AN ANALYSIS OF THE CRAY-1 COMPUTER

Richard L. Sites

Applied Physics and Information Science Dept. C-014
University of California, San Diego
La Jolla, California 92093

Abstract

The Cray-1 is an extremely high-speed computer, intended to be used for large floating-point scientific computations. However, it is a well-balanced machine that can gracefully be used on a wide class of problems. The machine has two major architectual innovations: (1) 128 backup registers which represent a new layer in the memory hierarchy, essentially a programmer or compiler-managed cache, and (2) 8 vector registers holding up to 64 words each, and operated on by vector instructions. In this paper, we will describe the entire machine, discuss efficient ways to use the 656+ programmer-accessible registers, and discuss some of the design shortcomings.

1. Introduction

The Cray-1 is an extremely high-speed computer build by Cray Research Inc., a company formed by Seymour Cray in the early 70's. Before that, Cray was responsible for the design of the CDC 1604, CDC 6600 and CDC 7600. In many ways, the Cray-1 follows the evolution of the 6600-7600. Briefly, the CPU has a 12.5 nanosecond clock, and executes scalar instructions in 1 to 14 clock pulses (CP), with many instructions taking 3 or fewer CP. It is possible to execute 80 million instructions per second. The CPU and memory are physically housed in a cylinder 6.5 feet tall and 4.5 feet in diameter (2.0m x 1.4m), with the power supplies in a 9-feet (2.7m) diameter skirt around the base. With a full-size memory, this compact package weights over 5 tons and consumes about 150,000 watts of electricity. There are no flashing lights or buttons to push on the machine. Instead, it is connected to an Eclipse minicomputer which can display the instruction counter and other registers/memory words on a crt screen.

The main memory can be up to 1 million (2^{20}) 64-bit words of 50 nsec cycle-time bipolar memory. The memory occupies 2/3 of the mainframe and the CPU occupies the middle 1/3. The memory is 16-way interleaved, so the CPU can easily achieve a data transfer bandwidth of one word per clock cycle (over five billion bits per second) with three fourths of the available memory cycles left over for I/O transfers. For sequential CPU accesses and no I/O at all, 4-way interleaving would be sufficient to achieve a bandwidth of one word per clock cycle, but the extra interleaving is cheap and allows for random access intermixed with I/O. Using Amdahl's rule of thumb of one bit of main memory per instruction per second executed [1], the one megaword memory (64 megabits) is in good balance with the 80 megahertz instruction rate.

The I/O system consists exclusively of channel connections to other computers and channel connections to high-speed permanently-mounted disks. The disks rotate at 3600 rpm, and have 18 sectors of 512 words each on a track, with 10 tracks per cylinder. The sustained transfer rate for one disk is about one word every 1.8 usec, or about 4.5 megabytes per second. Using Amdahl's rule of thumb of one bit of I/O per instruction executed [1], 80 million instructions per second, and 35 million bits per second per disk, the machine will need to have somewhat more than two disks actively transferring data all the time, in order to be well balanced. This is easily achievable by the hardware, but for the operating system it may be harder, as detailed in Section 4.

2. Memory Hierarchy

The Cray-1 contains four major elements in its memory hierarchy. The fastest level consists of 8 S-registers, 8 A-registers, and 8 V-registers. The S-registers are 64 bits wide, and are used primarily for full-word fixed and floating-point arithmetic. The A-registers are 24 bits wide and are used primarily for address calculations. The V-registers each contain 64 words, and are further discussed in the next section on vector instructions. The second-fastest level of the memory hierarchy consists of 64 T- and 64 B-registers. The T-registers are 64 bits wide and are used essentially as a programmed cache or backup for the S-registers. The B-registers are 24 bits wide and are used the same way for the A-registers. There are no backup registers for the V-registers. The third level of the memory hierarchy consists of the main memory, up to 1048576 (1M) words. The fourth level consists of the high-speed disks.

It is the introduction of the B- and T-registers that sets off the Cray-1 memory architecture from that of other machines. These registers contribute directly to the balance, versatility, and speed of the machine. Figure 1 shows the major interconnections between the various registers. Note that the transfers between (A,B) and (S,T) pairs are all one-cycle instructions, so access to the B- and T-register backup level is extremely fast. The backup registers can also be transferred to and from main memory, but only via block transfers which move from 1 to 64 words. These transfers have start-up times that are the same as the time for a single scalar load/store, but after that they transfer a new word every clock cycle, so that loading all 64 T-registers takes only 80 CP, or 1 usec. The A- and S-registers also have direct paths to/from main memory, but the scalar instructions which use these paths take two CP to issue, so the maximum scalar data rate is half that of the block transfers, or one word every 2 CP.

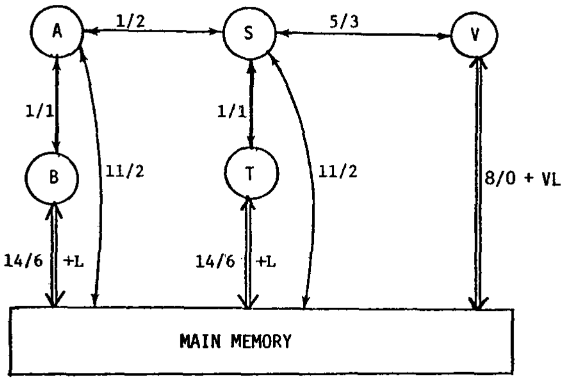

Figure 1. Interconnections of registers in the Cray-1, with transfer times in units of clock pulses (CP, 12.5 nsec each). Times are given as load/store, where load refers to changing the contents of the higher or left-most register. Double lines indicate block transfers, which take the given startup time plus one CP per word.

Arithmetic can only be done in the A- and S-registers, and there are no memory-to-register arithmetic instructions. To use the registers efficiently, some variation on the following scheme is needed. First, keep all scalar local variables for any procedure in the B- and T-registers, in order to have fast access. Second, keep all local arrays, records, and long strings in main memory, since there are no instructions which can index into the B- or T-registers, and there generally isn't enough room in these registers anyway. Third, do all expression evaluation by loading the operands into A- or S-registers, doing the appropriate operation, and finally storing the result back into B- or T-registers or main memory. To do a subroutine call, block store the caller's B- and T-registers to main memory, so the empty registers will be available for the called routine to use. To do a subroutine return, block load the caller's old values. It is awkward to try to preserve the A- and S-registers across subroutine calls, because all the saving/restoring of these must be done with separate instructions for each register, for a worst case of 16 stores and 16 loads. This element of the architecture strongly discourages compilers from trying to keep common subexpressions or local variables in the A- and S-registers; instead, the fast-access B- and T-registers are used. This is somewhat counter to the usual optimizing compiler goal of heavily using the fastest registers. We shall return to this point in the discussion of instruction scheduling in Section 6.

The register architecture is designed so that operands and instructions (see Section 7) never come directly from main memory, and so that when access to main memory is necessary, efficient block loads and stores can often be used.

3. Vector Instructions

The second architectural innovation in the Cray-1 is the vector facility. The 8 V-registers can each hold a vector of 1 to 64 words, and there are vector instructions of the form "add elements of V1 to elements of V2 and put the results in V3." A single 16-bit instruction can do up to 64 additions, and after a small start-up time of 8 CP these additions are done at the rate of one per clock cycle. Thus, 64 floating-point adds can be done in 72 CP, or 900 nsec, or about 70 million additions per second. As we shall see below, this figure can be improved by another factor of 2-3. The intended use of the vector registers and instructions is to block load vector operands (or 64-word pieces of longer vectors) from memory, operate, then block store the results back into memory. The vector registers provide exactly the same advantages that scalar registers provide: (1) instruction formats are compact compared to formats which contain many memory addresses, (2) common subexpressions can be retained in registers and reused quickly, and (3) the computation is insulated from the slow speed of the main memory, since operand loads can be started early, and both loads and stores can be overlapped with other computation. The re-use of common subexpressions in registers gives the Cray-1 an advantage over memory-to-memory vector architectures, such as the Texas Instruments ASC or the Burroughs BSP. The very low start-up times give the Cray-1 an advantage over the long start-up times of the CDC STAR-100. Vector instructions can be faster than scalar arithmetic for vectors as short as 3 words, and do noticeably better on vectors of length 5 or more. The fact that the vector instructions are fast even for short vectors, combined with the fact that the scalar instructions are also quite fast makes the Cray-1 an extremely well-balanced machine which can efficiently run programs that have very little to do with floating-point matrix computations.

In addition to the expected vector operations (+,-,*,/,and,or,xor), the Cray-1 has similar operations which combine the elements of a vector with a single scalar from an S-register. There are also vector shifts which effectively shift an entire vector register (4096 bits) right or left.

The four vector comparison instructions compare up to 64 elements of a V-register to zero and set corresponding bits in the Vector Mask Register (VM) to indicate whether each element matches the comparison (zero, nonzero, positive, or negative). The VM can then be used in a Vector Merge instruction to selectively merge two input vectors into a single output vector.

Example 1. Vector absolute value. Each line is one instruction. No branch instructions.

a.	Load 64 elements of vector ABC into V1.	
b.	$V2 \leftarrow 0-V1$	Zero is in an S-register. Negates each element.
c.	$VM \leftarrow V1 < 0$	Form mask of 64 0's and 1's. 1 if element is negative.
d.	$V3 \leftarrow V2$ merge V1	Pick element from V2 ($-ABC_i$) if corresponding VM bit is one, else from V1 (ABC_i)
e.	Store 64 elements from V3 back into vector ABC.	

The vector merge can be used to do 64 interchanges as the inner loop of a sorting routine by (a) loading up two sets of elements to be compared, (b) subtracting them, (c) comparing the differences to zero and setting the VM appropriately, (d) doing a vector merge to put all the larger elements in one register (pair-wise MAX), (e) doing a vector merge with the operand registers reversed to put all the smaller elements in another register, and (f) storing the 64 sorted pairs. Again, no branch instructions are used.

The vector load and store instructions are block transfers, just like the B- and T-register transfers, except that the vector instructions include the extra flexibility to specify an increment other than +1 for generating the sequence of memory addresses. Any 24-bit increment (positive or negative) can be used. This makes it quite easy to pick up columns of arrays stored in row-major order, or diagonals, or whatever, so long as the addresses form an arithmetic progression. The transfers will run at full speed of one word per clock cycle unless the increment is a multiple of 8, in which case all the data words are in only 1 or 2 memory banks and the banks have a 4 CP cycle time. This is the one instance in the entire machine where the interleaving may not be sufficient. The problem is mitigated by three factors: First, the worst slow-down is only a factor of four, not the factor of 16 expected on some 16-way interleaved machines (or the 17-way BSP); second, once an operand is loaded into a V-register, it can be used many times without going back to memory; and third, the transfers can be completely overlapped with other vector and scalar computation (but not other memory references).

There is one remaining feature of the Cray-1 vector facility which is not strictly part of the machine architecture seen by the programmer, but rather is part of the particular implementation built by Cray Research. Nonetheless, the feature is important enough to be discussed, considered, and copied in other machines. The feature is called chaining, and it simply allows a vector instruction which uses the result of a previous vector instruction (including load) to start as soon as the first word of the previous result is available, instead of waiting until the entire previous vector is formed. In evaluating an expression such as A*B+C, chaining allows the add to start 9 CP after the multiply, instead of waiting 73 (9+64) CP. Thus, the 64 results of the entire expression are ready 64 cycles sooner than without chaining. In doing this, the machine actually generates two results in one clock cycle -- after the start-up time for the add, the floating adder starts delivering one sum per clock cycle, while the floating-point multiplier is still delivering products for the remaining 56 intermediates A*B. With two vector units operating at once, it is possible for the Cray-1 to deliver 160 million floating-point results per second; with three units running, 240 million results. This effect is in fact independent of chaining, but crops up most often in that context. With most vector instructions requiring two operand registers and one result register (all usually distinct), it is clear that with just 8 V-registers available, three results at once is unlikely, and four is out of the question. Two results at once, sustained over quite some time is practical, though. (Some of the hand-written matrix routines at Los Alamos achieve 140 million floating-point operations per second in production programs.)

4. Interrupts and I/O System

Like the CDC 6600 and 7600, the interrupt system on the Cray-1 is based on Exchange Packages, containing 16 words of processor-state information. To change tasks, it is necessary to save the current processor state, and to load up a new processor state. The Cray-1 does part of this job automatically when an interrupt occurs, or when a program executes either a normal or error exit instruction. When such an event happens, the Cray-1 saves the 8 S-, the 8 A-registers, and a few more pieces of information, including the program counter and the Monitor Mode bit. These are packed into 16 words and swapped with 16 words at the address specified by a hardware Exchange Address register. This entire process takes 48 CP, or 600 nsec. The exchange is quick partly because the exchange packages are constrained to have exactly one word per memory bank, starting at bank zero, so the 16 fetches and 16 stores can go at full speed.

When an I/O interrupt occurs, there often will be more than one instruction in the middle of execution. In order to save a consistent set of register contents and program counter, it is necessary to wait until all instructions complete before doing the exchange. Unless a long vector instruction or a scalar divide was just started, this wait will usually be just a few CP, perhaps another 100 nsec.

When a running program generates an interrupt, such as floating point error or bad memory address, the same process of waiting for currently issued instructions to complete must happen. Thus, in general, the program counter (interrupt address) for a program fault will be a few instructions past the one actually causing the error. This is the well-known imprecise interrupt, and causes the same debugging problems on the Cray-1 as on other such machines (IBM 360/91, CDC 6600).

The exchange package does not contain all of the hardware state information, so software interrupt handlers must save the rest of the state. "The rest" includes the 512 words of V-registers, the 128 words of B- and T-registers, the Vector Mask, and the Real Time Clock. Saving the V- and T-registers alone takes over 640 CP, or 8.0 usec. Loading up those registers for a new task takes another 8.0 usec. So complete task switching takes on the order of 20 usec, which is enough time to execute over 1500 instructions. The Cray-1 is not a machine which will run efficiently if full task switches are done very frequently.

The hugh number of words in the machine state encourages a software approach which often does only partial task switches. For example, the operating system supplied by Cray Research does not save the V-registers on an I/O interrupt, and no part of the interrupt handler uses vector instructions (and hence V-registers). So long as control returns to the original user task, this scheme is logically sound, and certainly faster than a full save. It costs something, though. The I/O interrupt handler, and potentially a large part of the operating system, must run using only a subset of the machine's architecture. There is a wide-open chance for errors, if not this year, then maybe two years hence as software is modified by people other than the original coders. If sometimes an I/O interrupt occurring at a random time changes a value in, say, B00 upon return, then some very mysterious bugs will occur. There is also a potential for security violations -- if a noticeable portion of the operating system runs with some other task's live data in 640 registers, it becomes likely that someone can find a way to read those registers and "steal" the data in them. The issue is basically that of trusting a part of the operating system to run with access to more data than it needs, but hoping that it doesn't peek! This is not a situation one usually designs into an operating system which must strictly enforce access privileges in a security installation such as Los Alamos, but it is a situation encouraged by the hardware architecture.

Unfortunately, the slowness and tempting shortcut problems of full task switches are exacerbated by the I/O structure on the Cray-1. The entire I/O structure in the CPU consists of one privileged instruction which sets limit and start addresses for one of the 24 channels (12 in, 12 out). When the start address is set, the channel transfers words (in or out, depending on what kind of channel it is) until the limit address is reached. Then the channel stops and generates an I/O interrupt. What is actually transferred, or its meaning, is of no concern to either the CPU or the channel. Unfortunately, the disk controller is concerned, and must go through a rather awkward sequence in order to read a 512-word record from disk.

First, the actual disk might be shared by two controllers, perhaps on different computers. There are "reserve" and "release" command sequences which can be sent to the controller, but to shorten the example, we will assume no such sharing. We will, however, assume that the disk is not positioned at the cylinder we want to read from. The CPU constructs a one-word Seek command at, say, location 5300. The limit register for the disk controller output channel, say 2, is set to 5300. Then the start register is set to 5300 and one word is transferred by the channel to the disk controller, i.e. the Seek command is sent. The amount of time this transfer takes is not guaranteed, but is often in the range of 100 nsec to 1 usec. If the disk controller is dead or busy, it might take much longer. Thus the CPU program which activates the channel cannot afford to just go into a "busy waiting" loop until the single word is transferred. Instead, it must do an exchange jump back to a user (or another system) task, with the I/O interrupts enabled. This takes at least 600 nsec, and usually the channel is ready to interrupt as soon as the exchange is complete, thus immediately triggering a second exchange. If the channel or controller is dead, the interrupt might never come. But in the usual case, we are back where we were 1.2 usec before, having just transmitted a one-word command to the disk controller and having received confirmation that the command arrived. The command has not yet executed. There is nothing more for the CPU I/O task to do until the Seek is complete, so it immediately triggers a third exchange, back to another task. Unfortunately, the Cray Research disk controllers have no way to generate an interrupt when the Seek is completed. So some kludge must be invented, such as "whenever an exchange to the operating system occurs, check to see if an outstanding Seek is completed." (The following Read command can be sent, but this ties up the controller during the Seek.) In whatever way, we eventually end up back in the I/O task with the disk at the right cylinder. The I/O task must then send a one-word "Read" command to the disk controller. This also is accomplished via a one-word output sequence and two exchange jumps. As soon as the read command itself is sent and accepted, the proper input channel (number 3 in this case) must have its limit and start registers set to specify the 512-word data buffer in memory. When the I/O interrupt for this transfer arrives, we have finally read one record, and have done at least 10 exchange jumps, or 6 usec of CPU time. If all those exchange jumps also involve saving and restoring user and I/O task B-registers, T-registers, and V-registers, the CPU time can jump to 200 usec. All this for reading a single disk record with no error situations. The next record will come spinning around 900 usec after the first, so 200 usec of dedicated CPU time can be quite noticeable. Clearly, shortcuts must be used.

The above analysis is for the case of Seek, then read/write. For transfer of consecutive sectors on the track, the process is much faster, as successive Read/Write commands can be sent with no further overhead. Each disk controller contains two sector buffers (512 words each), and if the CPU program reads sector 5, that sector is actually read into buffer A, then transferred to main memory from the buffer. The buffer-to-memory transfer is done about six times faster than the disk-to-buffer transfer. In addition, the disk controller automatically copies sector 6 into buffer B as that sector passes the read head. Normally, this overlaps with the transmission of buffer A to memory. Buffer B can then be transferred to memory while sector 7 is read into buffer A. The effect of this design is that there is no critical between-sector time of just a few microseconds in which the CPU must start the read of the next sector. Instead, nearly 5/6 of a sector time (about 750 usec) is available for the CPU to respond to the "sector 5 read" interrupt and initiate the "read sector 6" sequence. This makes it much easier for a loaded operating system to keep the disks busy. A similar process occurs on output, in which sectors are written on disk without missing a revolution so long as the CPU keeps filling buffers with new data. The entire buffering process also allows wide margins on how quickly a given I/O word is accessed in memory, so the channel does not lose data if there are occasional bank conflicts with CPU requests.

5. Arithmetic

The Cray-1 arithmetic follows the style of the CDC-6600-7600 series. A floating-point number consists of a sign bit, 15-bit exponent, and 48-bit binary fraction (sign-magnitude). The large exponent is used to represent numbers in the approximate range 10^{-2500} through 10^{+2500}, plus an overflow range and an underflow range. The lowest 1/4 of the exponent range (octal leading digit 0 to 1) represents underflowed values, the middle half (octal leading digit 2-5) represents normal values, and the highest 1/4 (octal leading digit 6 or 7) represents overflowed values. An operand or a result in one of the extreme ranges will generate an interrupt if the machine is not in monitor mode, the floating-point interrupts are not masked off, and the operation is a scalar instruction. No interrupt occurs for vector instructions, which instead run to completion, leaving results with large exponents (for overflow), or zeros (for underflow).

One implication of this scheme is that range errors are sometimes not detected when a result is formed, but only much later when a subsequent instruction detects an out-of-range operand. For the current implementation of the architecture, even detected out-of-range results do not generate an interrupt until 2 to 11 more instructions are executed, so all floating point interrupts (and in fact all others) are imprecise. The current implementation also detects range errors before post-normalizing, so some in-range results generate interrupts, and some out-of-range results don't. All of this contributes to the kinds of numerical processing difficulties that Kahan has documented so well [2].

There are other anomolies in the floating-point arithmetic. Addition is done with one guard bit and the result is truncated after normalization, with no rounding. Multiplication is not commutative, because part of the partial sum pyramid is not implemented. Thus, instead of forming a full 96-bit product fraction from two 48-bit operands, the Cray-1 forms an approximate 56-bit product with a constant rounding factor added in to somewhat compensate for the missing bits. The overall effect of this is that A*B and B*A may differ in the last bit, and that products in general may differ from the mathematically correct result by one in the lowest bit.

There is no divide instruction on the Cray-1. Instead, there is a unary reciprocal approximation instruction which produces a result accurate to 31 bits. This accuracy can be improved to 47 bits (not 48) by a reciprocal iteration instruction, which takes A and 1/A (31-bit approx.) and calculates 2 - (A*1/A), which is a number slightly larger than 1.0 if the reciprocal is slightly small, and a number slightly smaller than 1.0 if it is large. Multiplying the 31-bit approximation by this correction factor results in a 47-bit approximation (Newton's iteration). One further multiply by B gives the quotient B/A.

The advantage of this scheme is that the reciprocal approximation unit can be built out of strictly combinatorial logic, so the entire unit can be pipelined, with the ability to start a new reciprocal every single clock cycle. This, plus chaining the reciprocal results into one of the subsequent multiplies, allows complete vector divides to be done at the rate of one result every 3 CP, or almost 27 million divisions per second. This contrasts sharply with earlier machines, which are hard pressed to achieve even 1 million divisions per second.

The disadvantage of this scheme is that the results are not quite right, so some of the speed advantage may be lost in further correction steps. For example, the integer A mod B routine must use the floating-point divide sequence, truncate the quotient, then back-multiply and subtract to find the mod. Sometimes, the floating quotient is just larger than an integer, when in fact it should be just smaller. This forces the mod routine to make an extra check for a negative final result, then add B if this occurs. It is also possible that the result is too large by B.

Integer arithmetic in the Cray-1 is done on either 24-bit or 64-bit two's complement numbers. There are add/subtract instructions for both formats, and there is a multiply instruction for the 24-bit (address) format. Long multiply and divides of either kind must be done by converting to floating-point and back. The effect of needing to convert is that the full range of operations can only be done on integers up to 46 bits long. There are no overflow detection mechanisms for any of these three lengths.

6. Compiler Optimization

The major goal of any optimizing compiler for the Cray-1 must be effective use of the hundreds of fast registers. This use falls into four categories: storage of variables, expression evaluation, temporary storage of common subexpressions, and addressing/subroutine linkage overhead. The last tends to take a handful of dedicated registers, and the compiler cannot change this number, so we won't discuss addressing/subroutine linkage further. As discussed in Section 2, small local variables should be kept in the B- or T-registers, expression evaluation should be carried out in the A-, S-, or V-registers, and common subexpressions should normally be stored in the B- or T-registers. Within this general framework, let us look at the register allocation problem in more detail.

In a traditional optimizing compiler for a register machine, one goal is to minimize the number of registers used to evaluate a group of arithmetic expressions, so that more fast registers can be left over for holding common subexpressions. This is usually done by evaluating the more complicated operand of a binary operator first, in effect minimizing the "stack height" of the set of intermediate temporaries. For example, the expression

```
        A - B*C
```

can be evaluated using only two registers by doing the multiply first:

```
        S1 ← B
        S2 ← C
        S1 ← S1*S2    (arithmetic is register-to
        S2 ← A          register only)
        S2 ← S2-S1.
```

Loading A first (strict left-to-right) would require three registers. Note that the evaluation order can be changed without assuming that any of the operators are commutative (and in fact floating multiply is not, as mentioned in Section 5). Although it is not clear in the above example, the Cray-1 is in fact a three-address machine; we will use this flexibility below.

The above sequence of 5 instructions would take 31 CP to execute on the Cray-1, with a timing chart that looks like this:

```
S1 ← B        IIxxxxxxxxx
S2 ← C          IIxxxxxxxxx
S1 ← S1*S2        .........Ixxxxxx
S2 ← A                      IIxxxxxxxxx
S2 ← S2-S1                    .........Ixxxxxx
              0---------10--------20--------30---
                         Time in CP
```

Where "I" means the instruction is issuing, "x" means it is executing, and "." means it is delayed, waiting for a previous instruction to finish. Note that the multiply waits for the second load, and the subtract waits for the third load. Note also that the execution (but never the issuing) of the various instructions is overlapped, through the use of 12 different functional units.

We can speed up this sequence by doing the load of A sooner:

```
S1 ← B        IIxxxxxxxxx
S2 ← C          IIxxxxxxxxx
S3 ← A            IIxxxxxxxxx
S1 ← S1*S2          .......Ixxxxxx
S2 ← S3-S1                 ......Ixxxxx
              0---------10--------20--------30---
                         Time in CP
```

The new sequence takes only 26 CP because the load of A is better overlapped, but we have used three registers instead of two. Not only is this the best strategy on the Cray-1, it is the best strategy even if a third register is not available, since it only takes one CP to save an S-register in a T-register, and one more CP to restore it. Spending these two CP saves us 5 CP, a net gain of 3.

The essential timing problem that we ran into with the two-register sequence is that we tried to re-use S2 for loading A before the previous value of S2 had had time to be used. In general, an optimizing compiler will need to simulate the instruction timing, and keep assigning intermediate results to new registers until the timing simulation shows that a previously-used register is available again. In the absence of a careful timing simulation, the best strategy that a compiler can adopt is the one which uses the most registers: round-robin allocation!

A more complete solution to the register allocation problem is to generate code for a group of arithmetic expressions (i.e. all those in a basic block that has no branches or labels) using an arbitrarily large number of pseudo-registers, then schedule the resulting code to minimize issue delays, then map the pseudo-registers into a minimum number of real registers. This process has the danger that it may require more than 8 real registers, in which case some spilling must be done. It also has the problem that moving instructions around to minimize issue delays also moves around the definition and use points of common subexpressions, which can sometimes result in a worse overall allocation. This issue is worthy of further study.

It is almost always an advantage to store local scalar variables in the B- and T-registers, instead of main memory, but the need to save and reload these values across a subroutine call cuts into the effectiveness of using these registers. Also, modular programs with large numbers of small subroutines tend to have only about 5-10 local variables which could possibly be kept in these fast registers, leaving the other 50-odd wasted and unused. If an optimizing compiler is allowed to compile a group of subroutines all at once (as in Pascal), then it is possible for the compiler to merge the activation records (local variables) of many subroutines until almost all 64 B- or T-registers are used. If this is done, it is not necessary to save and restore these registers across all subroutine calls, but only those which call routines whose local variables won't fit in the used registers. For 10 local variables per routine and 64 T-registers, this means spilling only at about every sixth subroutine call. This activation record merging results in quite efficient use of a large but finite number of registers, and might well be a technique to be adopted in generating code for microprocessors which have from 20 to 200 words of storage right on the microprocessor chip, and the rest of storage off-chip.

Register allocation is one area of concern for an optimizing compiler; a second area is that of actually using the vector facilities of Cray-1 architecture. This is still an open area for research, but some of the techniques used in compilers for previous machines, such as the Texas Instruments ASC Fortran compiler [3], can be easily adapted. Although this is an important area to explore, partial solutions will be acceptable for a while because the scalar arithmetic on the Cray-1 is so fast that only another factor of 2-4 improvement is available by using the vector instructions. This contrasts sharply with other machines in which the scalar version of a program may run 20-100 times slower than the vector version.

7. Minor Features

One very useful feature of the Cray-1 is that it has a 64-bit clock which counts once every 12.5 nsec, synchronously with the rest of the machine. This clock register can be read by a one-cycle non-privileged instruction. This makes it both easy and fast to instrument code and find out precisely how many CPs it takes to execute. Results will be identical from run to run since the clock is synchronized with the CPU, instead of being sampled periodically. At Los Alamos, we have answered many obscure questions about the machine's timing by reading the clock, executing a sequence of two or three instructions, reading the clock again, then subtracting. Since no Cray-1 instruction times are data dependent, and there is no I/O interference for scalar loads and stores (they have priority over I/O), the clock times are quite reproducible, even when the total time for a sequence is under 100 nsec. A synchronous clock should be included on more machines.

The Cray-1 has good facilities for generating constants. Small integers in the range 0-63 can be loaded into an A-register with a 16-bit immediate instruction. Long constants of ±22 bits can be loaded into A-registers or S-registers with 32-bit immediate instructions (this is well matched to the 20-bit implemented memory address range). Masks of left- or right-justified one-bits can be generated in an S-register with 16-bit immediate instructions. Other patterns, including floating-point numbers with simple fraction parts (e.g. 2.0) can be generated by an immediate instruction followed by a shift. In addition, A0 used as an index register or as a first operand always supplies the value zero, S0 used as a first operand always supplies the value zero, A0 used as a second operand always supplies the value 1 (as in $A4 \leftarrow A4 - A0$ to decrement A4), and S0 used as a second operand always supplies a sign bit, 2^{63}. These turn out to be just the right constants to have, except S0 used as a second operand could more usefully supply the value 1.

There is no Load Address instruction with indexing, so construction of argument lists is slightly awkward.

The Cray-1 has four non-contiguous instruction buffers of 16 words each. Instructions take 1/4 or 1/2 word each, so one buffer holds from 32 to 64 instructions. These buffers have an extremely wide four-word data path to main store, so an entire buffer can be loaded by saturating the 16 memory banks in 4 cycles of 4 words each (due to routing delays and single-bit error correction, the actual buffer load time seen by the executing program is 11 CP). So long as a loop, including subroutine calls, branches, or whatever, fits in the four buffers, no time at all is spent on instruction fetch. Even straight-line code benefits from the block-load mechanism.

8. Conclusion

The Cray-1 is an extremely fast and versatile computer, but it will take many years of software research and development to fully exploit the new aspects of the Cray-1 architecture.

The experience on which this paper is based was gained while working at the Los Alamos Scientific Laboratory. I gratefully acknowledge the support and encouragement of Forest Baskett and my other friends in the group there.

References

[1] G.M. Amdahl, "Storage and I/O Parameters and Systems Potential," IEEE Computer Group Conf., June 1970, Washington, DC, pp. 371-372.

[2] W. Kahan, "A Survey of Error Analysis," Information Processing 71, North Holland Publishing, pp. 1214-1239.

[3] D. Wedel, "Fortran for the Texas Instruments ASC System," SIGPLAN Notices, March 1975 (Vol. 10, No. 3), pp. 119-132.