Pascal and Virtual Memory in a Z8000 or MC68000 based Design Station

Forest Baskett

Stanford University

Abstract

This paper describes a run time environment for Pascal-like languages (Pascal, C, etc.) that fits the hardware characteristics of the Z8000 computer and a virtual memory mapping scheme for that computer and the MC68000 computer that allows for complete mapping of the 8 or 16 megabytes of address space and demand paging of that address space. The result is a powerful and programmable dedicated processor that is to support a high performance graphics design station.

Introduction

This paper is the partial result of a project to design and build a high performance graphics design station with substantial dedicated computing power and programmability. The project was undertaken to support a VLSI design system project that needed such a thing. The latest microprocessors and RAMs made it appear possible to easily carry out such a project. In fact, there are a few difficulties if one wants the result to be a reasonably traditional general purpose computing system.

The first microprocessor considered for this project was the Zilog Z8000 since that was the first available microprocessor with a large address space and a reasonably programmable structure. The next microprocessor considered was the Motorola MC68000. The linear address space of the MC68000 makes it relatively straightforward for the language environment designer to use. The segmented address space of the Z8000 and the corresponding addressing modes means that the language environment designer must make some compromises to get an easy effective use of the machine resources. Since we have worked out one set of such compromises that seems to work well, we will describe them here.

Both of these machines have a supervisor state and a class of priviliaged instructions. However, both of them do not seem to be able to support demand paging. Neither of the processors is designed to save the state information that must be maintained in order to interrupt an instruction before completion (because the memory referenced isn't quite available) and restart the instruction correctly. We describe an unorthodox method for achieving this desirable state of affairs.

Code Generating Conventions and Run Time Environment for the Z8000

The goal of the run time environment is to use the machine resources effectively, especially the addressing modes. The run time structure described here achieves that goal and is suitable for a variety of actual hardware configurations with the Z8000 microprocessor. The machine designers have made a number of unusual compromises in the interests of getting large addresses and short instructions. Those compromises mean that different addressing modes apply to different instructions. This section is about learning to live gracefully with those compromises.

The approach is to match the addressing modes of the Z8000 with the different addressing requirements of Pascal-like languages. The machine has eight addressing modes. Different classes of instructions can use different subsets of these addressing modes. These are described first. A Pascal-like language has eight different types of storage to be accessed at run time. A set of conventions in using the segment structure of the machine results in a match between the storage types of the language and the addressing modes of the machine that promotes efficient machine code. These conventions and their implementation interact with future expansions of the Z8000 in several ways. This section closes with comments on these interactions.

This discussion only considers the segmented version of the Z8000. The conventions proposed for this version of the machine will work well on the non-segmented version of the machine although other conventions might be slightly better on that version. In addition, one important convention proposed is aimed at a run time environment that is suitable for use with or without memory management.

Most instructions of the Z8000 are in a two address format. One of the addresses almost always designates a register. The other address is determined by the addressing mode of the instruction. Thus, if the addressing mode is *Register* then both operands for the common two operand instructions will come from registers. The first operand which is almost always a register is also almost always the result. So the typical instruction can be read "the first operand register (the *destination*) contents are replaced by the result of the operation on the previous first operand register contents and second operand (the *source* which is determined by the addressing mode)". The MC68000 is quite similar in this respect.

The Z8000 has a segmented address space. A memory address is composed of a 7 bit segment number and a 16 bit offset in that segment. A "memory management unit" can map that virtual address into a real address or the trivial mapping can treat that address as a 23 bit real address. The machine has eight addressing modes. These are *Register* (*R*), *Immediate* (*IM*), *Indirect Register* (*IR*), *Direct Address* (*DA*), *Index* (*X*), *Relative Address* (*RA*), *Base Address* (*BA*), and *Base Index* (*BX*).

Register (*R*) mode means that the *source* operand (as opposed to the *destination* operand) is from a register. This gives a short instruction (16 bits) in most cases.

Immediate (*IM*) mode means that the source is a constant value in the instruction. Small integers (0-16) may be "loaded" into a register with short instructions (16 bits). For other instructions or larger constants *Immediate* mode results in a 32 or 48 bit instruction, depending on the size of the constant (less than 16 bits or less than 32 bits).

Indirect Register (*IR*) mode results in a short instruction format where the source is the contents of the memory address specified by the register pair in the instruction. This register pair contains a segment number (in the first register of the pair) and offset (in the second).

CH1491-0/80/0000-0456$00.75 © 1980 IEEE

Direct Address (*DA*) mode specifies a memory address whose contents is the source operand. The address is a full memory address including a segment number and an offset. If the offset is 255 or less the resulting instruction is 32 bits long. If the offset is larger the resulting instruction is 48 bits long.

Index (*X*) mode specifies a direct address in the instruction just like *DA* mode but it also specifies a register whose value is added to the direct address to get the final source address. Note that *Index* mode uses the same mode bit setting as *Direct Address*, the only difference being a non-zero index register specification in *Index* mode and a zero index register specification in *Direct Address* mode. Thus register 0 cannot be used for indexing.

Relative Address (*RA*) mode specifies a constant offset in the instruction relative to the current program counter. The resulting instruction is 16 or 32 bits long, depending on the type of instruction and the size of the offset. This mode is only available in the *load*, *load address*, *store*, *jump*, and *call* instructions.

Base Address (*BA*) mode specifies a register pair containing a full address and a constant offset in the instruction. Instructions using this mode are 32 bits long. This mode is only available in the *load*, *load address*, and *store* instructions.

Base Index (*BX*) mode specifies a register pair containing a full address and another register with an offset. These instructions are 32 bits long. This mode is only available in the *load*, *load address*, and *store* instructions.

The addressing modes that are most useful, because they can be used with the most common instructions, are *R*, *IM*, *IR*, *DA*, and *X*. Mode *RA* is quite useful for jumps and calls, but for variable data references it falls in with *BA* and *BX* as a mode to avoid if there is an alternative. *RA*, *BA*, and *BX* modes should be avoided because they can be used for data references only in loads and stores. The other modes can be used on most instructions. Thus operations that require operands to be accessed using the *RA*, *BA*, or *BX* addressing modes often require more instructions and hence are slower than the same operations using *R*, *IM*, *IR*, *DA*, or *X* modes.

In the run time environment of a Pascal-like language there are a number of addressing structures that can be usefully distinguished. In this discussion, we will refer to three types of data: global data, local data, and heap data. Global data is the statically allocated data in the outer block of a Pascal program. Its size is known at compile time so it can be statically allocated at run time. On some machines and some implementations a base register is used as a part of accessing global data as though it were a dynamic activation record. Note here that this is not strictly necessary. This observation will be used shortly.

Local data is that data in dynamically allocated activation records. A Pascal-like language does not require that an activation record for a procedure be retained after the procedure returns so that activation records can be dynamically allocated from a stack. Local data is normally accessed relative to an activation record pointer, here called a frame pointer for brevity. Heap data is dynamically allocated data that is explicitly created and destroyed and whose lifetime is not tied to procedure calls and returns. For example, in Pascal, the **new** and **dispose** procedures are used to create and release heap data. The storage pool from which heap data comes and to which it is returned is the heap. Heap data is accessed relative to pointers.

Consider now the following types of variables: simple global variables (including records), global arrays, simple local variables, local arrays, simple heap variables (almost always records), heap arrays, and reference (**var**) parameters. Efficiency of access has been shown to be most important for local variables, global variables, and global arrays, in that order, by study of the run time characteristics of existing Pascal programs. So let us begin by studying how to structure the run time environment so that these types of data accesses will use the most convenient addressing modes of the Z8000 well. The result should be efficient and compact programs.

Local variables can be accessed with a constant offset from a full address that is variable (the full address is in the frame pointer and it depends on the dynamic call sequence). The only addressing mode of the Z8000 for which this is apparently possible is the *Base Address* mode. This seems like a disaster since it is one of the less convenient addressing modes and this is our most important kind of access. However, if we are willing to decide at compile time (or load time) what segment any activation record for a given procedure will be in, then we can use the *Index* mode in the following way. In the *Index* mode, a register contains a variable offset for a constant address that appears in the instruction. The constant address has two parts, the segment number and a fixed offset, either short or long. By fixing the segment number of the activation record for a procedure at compile (or load) time we can put that segment number in the instruction, we can put the constant offset of the local variable from the frame pointer in the offset part of the address in the instruction, and we can put the variable offset of the frame pointer itself in the register.

This convention has a number of implications. The first one to note is that the offset from the frame pointer for the local variables must be positive. Thus if we grow our stack of activation records toward lower numbered addresses, the frame pointer will have to be at or near the top of that stack. Secondly, by arranging variables in the activation record (at compile time) we can get near maximal benefit from the short constant offset form of *Index* mode as opposed to the long offset form. A simple arrangement that gets most of the benefit is to put all simple local variables as close to the frame pointer address as possible and to put local arrays as far away as necessary.

A third implication, though less compelling than the first two, is that it would be far simpler if all activation records were allocated in the same segment. We have seen that all the activation records for a given procedure should be in the same segment because that segment number should be in the instructions of that procedure. Having different procedures have activation records in different segments might enable us to have the sum of all activation records be larger than a single segment but it is difficult to know by how much since the allocation of segments would be by procedure, not by activation records. Our need for a large stack of activation records might be due to one heavily recursive procedure, all of whose activation records would be in the same segment. In addition, procedure call and return overhead is lower if all activation records are in the same segment. There is yet another reason for a single activation record stack segment that we will get to shortly.

Simple globals and global arrays must be treated together. Simple globals can be accessed just like simple locals if we have a compile (or load) time constant segment in which to put them and we use a global base register like a frame pointer. However, since the storage is static we can make global arrays more convenient if we do not use a frame pointer for the globals but use *Direct Address* (*DA*) mode instead. Global arrays have a compile (or load) time constant base and a run time variable index. If we do not use a base register for the base of all the globals then the base of any given array can be a constant at run time and we can use the *Index* mode for global array referencing. The alternative, if we were to use a global base, would be *Base Index* addressing, a mode to be avoided if possible.

Thus we have handled our three most important addressing structures quite well with these decisions. There is one important implication of the last decision that should be mentioned. Because global arrays are constant in size at load time, we can allocate them to many different segments as needed and fix up the segment numbers in the programs that reference them. However, we have implicitly assumed that no single linear array is larger than 64K (K = 1024) bytes, the current size of one segment. It would be possible to do the address arithmetic to have arrays span consecutively numbered segements, but the resulting code is unpleasant to contemplate. We could, however, have large multidimensional arrays by using dope vectors and only restricting our lowest level linear array to be less than one segment in length. Deciding that all multidimensional arrays should use dope vectors, even those that totally fit in one segment may, in fact, be a good strategy for the Z8000 because of the relative slowness of the multiply instruction even though such a decision results in *Base Index* addressing.

We now must clean up the (fortunately) less important and (unfortunately) less pleasant addressing structures. Simple reference parameters and simple heap variables are both pointer based and can use the *Indirect Register* addressing mode. Structured reference parameters and structured heap variables (records) must use the *Base Address* mode . Local arrays and heap arrays must use the *Base Index* mode. Since it seems fair to assume that we can't have everything on one chip (yet), this seems like a good compromise. Since it also seems fair to assume that one can't do a perfect job in a fixed amount of time, this compromise is no doubt imperfect.

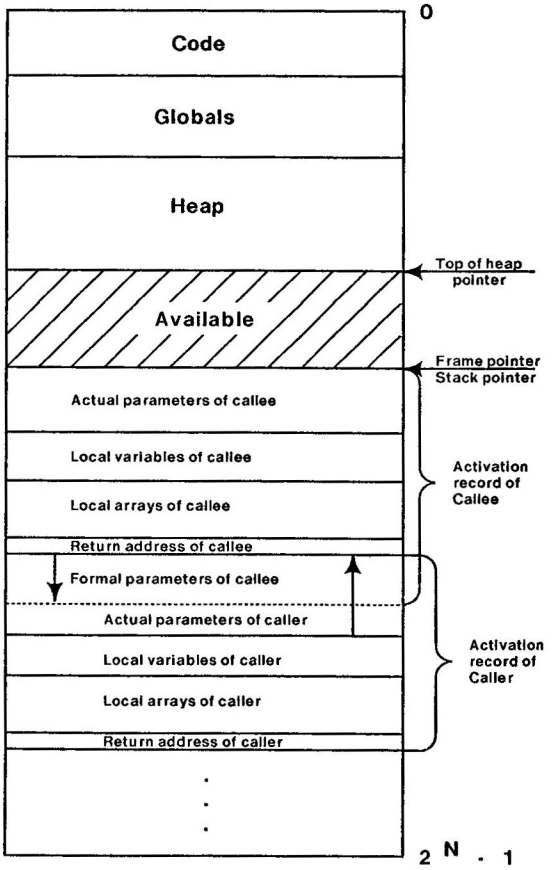
So far, we have not said what segment our activation record should be in and whether this segment should have anything but a parallel relationship to the "stack segment", the one that is manipulated automatically by the procedure call and return instructions. That segment is pointed to by the register pair *R14* and *R15*. Now we did say that local arrays must use the *Base Index* addressing mode and we recall that the base must be a register pair. Since local arrays are in activation records and since we have decided to have that be a constant segment at run time, one element of the register pair used to access local arrays will always have that constant value in it when local arrays are accessed. Since *R14* already has that property, we now have a compelling reason to make the activation record stack and the return address stack share the same segment. However, the return address stack is grown toward lower numbered addresses by the machine so we are obliged to grow our activation record stack in that direction also.

Since local variable offsets must be positive with respect to the frame pointer, consider the following scheme that makes the frame pointer and the stack pointer the same. The frame pointer (register *R15*) points to the last piece of storage used by the current procedure (the very top of the stack). A procedure call automatically decrements *R15* and pushes the return address onto the stack. The called procedure begins execution and its first action is to subtract the size of its activation record from *R15*. Local variables can now be generated and used. The last thing the called procedure does before executing a return instruction is to add the size of its activation record back to *R15*. Now a possible layout within an activation record, going from the frame pointer toward higher numbered offsets, is as follows: (1) actual parameters for any procedure that might be called, (2) simple local variables, (3) local arrays, and finally (4) the formal parameters from the calling procedure. The activation record of the called and the calling procedure now logically overlap in the parameters that are passed by the caller to the callee. This means that the area nearest the frame pointer has to be big enough to hold the longest parameter list that might be passed to another procedure and it means that any temporaries that a procedure needs must use preallocated space in the activation record like any other local. It is not convenient to make dynamic space for temporaries out past the stack top with this scheme. However, variable length parameter lists (for a language like C) fit into this scheme nicely.

This author would use the parameter list area in the above scheme only for parameter list overflow. I would reserve three register pairs for parameter passing and use the activation record only for parameter lists that wouldn't completely fit in three register pairs. Again, measurements of programs indicate that most procedures need three or fewer parameters and there is considerable speed to be gained in passing parameters in registers.

Now that we are growing the activation record stack toward lower numbered addresses, we can consider exactly which segment number should be assigned to that stack. If we assign the number 127 (the highest numbered segment) and if we allocate code, globals, and the heap into low numbered segments, then we can use our resulting programs on machines without memory management and with a memory size that is an even power of 2 just by throwing away whatever high order address bits come out of the machine that we can't use. We have just converted our segmented address space into a linear address space with the stack and the heap growing from opposite ends toward the middle. We have used the trivial address mapping. If our memory size is not an even power of 2 and we still do not have a memory management unit, then we can just watch for an all 1's segment number (a seven input AND gate) and do the appropriate fix up when that condition arises. Thus our run time environment can be used on modest systems without change from the environment used on large systems.

The MC68000 can use the same basic structure to advantage although the details of code generation and variable accessing are much more straightforward and there are no architectural biases toward 64K segments. The activation record stack can be as large as memory or mapping allow and arrays can be any lenght as well. The following diagram illustrates the proposed run time organization.

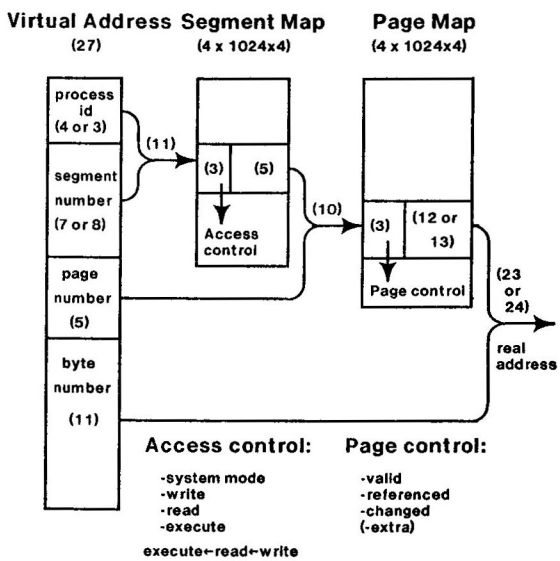


Run Time Environment

Virtual Memory and Demand Paging for the Z8000 and the MC68000

The large address space of these two microprocessor can most effectively be used if we provide some kind of memory mapping of a large virtual address space into a smaller physical address space. The next diagram illustrates a scheme for providing a complet map of the address space of either of these computers. It relies on 8 high speed 1024x4 stactic RAMs such as Intel's 2148's. There is some additional logic not shown in the diagram, of course. However, it is important to note that this scheme may compete effectively in board space with the large package memory management units being planned by the manufacturers of these microprocessors when one wants an equivalent amount of simultaneous mapping as this scheme provides. This scheme is, of course, much faster than the MMUs planned by the manufacturers.

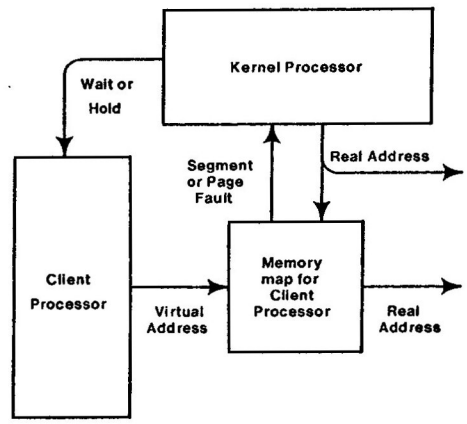
This structure is a two level map with no adders and no comparators. It achieves denisity and speed by just using bipolar RAMs in this two level scheme. The first level can be thought of as a segment map. Entries in this 2K by 8 map indicate if a segment is currently mapped and, if so, which part of the page map contains the page table entries for the segment. The real, complete address map is contained in main store and this map is just a translation look aside buffer. The use of an *active process id* thus means that several active processes can be simultaneously mapped so that most process switches do not require reloading of the entire active map. Thus we achieve virtual memory.



Virtual Memory Mapping Scheme
Z8000 or MC68000

To achieve demand paging, we must stop an instruction that references an unmapped segment or an unmapped page, map the segment or fetch the page, and then continue the instruction. As we noted earilcr, these two microprocessors cannot be interrupted and restarted correctly without a bit of external help. The external help we propose is another processor. The essence of this scheme is that we treat a segment fault or a page fault as a very long memory reference for what we here call the client processor. We simply assert the Wait or Hold line for that processor while our Kernel processor comes into action and fixes up the map and the main store. This is illustrated in our last diagram. This achieves demand paging for a single process.

Note that we are not able to switch processes during a segment or page fault because we are using the client processor to hold the state information necessary for restarting the faulting process. We can only switch processes in response to other kinds of interrupts. While this is not perfect, it does allow us to run a large virtual memory process on our dedicated system. A number of other considerations in our system design indicate that we can easily live with this limitation, especially since it puts us well ahead of planned manufacturer systems (at the moment).



Dual Processor Demand Paging Scheme

Summary

We have designed a Pascal environment and virtual memory system with single process demand paging for the Z8000 and the MC68000 to be used in a dedicated graphics design station. We have implemented the Pascal compiler for the Z8000 based on this scheme and we are beginning the implementation of the Pascal compiler for the MC68000. We are also beginning implementation of the memory mapping scheme. If the results are sucessful, we will report on them in the near feature.