can be read in either two, three, or five ticks; the use of slow registers increases an operation's execution time. A maximum of either one, two, four, or eight operations can start each clock tick; a start-limit of one represents traditional, scalar execution, while a start-limit of four represents either the overlapped execution of four vector instructions or the execution of complex instructions specifying four operations each.

The simulator maintains a dependency graph, a ready list of operations with all of their operands available, an active list of operations being executed, and counts of available registers. Initially the ready list contains operations that depend on no other operations, the active list is empty, and the register counts are the maximum number of available registers of each speed.

Each tick the following events occur:

(1) The remaining execution time of each operation in the active list is decremented, and each operation with no remaining time is removed from the active list and its result is made available to any operation that uses it.

(2) Any operations in the dependency graph that now have all their operands available are added to the ready list.

(3) Up to start-limit operations are moved from the ready list to the active list, result registers are allocated to them (if their results do not spill to memory), and the count of available registers is decreased. When an unlimited number of registers is available, operations in critical path (the set of longest paths in the directed, acyclic dependency graph) are selected from the ready list before operations that not in the critical path, but no attempts are made to schedule more cleverly. Nevertheless, execution times are usually less than 3% longer than optimal, and are never more than 26% longer than optimal [10]. Therefore, better schedulers are unlikely to significantly change the results of these simulations. When available registers are limited, operations that produce results to be used sooner are selected before operations that produce results to be used later. This heuristic conserves register usage, but sometimes results in longer execution times than the other heuristic. When both fast and slower registers are available, operations in the critical path are allocated fast registers if doing so could speed up execution.

(4) If an operation added to the active list is the last to read a previous operation's result, that result's register is freed and the available-register count is incremented.

The number of clock ticks from the time the first operation is added to the active list until the last operation is removed from the active list is the total execution time.

2.3. Dependence of Performance on Number of Registers

Figures 1a–1n show curves for the ratios of simulated execution times with n registers to the execution time with an unlimited number of registers, for Livermore Kernels 1–14, for n equal to 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 2048, and for start-limits of one, two, four, and eight. The results do not vary significantly when the register-allocation heuristics are varied.

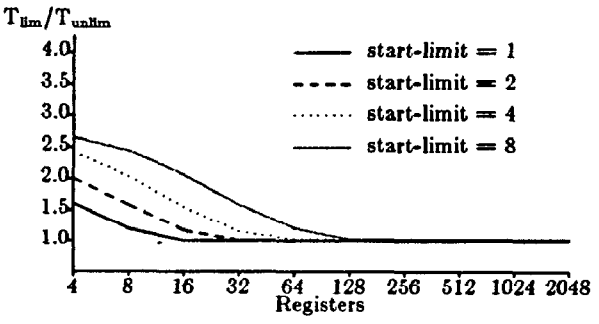

Figure 1a: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 1.

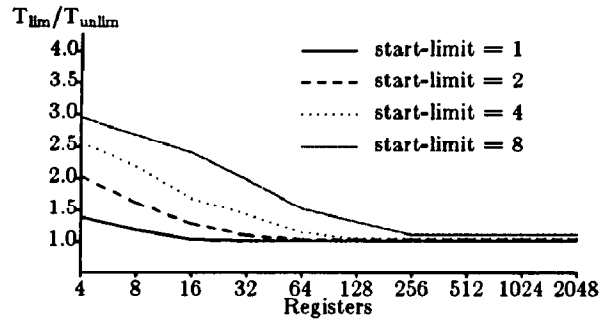

Figure 1b: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 2.

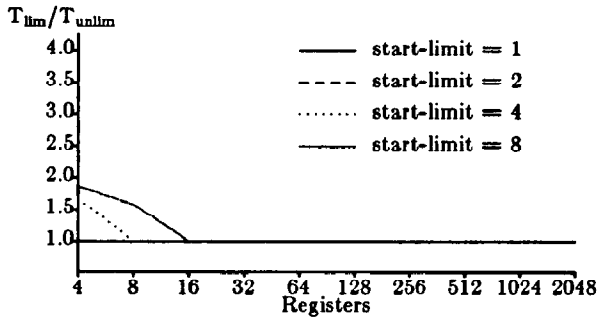

Figure 1c: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 3.

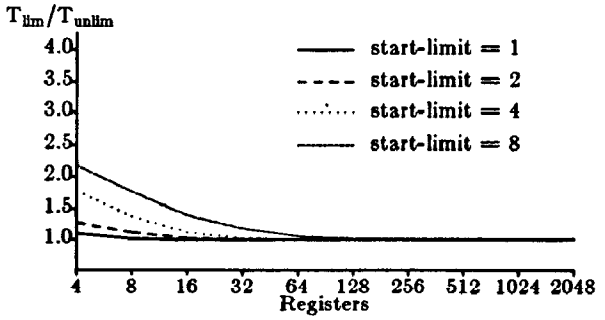

Figure 1d: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 4.

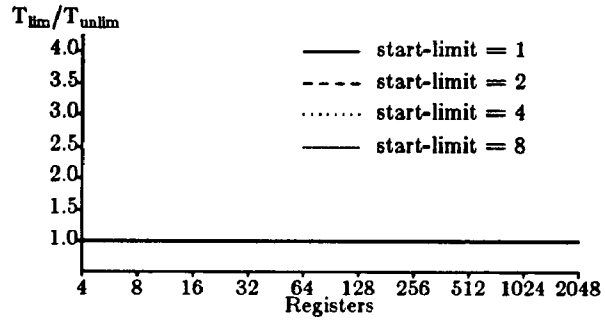

Figure 1e: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 5.

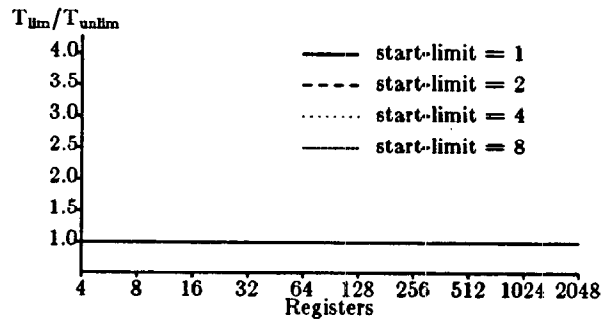

Figure 1f: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 6.

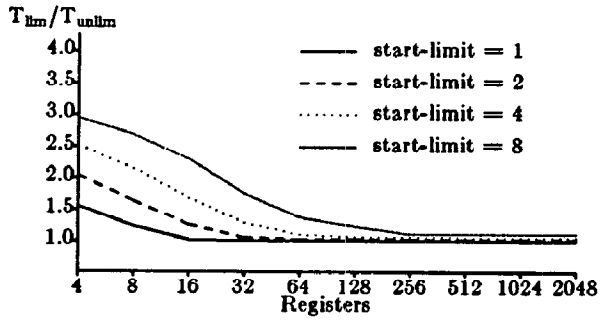

Figure 1g: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 7.

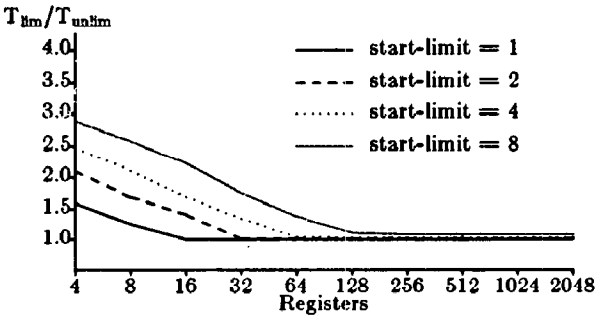

Figure 1h: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 8.

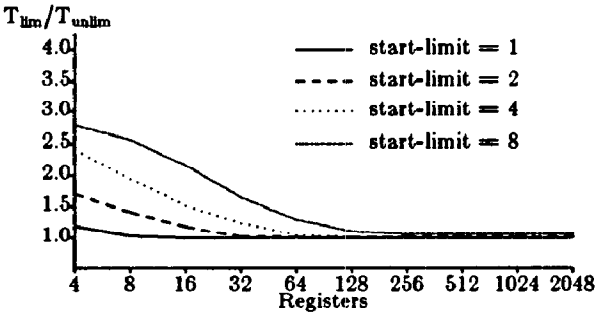

Figure 1i: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 9.

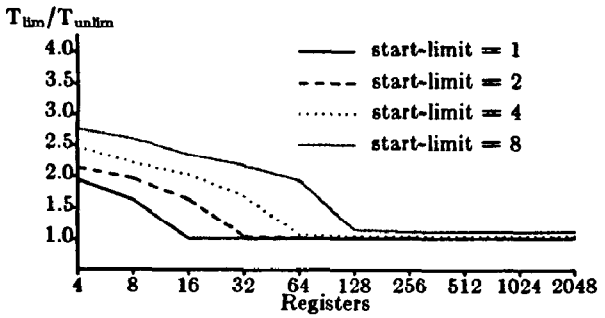

Figure 1j: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 10.

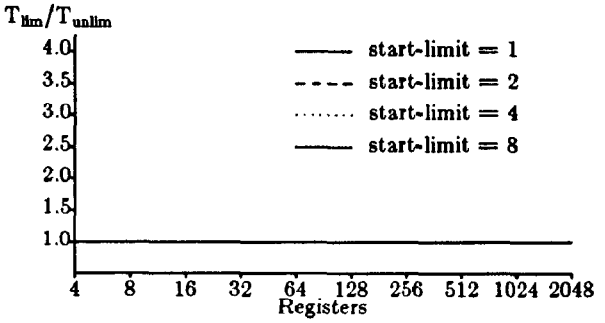

Figure 1k: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 11.

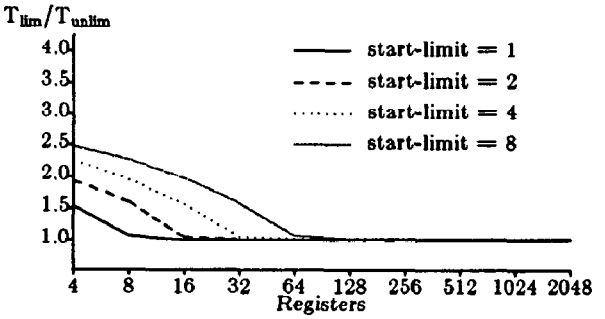

Figure 1l: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 12.

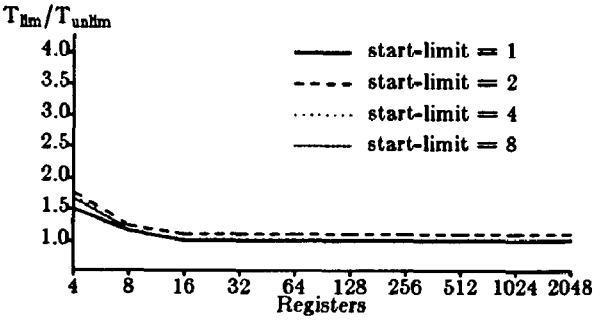

Figure 1m: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 13.

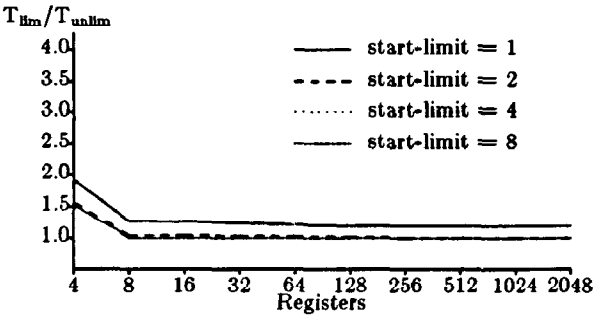

Figure 1n: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 14.

The very-serial kernels 5, 6, and 11 run as fast with four registers as they do with an unlimited number of registers. The narrow critical-path widths of these kernels ensure that few operations are executing concurrently, and that most results are used immediately after they are generated.

The moderately-serial kernel 13 requires approximately sixteen registers to run at close to its ultimate speed, but the execution speed with the register-conserving schedule never reaches the speed of the kernel with an execution-time-conserving schedule that happens to require more registers.

The moderately-serial kernel 14 requires approximately 128 registers to reach its ultimate speed for a start-limit of one, but this is 30% slower than the execution speed of the kernel with a minimal-execution-time schedule. Kernels 13 and 14 both have narrow critical paths, but they also have many operations that can execute concurrently. Schedules that do not attempt to conserve registers allow the non-critical-path operations to start long before their results are needed. This ties up registers until the results are used, but it also ensure that most of the critical-path operations run without interference from non-critical-path operations.

The parallel kernels require between eight and 256 registers in order to run at their ultimate speeds, depending on the start-limits. With only eight registers, most of the parallel kernels run 2–2.5 times longer than with an unlimited number of registers. Most of the parallel kernels have wide critical paths, and their execution times are often limited by the time required to start all the operations. Thus, many operations are executing at a time, and each operation is allocated a result register. Also, operations do not always start as soon as data dependencies allow because of the limited start-bandwidth, and result registers cannot be freed until the operations that use the results start.

The fact that programs scheduled to use few registers run slower than programs scheduled without regard to register usage is a motivation for machines with many registers. If the machine has a sufficient number of registers, scheduling time can be spent increasing execution speed, rather than minimizing register usage.

It should be noted that the Cray-1 has a total of 512 vector register elements; therefore much vectorizable code could potentially execute as fast as it could with an unlimited number of registers. However, unvectorized loops like kernel 14 could not make effective use of the vector registers, and therefore, more registers or a more general register structure than the Cray-1 has are needed for the fastest execution of Livermore Kernels 1–14.

When the number of registers is restricted to four, execution times of most kernels are increased by a factor of 1.5 to 3. The execution times of the kernels would be even greater with fewer registers, suggesting that pure, memory-to-memory, scalar architectures are terribly inefficient, at least for programs with characteristics similar to Livermore Kernels 1–14.

2.4. Fast-Register Requirements

During serial sections of programs, when few operations are ready to execute, the operation-execution times dominate the total execution time, so register access should be as fast as possible. During parallel sections of programs, however, longer register access times can be tolerated by overlapping the execution of more operations.

The maximum number of fast registers reserved at a time is tracked for execution with start-limits of one, two, four, and eight, for all 14 kernels. A count of the number of times fast registers are reserved is also maintained, as an indication of the importance of fast registers to fast execution of the program. For example, if fast registers are allocated 2000 times even though only two fast registers are required, the fast registers speed up at least 2000 critical-path operations, and, thus, have a significant effect on performance. If fast registers are allocated only ten times, they only speed up the computation rarely, and they have a negligible effect on performance.

These data are shown for registers with two-tick reads and one-tick writes in table 1.

Kernels 5, 6, 11, 13, and 14, with very narrow critical paths, can make use of no more than one or two fast registers, although they use them for the result of almost every critical-path operation. One or two fast registers speed up the execution of these kernels significantly.

Kernels 3 and 4 rarely use fast registers, and kernel 12 never uses a fast register. Kernels 3 and 4 essentially execute summation trees, and virtually all operations are in the critical path, so there can be a shortage of critical-path operations only at the bottom of their summation trees. Every operation in kernel 12 is in the critical path, which is many times wider than the largest start-limit of eight. Kernels 3, 4, and 12 do not need fast registers for fast execution.

Kernel 2 can use as many as 40 or more fast registers if its loop is executed enough times and if start-limit is large enough. The reason for this behavior is that kernel 2 computes the inner products of sub-vectors of length five, and the unbalanced summation tree causes some results to wait longer before they are used. Better scheduling of the operations in this kernel would eliminate the anomalous behavior.

Table 1: Number of Fast Registers Required and Number of Times Fast Registers Used (Slow-Register Read-Time = 2, Slow-Register Write-Time = 1).

	Number Parallel Starts							
	1		2		4		8	
Kernel	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used
1	1	2	2	3	4	5	8	9
2	1	1	2	2	4	4	40	68
3	3	9	3	9	3	9	3	9
4	4	11	3	11	3	11	3	11
5	1	1993	1	1992	1	1992	1	1992
6	1	1995	1	1994	1	1994	1	1994
7	1	2	2	3	4	5	8	9
8	1	2	2	3	4	5	8	9
9	1	1	2	2	4	4	8	8
10	1	8	2	16	4	32	8	64
11	1	1000	1	999	1	999	1	999
12	0	0	0	0	0	0	0	0
13	2	364	2	391	2	391	2	391
14	2	606	2	606	2	607	2	607

These results summarized in table 1 show that only a few fast registers can be effectively used for temporary-result storage.

The ratios of execution times of all kernels with mostly slow registers to the execution times with all fast registers are shown in table 2. The number of fast registers are the same as the maximum required in table 1, the slow-register read time is two ticks, and the slow-register

Table 2: Ratios of Times for Simulated Execution with Few Fast Registers and Unlimited Slow Registers to Times with Unlimited Fast Registers, for 2-Tick Slow Read, 1-Tick Slow Write.

	Number Parallel Starts			
	1	2	4	8
1	1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.02
3	1.00	1.00	1.00	1.01
4	1.00	1.00	1.01	1.03
5	1.00	1.00	1.00	1.00
6	1.00	1.00	1.00	1.00
7	1.00	1.00	1.00	1.00
8	1.00	1.00	1.00	1.00
9	1.00	1.00	1.00	1.00
10	1.00	1.00	1.00	1.00
11	1.00	1.00	1.00	1.00
12	1.00	1.00	1.00	1.00
13	1.00	1.00	1.00	1.00
14	1.00	1.00	1.00	1.00

write time is one tick. Almost without exception, the programs run no slower with mostly slow registers than with all fast registers. In the worst case, kernel 4 with a start-limit of eight requires 3% more time to execute with mostly slow registers than with all fast registers.

The results of simulations when the slow read time is increased to three ticks are summarized in table 3. There is a worst case degradation of 5% for kernel 4 and a worst case degradation of 12% for kernel 14. For most of the kernels most of the time, however, the execution times and fast register usages are the same as when the slow read time is two ticks.

Table 3: Ratios of Times for Simulated Execution with Few Fast Registers and Unlimited Slow Registers to Times with Unlimited Fast Registers, for Slow-Register Read-Time = 3, Slow-Register Write-Time = 1.

	Number Parallel Starts			
	1	2	4	8
1	1.00	1.00	1.00	1.00
2	1.00	1.00	1.01	1.03
3	1.00	1.00	1.01	1.02
4	1.00	1.01	1.02	1.05
5	1.00	1.00	1.00	1.00
6	1.00	1.00	1.00	1.00
7	1.00	1.00	1.00	1.01
8	1.00	1.00	1.00	1.01
9	1.00	1.00	1.00	1.01
10	1.00	1.00	1.00	1.01
11	1.00	1.00	1.00	1.00
12	1.00	1.00	1.00	1.01
13	1.00	1.00	1.00	1.00
14	1.04	1.12	1.12	1.12

If the slow-register read time is increased to five ticks, the execution times are as much as 20% greater than execution times for all fast registers, so five-tick slow registers are not nearly as useful as faster registers.

These results suggest that three different register speeds could be useful: a small set of less-than-one-tick registers for the most critical operations, a larger set of two-tick registers for moderately critical operations, and an even larger set of three-tick registers for the non-critical operations.

The results summarized in figures 1a–1n and in tables 1–3 suggest that a small set of high speed registers supplementing a large set of slower registers provide the same performance as an unlimited set of high speed registers. This performance is about a factor of two faster than if only eight registers are provided.

3. A Hierarchical, General-Purpose Register Organization

Consider the design of a set of 1024 general-purpose registers, a subset of which can be accessed in less than one clock tick. Note that accessing one of many registers requires more time than accessing one of a few registers, because the many registers require more levels of logic to multiplex than do the few registers, and, therefore, increasing the number of registers increases the worst-case access time.

The organization of one bit of a hierarchical set of registers is shown in figure 2. This register set can support up to two register reads and one register write each clock tick. At the left of figure 2 are 960 registers, organized as 30 sets of 32 registers each. A collection of 30 pairs of 32:1 multiplexers select the contents of any two registers. The multiplexer outputs are held in 30 pairs of latches controlled by the system clock. In the middle of the figure, 59 additional registers and the 30 pairs of latches feed a pair of 89:1 multiplexers, the outputs of which are held in a pair of latches controlled by the system clock. At the right of figure 2, five additional registers plus the pair of latches plus f other functional unit outputs are selected by a pair of $(7+f)$:1 multiplexers. The outputs of these last two multiplexers feed a logical functional unit that performs operations like AND, OR, exclusive-OR, etc. The logical functional unit output feeds back to the five close registers, and is also held in a latch controlled by the system clock. The output of this latch is fanned out to the 59 middle registers and to the 960 distant registers. In addition, a pair of 7:1 multiplexers can send the contents of any of the registers to the other functional units in the CPU.

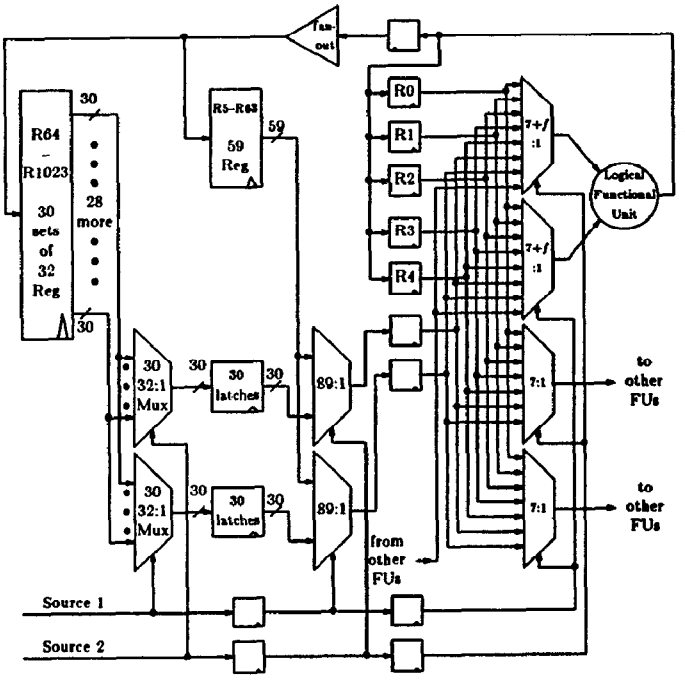

Figure 2: Organization of a One-Bit Slice of a Hierarchical Register Set.

Distant registers can be read and sent to the logical functional unit in two clock ticks, middle registers can be read and sent to the logical functional unit in one clock tick, and close registers can be read and sent to the logical functional unit in less than one clock tick. Read addresses reach the distant multiplexers directly from the operation start hardware, they reach the middle multiplexers after a one-tick delay, and they reach the close multiplexers after a two-tick delay.

The output of the logical functional unit can be written to any close register in the same clock tick and to the middle and distant registers in the next tick. Decoded write addresses reach the close registers at the same time that the read addresses reach the close multiplexers. Decoded write addresses reach the middle and distant registers one tick after read addresses reach the close multiplexers.

In order to support a start-limit of eight, this hierarchical-register organization must be generalized to support up to 16 register-reads and eight register-writes each clock tick. This generalization and other implementation details are discussed by Swensen [10].

4. Discussion

4.1. Hierarchical-Register Allocation

Close, middle, and distant registers should be allocated such that close registers are used for the most time-critical temporary results, middle registers are used for less time-critical temporary results, and distant registers are used for even less time-critical temporary results.

As long as an operation is separated from its source operations by at least

start-limit * (operation-time + write-time + read-time)

other operations, it can start without any wait. (Note that even if data forwarding is used, the result must be written into some latch, and the destination operation must select from some number of source latches and registers, so a write time and a read time must still be included in the inter-operation separation.) Operations are allocated distant result-registers, and are scheduled so that they are separated from their source operations where possible. Based upon the studies summarized in section 2, with 512 or more distant registers there should always be enough distant registers for all temporary results. For pairs of operations which cannot be scheduled far enough apart, the available middle registers are allocated to the more time-critical temporary results, and the available fast registers are allocated to the most time-critical temporary results.

The use of registers with different speeds undoubtedly complicates the task of automatic register allocation. However, the use of a large number of registers simplifies some aspects of register allocation, because there should always be enough registers available for temporary results. Therefore, the net effect may be only a slight increase in compiler complexity.

4.2. Addressing Many Registers

With 1024 registers, three-address instructions require 30 bits to specify all three register operands explicitly. Relative to a shorter instruction format, more instruction-stream bandwidth and a larger instruction buffer are required to achieve the same instruction-issue performance.

It is possible to address the 1024 registers by specifying one register operand explicitly using a ten-bit address, and by specifying the remaining register or registers indirectly, using fewer bits. Each time an instruction with a register destination issues, the ten-bit, register address is added to a queue of register addresses. Register source specifiers are three bits long and they select which of the last eight registers written is to be used as the operand. If both source registers are not among the last eight written, a 30-bit, long-format instruction is used.

The sequence of written registers can be determined at compile time, so the correct specification can be determined then, as well. In addition, each register destination can be determined before the instruction actually issues, so the queue can be updated enough in advance that instructions can issue without delays. With only eight queue elements, the addresses can be selected quickly. Furthermore, results tend to be used soon after they are written, as discussed by Swensen [10], so very often the desired result is one of the last eight written.

4.3. Other Issues

Computers with many registers have long context-switch times. This is not a major disadvantage for scientific computers, however, because time-critical terminal and disk input and output operations are typically serviced by other computers dedicated to the tasks. In a time-sharing environment, the minimum running time quantum can be increased to the point where context-switch time is insignificant relative to running time. For procedure and function calls, most of the registers are not saved; the large number of registers allows different subsets of registers to be allocated to different procedures and functions.

In order to read distant-register operands in time, instructions with operation specifications must reach the start hardware at least two ticks before operations are actually started. This increases the delay following a conditional branch before operations start again. This also forces any hardware reservation mechanisms to check the availability of registers as much as two ticks in the future, or longer if reservation checking takes more than one tick. With many close/distant registers there is, thus, a strong motivation to perform as much reservation checking as possible at compile time.

5. Conclusions

Execution of parallel sections of scientific programs on an architecture with 256 or more registers can be more than twice as fast as execution on a similar architecture with only eight registers. This is because the availability of many registers allow operations to be scheduled for maximum overlap of execution, rather than for minimum usage of registers. The greater the overlap of operation execution (either because of longer execution times or because more operations are started each tick), the more registers that are needed to hold the temporary results. The longer access times of large register sets does not significantly increase the execution time of parallel programs because a register's access can be overlapped with the execution of other operations.

Execution of serial sections of scientific programs is not any faster with many registers than it is with a few registers, because the serial dependencies prevent most of the overap of operation execution. Longer register-access times degrade performance because the operand accesses can not be overlapped with the execution of other operations.

Fortunately, programs do not require many registers at the same time that they require fast registers, so it is possible to use a large set of registers, a few of which are fast, and most of which are slow. A hierarchical organization of registers that provides direct access to a small set of registers and pipelined access to the remaining registers can support fast execution of both serial and parallel sections of scientific programs. Register allocation, while more complex than for a uniform set of registers, is straightforward. The instruction-stream bandwidth requirements for the large set of registers can be reduced by using limited, indirect addressing of the registers.

For architectures that are oriented towards vector processing, vector registers probably provide the necessary temporary storage more efficiently than hierarchical, general-purpose registers. In fact, vector registers can even be generalized to support fast serial execution [10]. However, many parallel programs do not vectorize easily, so that an architecture with a large, hierarchical, general-purpose register set may support faster execution than a vector architecture.

Acknowledgements

This work was supported in part by Lawrence Livermore National Laboratories, LLNL Contract 4695505. We gratefully acknowledge the continued enthusiastic support of the scientists at Livermore, in particular George Michael and John Ranelletti.

References

1. Control Data Corporation, *Control Data 7600 Computer System: Preliminary Reference Manual,* Control Data Corporation, Minneapolis (circa 1970).
2. Cray Research, Inc., "Cray-1 Computer Systems: Cray-1 S Series Hardware Reference Manual," HR-0808, Mendota Heights, Minn. (1981).
3. Cray Research, Inc., "Cray Computer Systems: Cray X-MP Model 48 Mainframe Reference Manual," HR-0097, Mendota Heights, Minn. (1984).
4. Cray Research, Inc., "Cray Computer Systems: Cray-2 Hardware Reference Manual," HR-2000, Mendota Heights, Minn. (1985).
5. Denelcor, Inc., "HEP Hardware Reference Manual," 9000003, Denver (1982).
6. ETA Systems, Inc., "Mainframe Subsystem Equipment Specification," 003106, St. Paul, Minn. (February 27, 1987).
7. ETA Systems, Inc., "Mainframe Subsystem Instruction Specification for the ETA^{10}," 000211, St. Paul, Minn. (June 4, 1987).
8. D.J. Kuck, *The Structure of Computers and Computations: Volume One,* Wiley, New York (1978).
9. F.H. McMahon, "LLNL FORTRANS KERNELS: MFLOPS," Lawrence Livermore National Laboratory (March 1984).
10. J.A. Swensen, "High-Bandwidth/Low-Latency Temporary Storage for Supercomputers," PhD Dissertation: University of California at Berkeley, Report No. UCB/CSD 87/383, University of California at Berkeley, Berkeley, California (December 1987).

APPENDIX A

Livermore Kernel 1: Hydro Excerpt

```
for k = 1 to 400
  x[k] = q+y[k]*(r*z[k+10]+t*z[k+11])
```

Livermore Kernel 2: MLR, Inner Product

```
for k = 1 to 40*5 by 5
  tp[k] = z[k]*x[k]+z[k+1]*x[k+1]+z[k+2]*x[k+2]
    + z[k+3]*x[k+3]+z[k+4]*x[k+4]
```

Livermore Kernel 3: Inner Product

```
for k = 1 to 1024
  q = q+z[k]*x[k]
```

Livermore Kernel 4: Banded Linear Equations

```
for l = 7 to 107 by 50
  for j = 1 to 128
    x[l-1] = x[l-1]-x[l+j-1]*y[j]
```

Livermore Kernel 5: Tri-Diagonal Elimination, Below Diagonal

```
for i = 2 to 997
  x[i] = z[i]*(y[i]-x[i-1])
```

Livermore Kernel 6: Tri-Diagonal Elimination, Above Diagonal

```
for i = 997-1 downto 1
  x[i] = x[i]-z[i]*x[i+1]
```

Livermore Kernel 7: Equation of State Excerpt

```
for m = 1 to 120
  x[m] = u[m]+r*(z[m]+r*y[m])
    + t*(u[m+3]+r*(u[m+2]+r*u[m+1])
    + t*(u[m+6]+r*(u[m+5]+r*u[m+4])))
```

Livermore Kernel 8: PDE Integration

```
for kx = 2 to 3
  for ky = 2 to 20
    du1[ky] = u1[kx,ky+1,nl1]-u1[kx,ky-1,nl1]
    du2[ky] = u2[kx,ky+1,nl1]-u2[kx,ky-1,nl1]
    du3[ky] = u3[kx,ky+1,nl1]-u3[kx,ky-1,nl1]
    u1[kx,ky,nl2] = u1[kx,ky,nl1]
      + a11*du1[ky]+a12*du2[ky]+a13*du3[ky]
      + sig*(u1[kx+1,ky,nl1]-2.*u1[kx,ky,nl1]
      +u1[kx-1,ky,nl1])
    u2[kx,ky,nl2] = u2[kx,ky,nl1]
      + a21*du1[ky]+a22*du2[ky]+a23*du3[ky]
      + sig*(u2[kx+1,ky,nl1]-2.*u2[kx,ky,nl1]
      +u2[kx-1,ky,nl1])
    u3[kx,ky,nl2] = u3[kx,ky,nl1]
      + a31*du1[ky]+a32*du2[ky]+a33*du3[ky]
      + sig*(u3[kx+1,ky,nl1]-2.*u3[kx,ky,nl1]
      +u3[kx-1,ky,nl1])
```

Livermore Kernel 9: Integrate Predictors

```
for i = 1 to 100
  px[1,i] = bm28*px[13,i]+bm27*px[12,i]+bm26*px[11,i]
    + bm25*px[10,i]+bm24*px[9,i]+bm23*px[8,i]
    + bm22*px[7,i]+c0*(px[5,i]+px[6,i])+px[3,i]
```

Livermore Kernel 10: Difference Predictors

```
for i = 1 to 100
  ar = cx[5,i]
  br = ar-px[5,i]
  px[5,i] = ar
  cr = br-px[6,i]
  px[6,i] = br
  ar = cr-px[7,i]
  px[7,i] = cr
  br = ar-px[8,i]
  px[8,i] = ar
  cr = br-px[9,i]
  px[9,i] = br
  ar = cr-px[10,i]
  px[10,i] = cr
  br = ar-px[11,i]
  px[11,i] = ar
  cr = br-px[12,i]
  px[12,i] = br
  px[14,i] = cr-px[13,i]
  px[13,i] = cr
```

Livermore Kernel 11: First Sum

```
for k = 2 to 999
  x[k] = x[k-1]+y[k]
```

Livermore Kernel 12: First Diff.

```
for k = 1 to 1000
  x[k] = y[k+1]-y[k]
```

Livermore Kernel 13: 2-D Particle Pusher

```
for ip = 1 to 128
  i1 = p[1,ip]
  j1 = p[2,ip]
  p[3,ip] = p[3,ip]+b[i1,j1]
  p[4,ip] = p[4,ip]+c[i1,j1]
  p[1,ip] = p[1,ip]+p[3,ip]
  p[2,ip] = p[2,ip]+p[4,ip]
  i2 = p[1,ip]
  j2 = p[2,ip]
  p[1,ip] = p[1,ip]+y[i2+32]
  p[2,ip] = p[2,ip]+z[j2+32]
  i2 = i2+e[i2+32]
  j2 = j2+f[j2+32]
  h[i2,j2] = h[i2,j2]+1.0
```

Livermore Kernel 14: 1-D Particle Pusher

```
for k = 1 to 150
  ix = grd[k]
  xi = ix
  vx[k] = vx[k]+ex[ix]+(xx[k]-xi)+dex[ix]
  xx[k] = xx[k]+vx[k]+flx
  ir = xx[k]
  ri = ir
  rx1 = xx[k]-ri
  ir = ir AND 63
  xx[k] = ri+rx1
  rh[ir] = rh[ir]+1.0-rx1
  rh[ir+1] = rh[ir+1]+rx1
```