## WORKING OF MMU

*World Wide Web editor's note:* This discussion of the MMU is still appropriate today. Change the processor name from 64180 to Z180, as Zilog now dominates this market, and change Archimedes (the compiler guys) to IAR, and most of the article is still correct.

Programs are getting big! Part of today's shift towards 16 and 32 bit processors comes from the need for correspondingly huge address spaces, since conventional wisdom holds that a 512kb program just cannot fit in the 64k address space of most 8 bit CPUs. Where performance is the overriding concern, a 32 bit CPU may be the only solution. It does seem a shame to abandon all the accumulated knowledge and code gleaned from two decades of 8 bit microprocessors just to get more programming elbow room.

For the past few years some 8 bit CPUs have been equipped with memory management units (MMU) that free programs from most memory limitations. It's tedious and complex to control a MMU manually; now, many languages and other tools include built-in MMU support.

#### Logical -> Physical

The problem of memory management is easy to define: we need some way of connecting lots of memory to a processor that just cannot handle or address it. For example, we might want to put 512kb on a Z80. Since the Z80 only generates 16 bit addresses, it can only directly address 64k of RAM. Somehow, though memory management, we must expand this capability.

For now let's assume that magic hardware gives us more address lines. Perhaps it is as simple as an I/O port loaded by the CPU with an extra upper 8 address lines (A16 to A23), giving us a potential address space of 24 mb. Or, it can be hideously complex, providing some ability to access different sections of address space in wild and wonderful ways.

In any event, as soon as some external mechanism is added to translate addresses in some fashion, the programmer suddenly must contend with two very different sorts of address spaces.

"Physical" memory is that actually connected to the hardware. For example, the 512kb we attach to the sadly-overloaded Z80 is physical memory. Its address ranges from 00000 to 80000 hex in a linear manner.

"Logical" memory is the memory currently located in the processor's address space. Obviously, if the computer can only issue addresses in the range of 0000 to FFFF (0 to 64k), then some of the physical memory is visible and some is not. As the code changes the memory manager's settings different memory becomes visible. That which is addressable at any time is the logical memory.

Thus, addresses generated by the program are always logical addresses - they get translated by some yet undefined hardware into real physical addresses.

So, at one time address 1000 logical might be translated into 28000 physical. Later, in the same code, 1000 could correspond to 80000 physical. The old one-to-one mapping of addresses we're all familiar with is gone!

In summary, addresses used by the code are logical; the memory array sees physical. Between these two the memory management unit (MMU) falls.

**Standard Architectures**

Several years ago Hitachi introduced the 64180, a high integration version of the venerable Z80. While other vendors were trying to push new proprietary architectures, Hitachi took what might seem a step backwards towards the Z80. They realized an important fact of the industry - customers had a fortune invested in Z80 code and were unwilling to switch to an incompatible instruction set.

The 64180 is a Z80 at heart. The designers resisted the temptation to add fancy new instructions and addressing modes that could have made it incompatible with the Z80. Rather, they integrated timers, serial ports, and DMA controllers onto the chip. Even better, they added a memory management unit to translate 64k logical addresses into a 1 mb physical address space.

Now Hitachi sells several other versions of the part. The 64180S is designed especially for telecommunications. The 647180X is a microcontroller version, containing a 64180 core, ROM, RAM, and parallel I/O. Zilog stepped into the act, offering the Z180 (a second source of the 64180) and Z280, a very high performance Z80 upgrade. Zilog is just now announcing the Z181 and will soon offer a microcontroller version of the part, probably a 647180X look-alike.

The most important peripheral on the 64180-family processors is the memory management unit (MMU). The MMU is a hardware device built onto the processor's silicon. The MMU translates every memory address from 16 to 20 bits.

The 64180's MMU uses three internal control registers. In keeping with the chip's design philosophy, on reset the MMU gives a straight logical to physical mapping, simulating the Z80 and, of course, limiting the address space to 64k.

You can divide the 64180's logical address space into one, two, or three areas. The logical space itself is unaltered; even when divided it is still a contiguous 64k.

CBAR is an 8 bit I/O port that can be accessed by the processor's OUT and IN instructions. The lower 4 bits specify the starting address of the bank area, and the upper 4 give the start of common 1. These bits determine the upper four bits of the address. If CBAR were A080, then the base area starts at 8000 logical, and common 1 starts at A000.

Common 0, if it exists, always starts at logical 0000 and runs up to the bank area. The bank area then runs to the start of common 1.

Therefore, you can always understand the logical address space by examining the contents of CBAR by itself. No other information is needed.

The logical address is only part of the problem. How does logical space get mapped to physical? Two other ports provide the rest of the answer.

BBR (the Base Area Bank Register) specifies the starting physical address of the base area (remember, the logical start is in CBAR). CBR (Common Bank Register) provides the same information for common 1. Both of these specify the upper 8 bits of the 20 bit physical address.

A simple formula gives the translation from logical to physical address for the bank area:

```
Physical = Logical + (BBR * 4096)
```

The same formula gives Common 1:

```
Physical = Logical + (CBR * 4096)
```

BBR and CBR gives the upper 8 address bit only - hence the 4096 multiplier. The lower 12 bits come from the logical address. Thus, the translation only affects the upper 8 bits; the lower 12 physical bits are always identical to the lower 12 logical.

On reset, the 64180 sets CBAR to F0, and CBR=BBR=0. This maps logical to physical exactly, with no translation; the bank area starts at logical 0 and common 1 at F000 (since CBAR=F0), the bank area physically starts at 0000 (BBR=0), as does common 1 (CBR=0). If the logical address is 1000, then the MMU allocates this to the bank area (CBAR=F0; 1000 is less than the start of common 1 at F000), and adds the physical base of bank to it (0), giving a translated address of 01000. Similarly, logical F800 is in common 1, and translates to 0F800.

The most important point that can be made about the MMU is that it does not provide the 1mb linear address space we all crave. After all, Z80 instructions use 16 bit address operands and 16 bit register pointers - there is no way to address a number larger than 64k. A jump instruction will always have an argument that is 16 bits long - the logical destination address. The MMU translates this logical address to a possibly large physical number, but the software still operates in a 64k space.

This has a subtle implication - logical address space is a valuable commodity that must be conserved. Wasting physical memory isn't so bad, since the 64180 can deal with up to 1mb. As an example, suppose that your program will have three banks (COMMON 0, BANK, and COMMON 1). If the program is large you might want to bank it in and out of the BANK area, leaving COMMON 1 for data. If BANK is too large, you could be left with little data space - it is important to make BANK as small as feasible to maximize the (in this case) unbanked data.

**Language Support**

Despite the fabulous extra power offered by the 64180's MMU, we've all been making do with Z80 assemblers and compilers. Sure, some claim to support the new processor's extended features, but in truth, until recently, that support has been minimal.

Just what features are important in a 64180 assembler or compiler? Certainly it should be fast, efficient, and all that, but more than anything else the language should give you some sort of way of handling the MMU.

There are two related but different aspects to MMU management. The first is to provide some sort of mechanism to control the MMU with as little programer help as possible. An ideal solution would be a smart compiler that simulates a nearly linear huge address space. The second is to provide output files that contain compiled code and debugging records in some manner that supports current 8 bit tools (like the PROM programmer), but that accounts for the large address spaces.

Taking these two criterion separately, especially with a C compiler we'd really like some method of compiling an ordinary C program in multiple banks. Sure, you might have to tell the compiler or linker about your memory configuration, but ideally the tools should segment and package functions into memory banks as needed. Even better, we'd want it to remap the MMU automatically. Just like working with Turbo C, we would like to be able to invoke a function through a conventional function call, without worrying about its location in memory.

The second requirement is not quite so obvious. How will you burn ROMs for the final project? If the compiled/assembled code exceeds 64k, there may be a problem with using standard Intel hex records for output. Every ROM programmer in the world takes Intel hex input, but the format only supports 16 bit addresses.

One solution is to divide the source program into many separately compiled small pieces. This is especially hard in C, since the linker will not be able to resolve calls between pieces. Another approach is to insure that the compiler or assembler can produce "Type 2" Intel records. Whenever the code crosses a 64k bank the linker could output a type 2 record to specify a new segment address (physical address shifted right 4 bits). This does imply that the linker can handle large physical addresses, and the PROM programmer can accept type 2 records.

Decent debugging files are just as important as useful PROM files. You can't use an emulator, simulator, or monitor to debug the code if the debug records are inadequate. Suppose you wish to display the value of a variable. The debugger must know the physical address of that quantity, since only the physical address is constant. Remapping the MMU changes its logical address, and at times no logical address might correspond to the variable.

This implies that the software packages must maintain both logical and physical addresses for all lines and symbols. Compiling, say, jumps requires logical addresses. All jumps and calls take logical addresses as arguments (since they can only support a 16 bit number). Physical addresses are needed in the debugging records so debuggers can unambiguously resolve the location of symbols, functions, and line numbers, all whose logical address changes with the current MMU setting.