HOW TO USE 1000 REGISTERS

Richard L. Sites
Department of Applied Physics and Information Science
University of California, San Diego

ABSTRACT

The advent of VLSI technology will allow the fabrication of complete computers plus memory on one chip. There will be an architectural challenge in the very near future to adjust to this trend by designing balanced architectures using hundreds or thousands of registers or other small blocks of memory. As the relative price of memory (vs. random logic) drops even further, the need for register-heavy architectures will become even more pronounced. In this paper, we discuss a spectrum of ways to exploit more registers in an architecture, ranging from programmer-managed cache (large numbers of explicitly-addressed registers, as in the Cray-1) to better schemes for automatically-managed cache. A combination of compiler and hardware techniques will be needed to maximize effective register use while minimizing transmission bandwidth between various memories. Discussed techniques include merging activation records at compile time, predictive cache loading, and "dribble-back" cache unloading.

I. INTRODUCTION.

VLSI technology will soon make it possible to put an entire computer plus a large number of storage locations (perhaps 100-1000 registers) on a single chip. On a larger scale of a computer occupying a few printed-circuit boards, VLSI memories will allow economical designs with a number of localized memories that are "closer" to the processor logic using them than the larger main memory (Figure 1). How can computer architects make effective use of such <u>short-term memories</u>?

Figure 1. A computer with two localized <u>short-term</u> memories:

```
 --------------    --------------
| processing  |   | processing  |
|   logic     |   |   logic     |
 --------------    --------------
       |                 |
 --------------    --------------
| registers   |   |   cache     |     short term
 --------------    --------------        memory
       |                 |
 ---------------------------------
|          main memory            |    long term
 ---------------------------------       memory
```

In this paper, we first present a framework of issues for comparing short-term memory designs, then we present some new techniques for facing these issues. Our basis of comparison is a simple computer with <u>no</u> short-term memory, but only long- term main memory. All operations are done memory-to-memory, with no intermediate registers. Our quest is to find economical ways of adding short-term memory(ies) to this <u>base machine</u>.

A <u>generalized short-term memory cell</u> (STM cell) consists of three fields, some or all of which may be physically realized: a <u>short name</u>, a <u>long name</u>, and some <u>data</u>, as shown in Figure 2. In this model, a set of eight ordinary general-purpose registers would be represented by eight STMs, having short names 0-7, no long names, and one word of data each. An ordinary 1K word cache memory with 4-word lines (i.e. groups of four contiguous words are moved into or out of the cache) would be represented by 256 STMs, each having an anonymous short name (the physical cache memory location), a main memory address for a long name, and four words of data. Other STMs will be presented below.

Figure 2. Generalized short-term memory cell (STM cell):

```
 ---------------------------------------
| short |  long  |         data         |
| name  |  name  |                      |
 ---------------------------------------
```

Our spectrum of discourse ranges from one extreme of ordinary registers, whose use is totally explicit and cmpletely visible to the machine-language programmer, to the other extreme of ordinary cache memory, whose use is totally automatic and intended to be invisible to the programmer. We will examine two middle-range concepts of partly-explicit, partly-automatic management, as shown in Figure 3.

Figure 3. Spectrum of discourse:

```
                            Cache of
 Registers   Renaming     Register Sets    Cache

    |-------------|-------------|-------------|

    <---- explicitly       automatically ---->
          managed          managed
```

II. DESIGN ISSUES.

There are three major motivations for, and three related design issues in introducing short-term memory into a computer design:

1. Fewer address bits.
2. Faster access.
3. Lower bus bandwidth to long-term memory.
4. Load/store instructions for short-term memory.
5. Access to the most recent data.
6. Usage density of short-term memory.

Each of these ideas is briefly explained below.

<u>Fewer address bits</u>. If a frequently-used data item is moved from main memory to a general register, then the register number (short name) can be used to refer to that datum instead of the main memory address (long name). This results in instruction formats that are more compact than formats for our pure memory-to-memory base machine, and this compactness is a major advantage of conventional register-machine architectures. Pure

stack-machine architectures carry this concept even further by using zero instruction bits to specify the top of stack. An ordinary cache memory does not save any address bits in instruction formats.

Faster access. If a frequently-used data item is moved from main memory to a short-term memory location, then access to that datum is often speeded up by a factor of 4-20 (or even 1000 when considering main memory to be a cache on a paging drum). This faster access comes about through a combination of faster circuit technology for the short-term memory cell, shorter wire delays, less bus contention, and simpler address decoding. Small, explicitly-addressed register files will always offer slightly faster access than cache designs, since the cache must always take some time to find a match for the long name presented. Today's fastest cache designs have two-clock-cycle access, while registers often have one-cycle access.

Lower bus bandwidth. If most memory accesses are to the short-term memory, then the bandwidth needed for the long-term memory bus can be substantially lower than in our base machine. This allows a more economical design if multiple processors or I/O devices are connected to a single long-term memory, as in the PDP-11/60 design [1]. This may also allow use of a serial-multiplexed bus instead of a more expensive fully-parallel bus, thus saving wires, pins, or silicon area.

Load/store instructions. If moving data into and out of the short-term memory is done explicitly via software instructions, then two costs are suffered: first, a programmer or compiler must decide where to insert the data movement instructions; and second, instruction bits and time are consumed by these explicit commands. This overhead runs counter to the address bit savings discussed in the first point above. Cache memories neither save instruction bits nor cost data movement instructions. On the other hand, explicit register loads are easy to implement and can be positioned to pre-fetch data so that it arrives at the short-term memory just before it is needed; demand-fetch cache schemes cannot do this. Predictive cache hardware is just beginning to be investigated [2], and has recently been implemented in the Amdahl 470/V8. The address stream presented toa cache can be viewed as a group of interleaved arithmetic progressions. If a simple algorithm can be used to decompose address streams into these progressions, then a cache could prefetch data in each progression.

There is another kind of load/store overhead associated with using a short-term memory: when calling a subroutine, switching tasks, or starting an I/O transfer, it is often necessary to save the current machine state, or to force it to be consistent. This means explicitly saving and restoring all the programmer-visible registers in a machine architecture, and perhaps explicitly copying a cache to main memory, or purging a translation lookaside buffer (TLB) or some otherwise "hidden" short-term memory. As short-term memories become larger and more prevalent, this load/store overhead will become a dominant speed factor. Already, machiner such as the IBM 370 have introduced partial purge instructions to avoid invalidating an entire TLB of 128 entries, and the Cray-1 software has to struggle with trying not to save all 8+8+64+64+512 = 656 registers at every subroutine call or interrupt [3].

THE ADVENT OF LARGE, CHEAP SHORT-TERM MEMORIES DEMANDS BETTER SOLUTIONS TO THE LOAD/STORE OVERHEAD PROBLEM.

Stale data. If a datum is copied to a short-term memory, and then one of the two copies is changed, subsequent access to the other copy will result in fetching stale data. This is obviously a disaster. For a hardware-managed cache memory, simple preventatives for the stale data problem involve notifying the cache of the addresses of all main memory cells changed by an processing or I/O logic in any part of a computer system. This runs counter to the lower bus bandwidth issue above. For a programmer- (or compiler-) managed register, the stale data problem is often prevented by storing the register just before some operation that might access the long-term memory copy, then reloading the short-term memory after that operation. Such operations are surprisingly frequent unless an exhaustive analysis of the program is performed. For example, if a program makes many references to one element of an array, say A(3), then it is desirable to keep a copy of that element in some register. However, any other reference to the same array, such as A(I), potentially accesses the third element, so the register copy of A(3) must be stored before a fetch from A(I), and reloaded after an assignment to A(I). Depending on the language involved, it can take extensive flow analysis on the part of the compiler or programmer just to discover whether a reference to A(I) is possible during the time that A(3) is in a register. For example, in Fortran it is possible that A(3) is kept in a register inside some loop, but the loop includes a branch to some far-away piece of program that changes A(I) then branches back into the loop! If a compiler or programmer is not willing to do this sort of flow analysis, then IT IS NOT POSSIBLE TO KEEP A(3) IN A REGISTER without being exposed to the stale data problem. This is the fundamental reason why simple compilers rarely make effective use of registers, and why many assembly-language programs are difficult to modify by someone other than the original author. A copy of A(3) could be kept in a cache memory with no stale data problem, since the cache monitors all accessed addresses for a possible match, and supplies the most recent data if one is found.

The stale data problem also forces the saving and restoring of almost all registers across a subroutine call on register machines: continuing the above example, the subroutine might refer to A(3), expecting to find the most recent value in its allocated main memory location, not in some register.

There is one more aspect to the stale data problem — aliases. If a long-term memory location can be accessed via more than one name, either because two distinct virtual memory addresses are mapped to the same physical address, or because two distinct high-level language variables in fact refer to the same location (e.g. one is a global variable, and the other is the same variable passed to a parameter), then it is possible that neither a hardware nor a software (compiler) mechanism will detect that an assignment to one name should update a copy of the other name kept in some short-term memory. In virtual memory systems, avoiding this problem involves either prohibiting aliases by software convention, or building cache hardware that compares only physical addresses, not virtual ones. In compiler systems, aliases are either detected through extensive analysis of a program, or no copies of variables can be kept in registers across references to global variables, parameters, pointer assignments, subroutine calls, or a number of other such common occurrences.

AS SHORT-TERM REGISTER MEMORIES GET LARGER, SUBROUTINE CALLS WILL GET SLOWER, UNLESS WE FIND BETTER SOLUTIONS TO THE STALE DATA AND ALIAS PROBLEMS.

The stale data problem in all its forms is probably the hardest design problem to be faced in any system that creates copies of data. The extensive compiler analysis required to take full advantage of fast registers is one reason that cache memories have become so popular -- the hardware substitutes continual address comparisons during execution for compile-time comparisons. Thus, we have a trade-off: for simply-compiled code an N-word cache memory performs better than an N-word register memory, while for carefully-optimized code an N-word register memory performs faster (because of the inherently faster access mentioned above), and is simpler to build.

Usage density. If an architecture provides 200 words of short-term memory, but most programs use only 50 of these words, the memory is under-utilized. One "solution" in such a situation is to make the short-term memory smaller, but in the long run the opposite is preferable -- design the software to make effective use of more short-term memory. One example is in order: the Cray-1 provides 64 T-registers, each 64 bits wide with 1-cycle access (compared to 11-cycle access to a random word in main memory). To avoid load/store overhead, some system software uses none of these registers. One compiler that does generate code that uses the T-registers is the Pascal compiler at Los Alamos. It places local scalar variables into T-registers, but the short subroutines encouraged by clean Pascal coding style often have fewer than five such local variables. Thus many Pascal programs use only about 10% of the available short-term registers. For such a machine, we need software designs that use more registers. One such design is described in Section V below.

III. CACHE MEMORIES.

We will briefly summarize how ordinary cache memories fare with respect to the above six design issues. Figure 4 shows a cache memory as an STM cell associating a long name with some data.

Figure 4. Cache memory as an STM cell:

```
 - - - - ------------------------------------
 | short |   long   |         data           |
 | name  |   name   |                        |
 - - - - ------------------------------------
```

Address bits. Cache memories save nothing in instruction formats.

Access time. Faster than long-term memories, but not quite as fast as registers built out of identical circuits.

Bus bandwidth. As effective as registers with the same load/store characteristics. Often cuts down bandwidth by a factor of 10 (see e.g. [1]).

Load/store overhead. No instruction overhead, except for rare "purge the cache"-type instructions.

Stale data. The forte of cache design -- once the virtual address alias problem is dealt with, cache memories completely solve the software-level alias problem.

Usage density. This is also a strong point of cache systems -- blindly doubling the size of a cache will usually have a much better performance improvement than blindly doubling the number of equivalent registers.

IV. REGISTERS.

We will briefly summarize how ordinary register memories fare with respect to the above six design issues. Figure 5 shows a register memory as an STM cell associating a short name with some data.

Figure 5. Register memory as an STM cell:

```
 --------- - - - - - ---------------------
 | short |   long   |        data         |
 | name  |   name   |                     |
 --------- - - - - - ---------------------
```

Address bits. The forte of register designs -- instruction formats shrink.

Access time. The simplicity of explicitly and directly addressed registers gives an inherent speed advantage over caches.

Bus bandwidth. Similar to cache in cutting down data accesses.

Load/store overhead. On many register machines, 25% or more of all instructions are Loads or Stores (see [1,p.351] or [4] for examples). The instruction bits for these must be balanced against the address bits saved in other instructions. Data may be pre-fetched.

Stale data. No hardware or execution time is "wasted" in trying to detect stale data, but effective use of registers demands compile-time analysis.

Usage density. Again, careful compile-time analysis is needed to take advantage of more registers. Changing assembly language code to use more registers cannot usually be done automatically.

V. TECHNIQUES FOR EFFECTIVE USE OF LARGE SHORT-TERM MEMORIES.

Notice how complementary the above two lists are (compared to our base machine):

	cache	registers
address bits	-	+
access time	+	+
bus bandwidth	+	+
load/store	+	-
stale data	+	-
usage density	+	-

Can we find some way to use the best features of both schemes? Are there techniques that are a merging of the two extremes? How can we trade-off compile-time analysis vs. run-time analysis? Some possible solutions are discussed below.

Renaming. We can separate the idea of short names from the idea of fast access by defining a RENAME operator: RENAME X,Y means that the short name X will be used to access the long name Y until another RENAME involving the same X occurs. RENAME is like LOAD of a register in that subsequent accesses to Y can use just the short name, but it differs from a LOAD because no data movement is implied. Hence, we get the short name without necessarily getting faster access. So what is the advantage of RENAME over LOAD? Figure 6 shows a RENAME mechanism as an STM cell associating a short name with a long name. A similar instruction was implemented for the index registers of the IBM 7030 (Stretch) [5].

Figure 6. Rename memory as an STM cell:

```
-------------------- - - - - - - - - - -
| short |  long   |        data         |
| name  |  name   |                     |
-------------------- - - - - - - - - - -
```

First, RENAME can be implemented in conjunction with a cache memory, such that RENAME gives strong hints to the cache to load (or pre-fetch) the data at location Y. This restores the speed improvement of LOAD. Second, no explicit STORE instruction is associated with RENAME -- the use of the short name X instead of the long name Y is just discontinued at some point in a program. This saves a little instruction space, and it means that, for example, a compiler does not have to do the flow analysis to detect all branches out of a loop in order to find all the places to insert the STORE X,Y to match a LOAD X,Y at the beginning of the loop. Third and most importantly, a compiler does not have to do any alias analysis. As discussed above, when using LOAD/STORE to keep a copy of Y in register X, all other references to Y must be found and changed, or X must be appropriately restored to Y and refetched around any constructs that potentially touch Y. With RENAME, the implementation must ensure that references using the short name X and the long name Y both access the same actual data. Under these circumstances, use of the short name X does not require any flow analysis to find other uses or potential uses of Y.

Cache of register sets. Consider a machine with 16 general registers in its architecture. These registers are normally saved in main memory when calling a subroutine, and reloaded from main memory when returning from a subroutine. As discussed above, we desire to build machines with many more than 16 registers, but we don't want to slow down all subroutine calls. Assuming that almost all registers are in use at the point of call, and almost all will be used by the subroutine (so that we cannot avoid some sort of save/restore), then one way to speed up the call linkage is to have duplicate register sets. Say there are four sets, 0-3, and that the calling subroutine is using set 1. Then the called routine just starts using set 2, and no data movement of set 1 to main memory is needed. This makes the subroutine call quite fast, and it also makes the linkage overhead no longer proportional to the number of registers. When the subroutine returns, the machine just switches back from set 2 to set 1.

There are two flaws in the above scheme that need fixing. First is the obvious question of how to do the fifth nested subroutine call. The answer is that after switching from register set 1 to register set 2, a cache-like mechanism is needed to dribble-back register set 1 to the place in main memory that it would have gone in the simple machine. Dribble-back means that the requisite 16 STOREs are queued at a low priority, so that whenever the running subroutine (using set 2) does not need a bus cycle to main memory, one of the queued stores is done. After the first 16 unused memory cycles pass, all of register 1 is properly stored in main memory, so more nested subroutine calls can reuse that register set. This scheme stands in stark contrast to existing machines that provide multiple register sets, such as the RCA Spectra 45 (IBM 360-like), or some models of the PDP-11, which have four register sets, but they have dedicated uses (operating system, kernel, real-time interrupt, and all user code is a typical allocation of the four), and have no automatic recycling of data to main memory.

The dribble-back technique also stands in stark contrast to the usual STORE MULTIPLE of registers at time of call, because the called subroutine need not wait until the stores finish before starting its execution. In fact, by making the priority of dribble-back stores lower than that of other stores, the register saving always uses otherwise wasted bus cycles, i.e. the register saving is completely free in terms of execution time. Since register save/restore is already a significant overhead on many machines with general registers, and since the trend is toward more registers and more short subroutines as a programming style, dribble-back will become even more significant for saving subroutine linkage time. The Amdahl 470/V6 already uses a form of dribble-back to implement the PURGE TLB instruction (which must invalidate all 128 locations in the virtual address lookaside buffer) [6]. The implementation involves a duplicate set of VALID bits for the TLB, so the PURGE TLB instruction simply switches to the other set, which has previously all been set to "invalid". During the next 128*3 machine cycles, each bit of the just-used set is changed to "invalid". So long as two PURGE TLB instructions are separated by at least that many machine cycles, the implementation is extremely fast (in direct contrast to the IBM 370/168 implementation). This matches the operating system software, which only rarely executes a PURGE TLB. If a second such instruction is issued too soon, the 470/V6 CPU just waits until the previous invalidation cycle is finished.

A subroutine return can simply start using a previous register set, unless that set was dribbled-back to main memory then overwritten with registers for a more deeply nested subroutine call. In this case, the registers need to be reloaded from the data in main memory. In general, it is easy to keep a COPY bit associated with each register set, such that the COPY bit is on if the register set is an exact copy of the corresponding data in main memory. The copy bit is turned on when the last register has been dribbled-back to main memory, and it is turned off again if a nested call reuses that set. It is also turned on when the last register is reloaded from main memory. Then subroutine calls and returns can just use a register set if its COPY bit is on, and must wait for the main memory data movement to catch up if the bit is off.

Consider a deep subroutine nest of A calls B calls C calls D calls E calls F, with four sets of registers. A uses set 0, B set 1, C set 2, D set 3, and E uses set 0 after all of A's data is dribbled-back to memory. Similarly, F uses set 1 after B's data is saved. When F returns to E, it is possible to start reloading B's data, so that when C is later ready to return to B, there will be no delay. On the other hand, if the very next instruction after F's return to E is a call from E to G, set 1 is needed for G to use, and any of B's data loaded into set 1 is wasted effort.

The thoughtful reader will have noticed that we are just running a top-of-stack buffer for a stack of register sets. For such buffers, an amount of hysterisis is useful: once a register set is stored, do not reload it immediately. Instead, wait until the probable time to reload matches the probable time until the reloaded data will be needed. In the case of nested subroutine calls above, we would like to start reloading B's registers into set 1 exactly 16 main memory cycles before C returns to B (assuming no outside access interference). In general, we cannot exactly predict when to start reloading B's data, but we can perhaps safely wait until E returns to D and D returns to C. Similarly, we could apply hysterisis at the other end of the buffer by not even starting the stores of A's registers until 16 cycles before D calls E, i.e. until just before that register set will need to be reused by a deeper subroutine.

The major effect of introducing some hysterisis along with multiple register sets is that we diminish then needed bandwidth to main memory. In fact, instead of asking for a given bus how much bandwidth must be supplied, the computer designer could ask "here is a fixed bandwidth: how much short-term memory and hysterisis must be supplied in order to exceed that bandwidth only rarely?" If we delay storing a subroutine's registers until, say, two more levels of subroutine call have been done, then we never even bother to save registers of a subroutine that only calls one level down then returns. For a software system that rarely nests calls three deep, it would be possible to run for hours without spending any time or bus bandwidth saving and restoring registers, yet the occasional call chain that is 12 deep is handled gracefully, and never with more data transfer than the simple scheme with only one register set.

A few paragraphs back, we mentioned two flaws. The second flaw is that after subroutine A calls B, but before A's registers are dribbled back to main memory, B may try to fetch from the place in main memory that is supposed to contain one of A's registers. Alternately, after A's registers are all safely copied to main memory, B may change the contents of one of those memory locations. If B then returns to A without calling anyone else, the simple description above would have A use the stale data in its register set, without ever reloading the changed word in main memory. The solution to this flaw involves using standard cache techniques: the unused register sets that contain copies of main memory data are exactly cache locations, and all accesses to the corresponding main memory locations must update the cache also. Thus, four register sets look like a four-line cache, with a main memory address tag associated with each line (register set), and with an associative lookup of these four tags whenever main memory is referenced. This scheme effectively ties together the two ends of our short-term memory spectrum.

VI. CONCLUSIONS AND FUTURE RESEARCH.

Registers are simple to build, fast, and small numbers of them are easy for programmers and optimizing compilers to use effectively. Cache memories are more complicated, but easier to use. Providing many registers is an attractive way for the hardware designer to use VLSI technology to support economical short-term memory. Providing a combination of hardware alias resolution and stale data prevention via cache-like address comparisions, along with many registers, may be the best total-system design for effective use of 1000 or more register locations.

Cached register sets are particularly attractive for implementing fast subroutine calls, but the same ideas also apply to implementation of hardware stacks or queues (contrast the Burroughs 7700, with 32 top-of-stack buffer registers, the automatic saving and restoring of which significantly slows down subroutine calls [7]), and to the implementation of task switching. In the latter case, complete duplicate machine states could be kept in multiple register sets.

One line of future research is to measure existing software to discover how much short-term memory hardware would be useful, and what are the parameters for managing it (for example, carefully gathered statistics on dynamic subroutine call/return activity could help decide an optimum number of register sets, plus the parameters of the hysterisis algorithm).

A second line of research is just the converse -- given a fixed arbitrary amount of short-term memory hardware, how can software be automatically re-done to take full advantage of that amount? If only a few levels of subroutine nesting can be handled quickly, automatic insertion of subroutine code inline at the point of call would decrease the number of levels of call in a software package. If many subroutines have only 5 local variables available for short-term storage and a machine provides 30 short-term registers, then a compile-time mapping of the local variables from six subroutines into one merged activation record [3] could provide a much better match to the machine -- the usage density goes way up, and calls between subroutines in such a group would not need to save or restore registers at all: each subroutine just uses a different five of the 30 registers.

Both lines of research must be pursued simultaneously if we are to take full advantage of the short-term memory architectures that VLSI technology makes economical.

REFERENCES

[1] C.G. Bell, J.C. Mudge, and J.E. McNamara, Computer Engineering, chapter 13, Digital Press, Bedford MA, 1978.

[2] A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies", IEEE Computer, December 1978, pp. 7-21.

[3] R.L. Sites, "An Analysis of the Cray-1 Computer", Fifth annual symposium on Computer Architecture, April, 1978, pp 101-106.

[4] R.P. Blake, "Exploring a Stack Architecture", IEEE Computer, May 1977, pp. 30-38.

[5] IBM Corp., Reference Manual: 7030 Data Processing System, form A22-6530, 1960.

[6] Amdahl Corp., Amdahl 470/V6 Hardware Reference Manual, 1976.

[7] E.I. Organick, Computer System Organization, Academic Press, New York NY, 1973, p. 91, 101.