An Abundance of Registers

Paul E. Cohen

One striking feature of NEC's V60 and V70 microprocessors is their ample complement of general purpose registers. Thirty-two registers are available, each being thirty-two bits wide. There is no specialization of these registers in the sense of any being restricted to serve only as data registers or only as address registers; software can use these registers as it chooses. Consecutive pairs of these registers can be also used as sixty-four bit registers.

These processors do have a few instructions that treat particular registers in special ways. For example, R31 is assumed to be used as a stack pointer so PUSH and POP instructions as well as CALL and RETURN instructions give it special treatment. Similarly, R30 is used as an argument pointer and R29 is used as a frame pointer so that the PREPARE and DISPOSE instructions (used for stack and frame management) treat these registers in special ways. Finally, R26, R27 and R28 are used implicitly by various string instructions. An environment that does not need these instructions is free to use the registers for any other purpose, but it is likely in most applications that these six registers will be reserved for their intended special purposes. That nevertheless leaves another twenty-six registers that can be used arbitrarily.

Register Allocation

One can ask whether a compiler will be able to make effective use of so many registers and an objective of this note to argue that the answer is clearly yes. Even if a single procedure cannot usually take good advantage of so many registers, a program that is a thread of many procedures can.

Even with 26 available registers it is important to be very judicious about allocating variables to them. It is the job of a compiler (or assembly language programmer) to make these choices wisely. Possibly owing to the usual scarcity of registers, compilers are often quite sophisticated in this area, being able to track the lifetime of variables so that a register can be re-used once the variable that is first allocated to it will not be referenced again. In what follows, a register will be called "bound" if a variable is assigned to the register. In this case the variable will be called "alive." During execution of a routine, a register that is bound may later be "free" (not bound) if the compiler can recognize that the data will never be used again.

Saving and Restoring Registers

One difficulty with assigning variables to registers is that any change of context may require register values to be saved and then later restored.

There are many kinds of context changes, including procedure calls or returns, process switches, or entries to operating system code to handle exceptions or interrupts. Whenever a variable is used several times before it must be saved there is a performance advantage to placing it in a register rather than in memory. Nonetheless, the need to save and restore registers can significantly reduce the advantage. The advantage of having a large number of registers is that it becomes possible sometimes to use alternate registers rather than re-using the same registers, thus avoiding some costly saves and restores.

Software that is executing when an exception or interrupt happens generally has no way of anticipating the event and so is in no position to prepare for it by saving any registers. It therefore falls to the event handler to save whatever registers it will use. In the case of simple procedure calls, however, there is the additional possibility that the calling routine can save the registers. Unfortunately there are disadvantages to either of the obvious approaches.

If the calling routine is made responsible for saving registers then it must save the registers that it is using before it can call another procedure. The called procedure may not need the clean slate of registers that it receives, and when this is the case excessive number of saves is unavoidable.

Alternatively, if the called routine is made responsible for saving the registers, then it may unnecessarily save registers that were not used by the calling routine. The main disadvantage to this approach is that the called routine lacks information that the calling routine has readily available, namely which registers hold live data. This is merely a difficulty rather than a dilemma, however, since the calling routine can pass the needed information to the called procedure. It is important only to ensure that, on average, the overhead of passing along this information is justified by its benefits.

THE REGISTER MASK APPROACH

Since the V60 has an abundance of general purpose registers, it is not unreasonable to devote one of them to tracking register usage. Let us suppose that a register mask is always passed along in R25 to indicate which registers hold active data. Notice that the compiler has the information needed to compute such a mask since at any instant it knows which registers it is using.

The called routine knows which registers it will use and it begins its execution by saving each of those registers for which the corresponding bit in R25 is set. Fortunately, the V60's PUSHM and POPM instructions for saving and restoring multiple registers take just such a register mask as an argument.

An example will help illustrate what must happen. Suppose a procedure uses registers R0, R1, and R5. It begins execution by computing a save-register mask, say in R26 (this register can be used for a scratch pad since no string operations are to be done immediately). The register R25 is

ANDed with the word that has bits 0, 1, and 5 set, and the result placed in R26. It then executes

PUSHM R26

which saves up to three registers on the stack. Later it needs to call another procedure, but before it does it must re-compute the value in R25 (after saving the old value). Suppose when it makes the call, only registers R0 and R5 are still alive; then before the call, bits 0 and 5 of R25 must be set, and bit number 1 can be cleared since register R1 is now known to contain no useful information. Later, before returning, R25 and R26 must be restored to their earlier values and

POPM R26

executed to restore the registers that have been modified.

Some Operating System Considerations

Notice that in this example, R25 is updated only at procedure call interfaces. Another alternative is to update R25 whenever a register begins or ends holding useful information. This entails some additional overhead, but has the advantage that R25 holds accurate register use information to be used by interrupt and exception handlers. Otherwise, these event service routines must assume that all of the registers hold important information that must be restored on return to the user environment.

There is also the possibility of restricting some of the registers for the sole use of the event handlers. The purpose would be to guarantee that, except when handler nesting occurs, there is no need for a handler to save and restore registers (security considerations may require that the registers be zeroed before user code is re-entered, however).

Allotting a subset of registers to be used exclusively by event handlers is possible whether or not all of R25 is continuously updated, so long as just the bits corresponding to the subset are kept current. This is a better approach since it reduces the overhead of maintaining the register mask. So long as user code avoids these dedicated registers, keeping the mask updated means only keeping a small portion of the mask zero. User code can cheat and use one of the dedicated registers provided it first sets the appropriate bit in the mask; the only penalty for doing this is some added latency in the event handlers that need to save and restore extra registers. If a user were to fail to set the appropriate bit in R25, then anything that user stores in the corresponding register could be unpredictably corrupted by a fault or interrupt.

Selecting Registers

A compiler is at liberty to choose any or all of the low numbered twenty-five registers to use in a given procedure, although it is quite likely that

fewer than twenty-five will normally be needed. The decision of which registers a procedure will use must be made at compile time and unfortunately until run-time it is impossible to determine which registers are free. Some saving and restoring of registers seems unavoidable even when a sufficient number of free registers is available.

Any scheme that allocates the registers in a fixed order will take little advantage of the valuable registers that are low on the list. It may be beneficial to provide compiler options (perhaps as specialized comment lines) to allow the programmer to specify which registers to use. This could be a good performance tuning tool (as well as a way to make embedded assembly language escapes much more useful) but a better approach for the normal compilations would be for the compiler to use a random number generator to select registers. This gives a fair chance for a procedure take advantage of any given free register and unless some global flow analysis is used to provide some special knowledge about which registers are likely to be free, a fair chance is the best chance possible.

There are clearly some fairly sophisticated alternatives to the random approach that also deserve to be explored. It is not clear, however, that always using free registers when they are available would be an optimal strategy, even if some way were found to implement that strategy. The consequence of such an approach would be to bind all registers quickly before procedures are nested deeply. This could be good or bad depending upon how deeply procedures are typically nested.

In some environments it may be appropriate not to use all of the twenty-six registers in the way suggested above, but rather to allot some registers for other purposes. Applications for these other registers might be for passing parameters, to hold global values that a process uses frequently, or to hold temporary variables whose lifetimes do not cross procedure call boundaries. If the ten registers R16, ... , R25 were used in other ways then there would be the added benefit that the immediate values used for mask computations would be bytes or half-words but never full words.

Maintaining a register use mask for the purpose suggested above amounts to implementing a cache in the register file. Data is held in each register until that register is needed to hold newer data. Because the cache is managed in software there is some overhead, but the example above shows how small that overhead is likely to be. Of course particular software may experience only the overhead and derive no benefit, but the more usual case should be that enough register saving and restoring will be avoided to justify computing and preserving the masks. It should be noticed also that the mask itself can be saved to and restored from another register so that it, like other register data, can participate in the lottery of which data must be later stored in memory. This has the effect of increasing, by one, the number of registers used by each procedure that uses any register variables. Whether it is advantageous to do this or simply save it to memory of course depends on how often it is used.

The question of how many register saves would be avoided using a scheme such as suggested above is, like so many questions about computer performance, very dependent upon the characteristics of the user software. It is, however, possible to get some insights into the question through some statistical modeling.

Suppose a total of N registers are available to be allocated and we wish to determine how many registers are cumulatively bound when procedures are nested to a depth d. To simplify the analysis, we assume that the procedures of a task are uniform in the sense that each requires some number, n, of registers and that whenever a procedure calls another procedure, f of these n registers are not in active use. What we want to do is calculate the distribution function

$$B(b;\ N,\ n,\ f,\ d) = \text{The probability of } b \leq N \text{ registers being bound.}$$

For d = 0, this function takes the value 1 for b = 0 and is zero otherwise. For d = 1, it is easy to see that

$$B(b;\ N,\ n,\ f,\ d) = 1 \quad \text{if and only if} \quad b = n - f$$

More generally, B(b; N, n, f, d+1) can be calculated from B(b; N, n, f, d) using the recurrence formula,

$$B(b;\ N,\ n,\ f,\ d+1)$$

$$= \sum_{k=0}^{N} B(k;N,n,f,d) \cdot \text{Probability(choosing n registers so that (k + n) - (b + f) of them are among the k registers k = 0 that were previously bound)}$$

$$= \sum_{k=0}^{N} B(k;\ N,\ n,\ f,\ d) \cdot H((k+n)-(b+f);\ N,\ n,\ k)$$

$$= \sum_{k=0}^{N} B(k;N,n,f,d) \cdot \frac{\binom{k}{K+N-b-f}\binom{N-k}{b+f-k}}{\binom{N}{n}}$$

The coefficient H(x; N, n, k) in the above summation is the hypergeometric distribution. The derivation assumes only that in choosing the n registers from the total population of N registers, there is an equal probability of selecting any particular register.

If u(d) is the mean of the distribution B(k;N,n,f,d) then a straightforward calculation shows that

$$u(d+1) = n - f + (1 - n/N) \cdot u(d)$$

and consequently,

$$u(d) = \{ N (n - f) [1 - (1 - n/N)^d] \} / n$$

Some information about a more interesting distribution, S(s;N,n,f,d),can be easily derived from these facts. S(s) is defined as the probability that s registers are saved on entry to a routine at depth d. It can be computed easily from B(b) = B(b;N,n,f,d) and the hypergeometric distribution H(s;N,n,b) according to the formula

$$S(s) = \sum_{b=0}^{N} B(b)\, H(s;N,n,b)$$

An easy calculation then shows that the mean number of registers saved is

$$u(S) = (n - f) [1 - (1 - n/N)^d]$$

Notice that in the limit, as d tends to infinity, this mean approaches n - f. This is not too surprising since that is exactly how many registers each routine consumes.

As an example, suppose N = 20 registers are used, n = 5 registers are needed by each routine and f = 2 of the n registers are not alive at each procedure call. The following table is an application of the preceding formulas for the means.

Average Number of Registers for N=20, n=5 and f=2

Depth	Bound	Saved
0	0.000000	0.000000
1	3.000000	0.750000
2	5.250000	1.312500
3	6.937500	1.734375
4	8.203125	2.050781
5	9.152344	2.288086
6	9.864258	2.466064
7	10.398193	2.599548
8	10.798645	2.699661
9	11.098984	2.774746
10	11.324238	2.831059
11	11.493178	2.873295
12	11.619884	2.904971

As the depth increases, the number of registers saved approaches three, but for the first four levels of procedure calls, at least one additional

register save is avoided. Even for deeply nested procedure calls, however, there is an improvement over the five register saves that would be required with the usual approaches to saving registers. Such an improvement will occur whenever f, the number of freed registers, is greater than one.

Performance Tuning

Another opportunity for performance tuning is apparent from the low number of register saves that occur at low levels of procedure nesting. A programmer is frequently quite aware of which routines do the majority of procedure calls (and this is also not difficult to measure for a particular program). If a compiler option is provided to force particular procedures to save all of the registers and clear the register mask, then one (or several) new floors can be created so that selected procedures will behave as if they were nested at level d = 0.

Having Enough Registers

We now consider the effects of having fewer registers to allocate in the way suggested. If only six registers are available, instead of twenty then the table above becomes as shown below. This table consists of only three lines because the number of saved registers so quickly approaches its limit of three.

Average Number of Registers if N=6, n=5 and f=2

Depth	Bound	Saved
0	0.000000	0.000000
1	3.000000	2.500000
2	3.500000	2.916667

This is still an improvement over always saving five registers, but for code with shallow procedure nesting the additional advantage of having additional registers can be quite significant. If a process does 100,000 procedure calls at nesting depths of up to four, then the conventional approaches to register saving will mean that 500,000 register saves and restores will be required (although one should argue that, in the conventional approach, only 400,000 register saves and restores are required since in the register-mask approach it is usually necessary to save the register mask itself). By comparison, a register mask approach with six registers available will require approximately 300,000 register saves and restores while only about 130,000 will be needed if there are twenty registers available.

If No Registers are Freed

The results that are obtained by using the register mask approach can be expected to vary considerably depending upon the particular program, and

also depending upon the particular choices of registers. Another important variable is the compiler itself and in particular whether the compiler can, in a typical procedure, free any registers. If registers cannot often be freed, then using the register mask approach means that, in effect, one more register is consumed by each procedure than would be the case using the conventional approach to saving registers.

To explore this worst case, we assume that no registers are freed. Since the register mask must be saved (effectively increasing, by one, the number of registers that are used) then in fact a loss is likely to be incurred at sufficiently deep nesting. For example, using the formula for the mean that was derived in the previous section, if twelve registers are available for allocation and if each procedure uses two registers then at a depth of four or greater, more than one register must be saved on a procedure call. If fifteen registers are available, then it is not until a depth of five that the register mask approach requires more register saving than does the conventional approach. With 20 registers, this depth increases to seven.

Notice, however, that if no registers are used by a procedure then there is no need to save the register mask and consequently no increase in the number of registers used. Also notice that an increase in the number of registers used may or may not mean, in a particular instance, an increase in the number of registers saved. A particular instance may be much better than the average, and performance tuning provides an opportunity to frequently beat the odds.

SOME REAL C CODE

As an experiment in the effects of using the register mask approach, I instrumented a C program, unixusq.c, that had been broadcast over Usenet (a Unix network). In doing so, I made the assumption that no registers are freed by any procedure. Although this is likely to be the case with unsophisticated compilers, it is not as likely with the optimized compilers that are apt to apply the register mask technique. I also assumed that only the registers that are explicitly declared as register variables are put in registers.

For the program unixusq.c, this assumption means that the main routine uses two register variables, one other routine uses four register variables and that three other routines (the most frequently called) use a single register variable. A companion program to unixusq is designed to compress a file through Huffman encoding and unixusq decompresses the output of this other program. The file unixusq.c was compressed using unixsq. An experiment consisted of decompressing the result.

The results described below were obtained using unixusq to do this unsqueezing. A profile of this execution of unixusq shows that the three routines with only one register variable are called a total of 12738 times, and that the routines with more than one register variable are called only once each. Thus, out of a total of 12744 register saves, all but six are saves of only a single register. The flow of the program is that the main routine (which uses two registers) calls another routine (which uses four registers) once and that second-level routine calls two of the remaining routines. One of these calls

the remaining routine with one register variable. Thus the total nesting goes to a depth of four. This shallow nesting is a favorable factor for the register mask approach.

Twenty registers were assumed to be available, and a random number generator was used to select registers for each of the five routines. As separate, independent experiments, this was done a total of twenty times. For each routine, one extra register was assumed, since the register mask itself must often be saved. In the twenty trial cases, the number of required saves ranged from 189 to 19030 (a good indication that some provision for performance tuning would be useful). The average over all the trial cases was 11710 and the trials were distributed as shown below.

Number of trials	Registers Saved
1	189
5	6261 to 6451
10	12519 to 12770
4	18809 to 19030

On average, there was a slight performance gain (a savings of 12744 - 11710 = 1034 register saves) and in fact there is a dramatic reduction in one of the experiments. For frequently run code, such a favorable case would be worth the effort of a search; in fact with proper support from the compiler, such a search could be made a fairly easy task. Somewhat surprisingly, in this case the registers used by the main routine and the second level routine are disjoint, so that the remaining routines have only twelve free registers. It just happened in that case that the registers used by the remaining routines are largely disjoint from those used by these two.

A Simple Optimization

A natural question to ask is what the results would be if the compiler option (which was suggested earlier) to re-base the register mask to zero for the subroutine which is called only once. The effect of this is that the three registers that main uses are saved and these additional three registers are made available to the procedures that are lower in the execution tree. This does, naturally, improve the results shown above so that on average only 8224 register saves are necessary. The best and worst cases also improve so that the range of register saves over the same twenty trials is 3 to 12772. It is interesting to note that even the worst case now compares quite well to conventional approach of blindly saving all registers. A summary of the distribution is shown below.

Number of trials	Registers Saved
2	3
1	192
8	6262 to 6482
9	12521 to 12772

No claim is intended that the example provided here is typical code (whatever that is), but it is at least realistic code and not a program written specifically for the experiment. If these experiments are not conclusive because software is so variable and because the characteristics of a hypothetical compiler may not be well modeled, the experiments are at least instructive. The wide range of results that are found simply by selecting the registers with different random numbers is an indication that a compiler which applies the register mask approach should also provide some mechanism for performing similar experiments and for specifying registers.

CONCLUSIONS

It should be re-emphasized that the experiments assume that there are twenty registers that are available for allocation. Better results should be expected with additional registers just as worse results can be expected if fewer registers can be used.

It should also be noted that the relationship between saving registers and performance has not been addressed. This is another issue that is dependent on the characteristics of the code and the characteristics of the compiler itself. If registers are used in short routines that are called often then the cost will be much more than if they are used only in much longer routines. It is reasonable to assume, however, that if more registers are available then more registers will be used and consequently the costs associated with saving and restoring them will increase.

Whether there is an advantage in using register masks must be resolved for each particular compiler since there is some potential for a loss in performance as well as an opportunity for gain. The method will only result in additional overhead if there are so few registers available for allocation that they are nearly all used by each procedure. There will also be little benefit if procedure nesting is typically quite deep and if the compiler does not often free any registers and furthermore if no performance tuning is to be done.

There are trade-offs that must be evaluated, taking into account the characteristics of not only the compiler but also the characteristics of the software that is to be compiled and the even the characteristics of the programmers that will use it.

It is important how aggressive the compiler is with respect to using register variables and whether it recognizes when such variables are no longer alive. The nature of the code that is to be compiled is important because the depth of nesting and the size of typical routines is so important. Probably the least obvious consideration is the programming environment. Programmers in some environments (such as those doing real-time control applications) are more likely to take the trouble tune their code for performance than are programmers working in other environments.

The register mask approach appears to be most beneficial for those environments where programmers will make the extra effort to tune their code. However, for compilers which can track the lifetime of variables well enough to, on average, free at least one register variable per procedure, this method will provide an advantage even without performance tuning.