Design and PCB Layout Considerations
for Dymanic Memories
interfaced to the Z80 CPU
by
Tim Olmstead
08-16-96

Interfacing dynamic memories to microprocessors can be a demanding process. Getting DRAMs to work in your prototype board can be even tougher. If you can afford to pay for a multi-layer PCB for your prototype you will probably not have many problems. This paper is not for you. This paper is for the rest of us.

I will break down the subject of DRAM interfacing into two catagories; timing considerations for design, and layout considerations. Since information without applicartion is only half the batttle, this information will then be applied to the Z80 microprocessor.

TIMING CONSIDERATIONS

In this day, given the availability of SIMM modules it would be tempting to concentrate only on these parts. But, to do so would bypass a large supply of surplus parts that might be very attractive to homebuilders. We will then examine several different types of DRAM chips. The main distinction between these parts is whether they have bi-directional I/O pins, or seperate IN and OUT pins. Another distinction will affect refresh. Will the device support CAS-before-RAS refresh, or not?

Let's begin at the beginning. Let's have a look at some basic DRAM timing, and how we might impliment it.

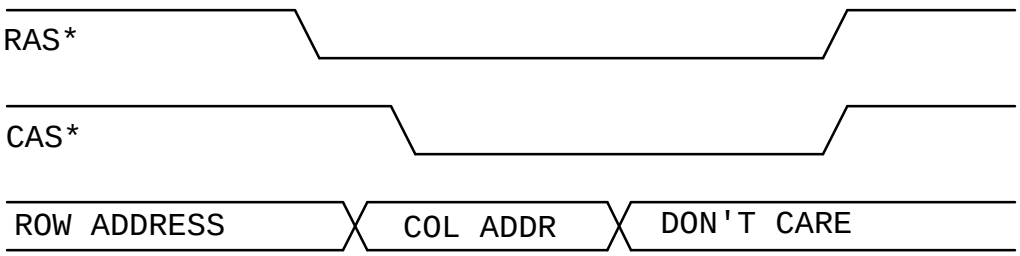


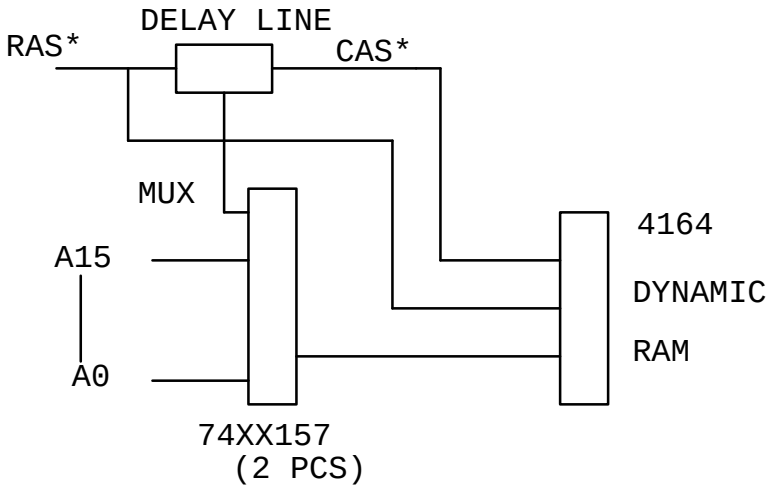
Figure 1. Basic DRAM read timing.

The basic timing diagram for a read cycle is shown in figure 1 above. Two control signale are used to sequence the address into the device; RAS, or Row Address Strobe, and CAS, or Column Address Strobe.

The address is multiplexed into dynamic memories to conserve on package pins. To access a 64K DRAM device, you would need sixteen address lines. Without multiplexing, this would require sixteen pins on the package. That's a lot of pins. By todays standards, a 64K DRAM is very small. to support a modern 16MB part you would need 24 pins. This would lead to some very large device packages, and reduce the number of them that you could place on a single PCB.

Multiplexing the addresses saves on package pins, and allows the device to fit into a muich smaller package, at the expense of a more complex circuit required to operate the devices when compared to static rams. We will discuss a variety of DRAM devices here, but, for now, let's stay with our 64K DRAM. This will be the smallest (in capacity) device we will discuss. It is included here because they are plentiful, and VERY cheap, on the surplus market. This would make them ideal for use in hobbiest projects.

Let us review the timing diagram in figure 1. On the top row of the diagram we see RAS*. This is our Row Address Strobe. Next we see CAS*, the Column Address Strobe. At the bottom we see the address lines that connect to the DRAM chip itself. OK. What is this diagrma trying to show us? First we present the row address to the DRAM chip. Some time later, we take RAS* low, or active. We wait a little while, then switch the address presented to the chip. Now we present the column address. After we present the column address, we wait a little while, then take CAS* active; low. Since this is a read cycle, some time after CAS* goes low, the memory will supply output data. Simple huh? Righhhhht! Ok. So how do we do it? What do we need to



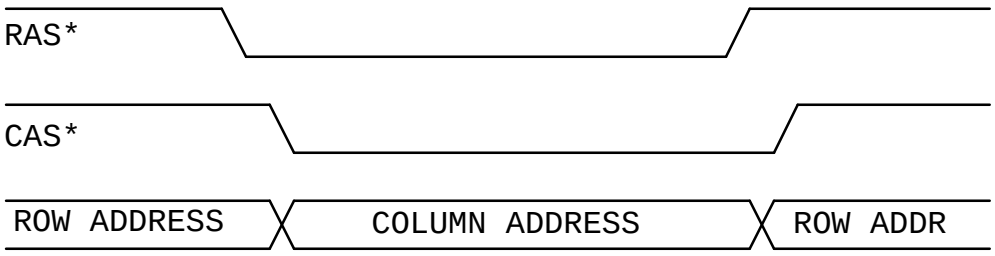
create this kind of timing? The following illustration will give us some hints.

Figure 2. Basic DRAM Timing Generation

In figure 2 we see the basic dynamic memory controller circuit that has been in use since the late 1970's. No, don't go out and grab your wire-wrap gun just yet. This circuit is not complete. It does. however, illustrate the basic elements needed.

The key element in figure 2 is the delay line. This is a special part that will geiv precise delays. You feed a signal into the input, then take what you want at various "taps", or outputs. In the past, delay lines were made from 7404 inverter packages. Sections were connected together to eliminate the inversion, and a coil inserted between sections to give the delay. The delay could be controlled by the number of turns of wire in the coils. Today, silicon delay lines are available. Dallas Semiconductor makes a line of silicon delay lines with very precise control of delays.They are pin compatible with the older mechanical ones, and cheaper too.

The first tap is used to generate a signal named MUX. This signal switches the 74xx157



multiplexers to change from ROW address to COLUMN address. The second tap is then used to generate CAS*. THis circuit will provide the following timing.

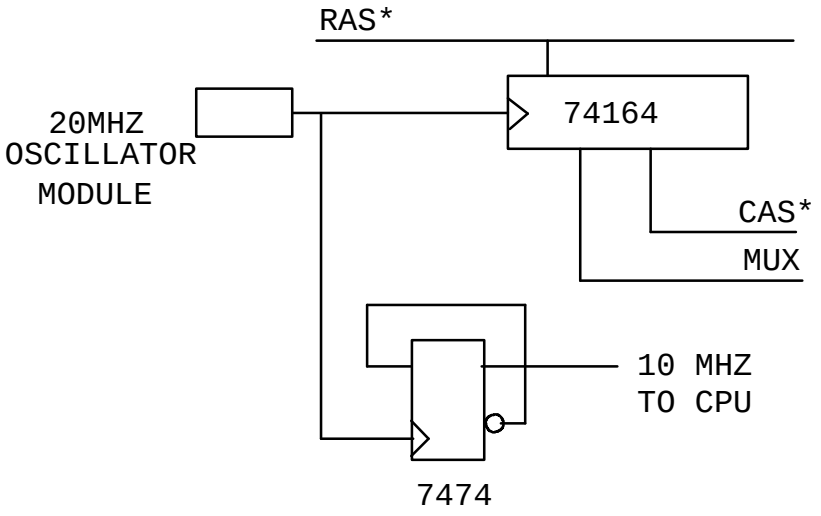
Figure 3. Timing for circuit in Fig 2.

As may be seen in Figure 3, our circuit generates the needed timing fairly well. The astute reader will notice some overlap between CAS and RAS at the end of the cycle. This is not only ok, but some early DRAMs required it; noteably, the 4116, 16k by 1.

Now let's examine a circuit to replace the delay line. If there is a high speed clock available in the design, we can generate the timing with a shift register. This works best if the CPU clock is also derived from this same source. Let's consider a 10 mhz Z80 design. We will use a 20 mhz oscillator module to derive timing from. The timing generation protion of the circuit in figure 2 could look like this.

As you can see in figure 4, we start with a 20 mhz clock source. This clock drives a 74164 shift register, and a 7474 D type flip-flop. The flip-flop divides the 20 mhz signal by two, giving us a 10 mhz clock source for the Z80 CPU. The shift register replaces the delay line in figure 2. It is

continously clocked by the 20 mhz clock. RAS* is presented to the data input of the shift register. When RAS* goes low, the shift register begins to shift zeros. On the next rising clock edge MUX will go low. On the following clock edge, CAS* will go low. THis circuit will generate the exact same timing as figure 3, assuming a delay line with 50 ns taps in the original circuit. The advantage of this circuit is that it uses cheap parts. The disadvantage is that it



requires a high speed clock source. Additionaly, the 10 mhz clock source developed in figure 4 may not be acceptable to the Z80 CPU as is (it most certainly is NOT). Additional circuitry may be required to condition the clock before using it to drive the Z80 chip.

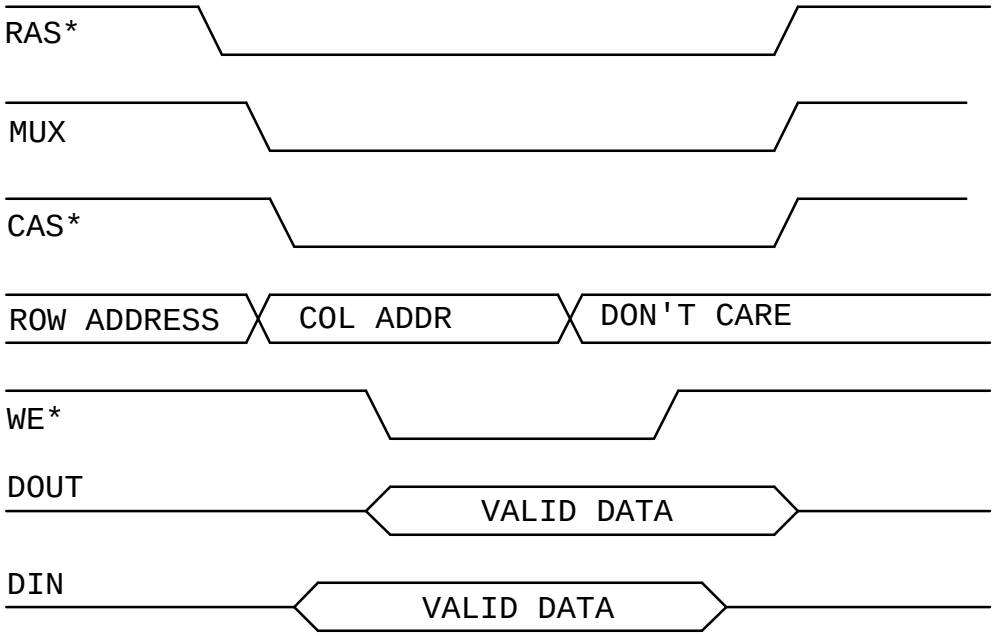
Fig 4. Shift Register Timing Generation.

The main difference between the circuits in figures 2 and 4 are this. The circuit in figure 2 is ASYNCHRONOUS while the circuit in figure 4 is SYNCHRONOUS. The asynchronous circuit in figure 2 may be easier to adapt to various processors while the synchronous circuit in figure 4 is more predictable when you go to make changes to the design. Consider this. You decide to change the CPU speed from 10 to 12 mhz.

At 10 mhz we are using a 20 mhz oscillator module in figure 4. At 12 mhz, we will use a 24 mhz oscillator. At 20 mhz the period, or time from one rising edge to the next, is 50 ns. At 24 mhz, this is now 42.5ns. Thus the delay from RAS to MUX to CAS is now 42.5 ns. Experience tells me that this is just fine. The only thing we have to worry about now is are the DRAMS we are using fast enough to get data back in time? The fact that the timing compresses automaticly when you change the oscillator module will help to speed up the memory cycle; in this case, by 15ns. By speeding up the beginning of the cycle, you have more time for the memory to access. This allows you to run faster with slower memories.

With the circuit in figure 2 you can do the same thing, but you will need to replace the delay line to get there. This could be a consideration when upgrading an existing design.

Well, if we only ever wanted to read our DRAMs, we would be about through. However, such is not the case. How does the data get into the DRAM in the first place? Now I just KNEW you



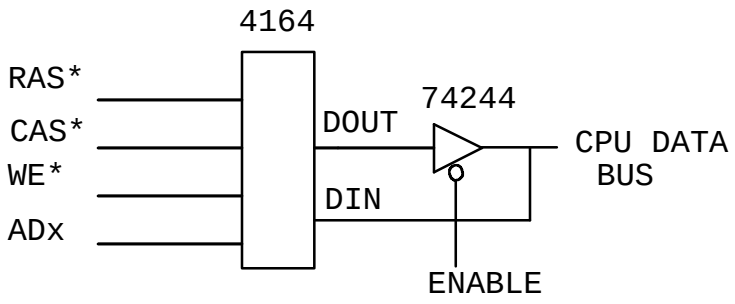
were going to ask that. OK! Let's look at a write cycle. First we will look at a basic write cycle. It is not much in use anymore, but does apply to the 4164 device we are discussing.

Fig 5. Basic DRAM WRITE timing.

In figure 5 we see the timing diagram for a basic write cycle. What is significant in this diagram is that the DRAM device actualy does both a READ and a WRITE. At the beginning of the memory cycle we generate RAS, MUX, and CAS, just as we did for a read cycle. Some time after CAS does low, data is available at the output pin of the device.

The interesting thing in figure 5 is that WE gets pulsed low shortly after CAS goes low. Data present at the data input pin is written into the DRAM when WE goes back high. The data presented at the data output pin will continue to be the old data that was in the accessed location before WE was pulsed.

This type of write cycle is refered to as a read-modify-write cycle in some data books. It can be useful in some designs because it will let you use slower memory than you might have needed for an early-write cycle (which will be discussed next). This is because the data is written into the memory late in the cycle; when WE goes high. For early-write, the data is written into the memory when CAS goes low; which is usualy early in the memory cycle.



Let's examine a design that will impliment this read-modify-write cycle as the standard write.

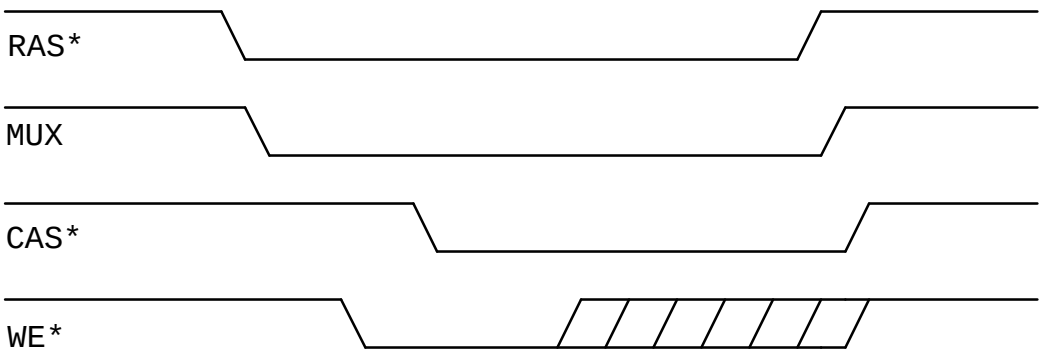
Fig 6. Seperate I/O implimentation.

In figure 6 we see our 4164 implimented for seperate data in and out pins. The key to this circuit is the enable. The 74244 buffer is only enabled during read operations. During writes, this buffer is left diabled. Thus, the data present at it's DOUT pin remains isolated from the CPU data bus. The new data is written into the device by the pulse on WE.

I once used this circuit to impliment a 10 mhz Z8000 CPU card with 150ns. memories, and now wait states. With common early write, it would have required 100 ns memories, and one wait state for writes.

OK. What is early write, and why would I want it. It sounds like it would cost performance. Well, it does. But, we have to learn how to deal with it because all the SIMM modules use it, as do the new byte and word wide DRAMS that are coming out. Seperate I/O is nice, but it uses too many package pins. On SIMM modules, where data may be 8, 9, 32, or 36 bits wide, there is no room on the connector for seperate in and out pins. The same is true on the byte and word wide parts.

So, that said, let's look at early write. On these denser parts package pins are conserved by tieing the in and out pins together and using a single pin as a bi-directional data pin. On some SIMM



modules, they literaly tie two package pins to gether on that tiny printed circuit board. Looking at figure 5 it is obvious that we can no longer use the read-modify-write cycle. It allows the output to be turned on, which would conflict with the data your CPU is trying to write. Not good. What we need is a way to tell the DRAM chip that we realy aren't doing a read, and not to turn its' output on. This would eliminate the conflict.

The way we do this is by taking WE low before we take CAS low. If WE is low when CAS goes low the DRAM will not turn on its' outputs. Yes, there is a catch to it. The data is written into the device AS CAS GOES LOW. This means that you must somehow hold off CAS for write cycles until you know that the data is valid. On some processors this means that you will need a wait state on writes. Since you had to wait till later in the cycle to activate CAS, it may take you longer to complete the memory cycle. How many of your 486 motherboards require a wait state on memory writes? It is very common for just this reason. The timing of an early write cycle looks like this.

Fig 7. Early Write cycle.

In figure 7 we see an early write cycle. Note that CAS is held off until after WE is low. How you will impliment this in hardware will depend on the processor you are using. We said we were considering the Z80 so we will look at how one might impliment this on a Z80. The following circuit should generate the needed signals. It is shown as discrete gates to illustrate the logic. It would be very cleanly implimented in a PAL, or Programmable Array Logic device.

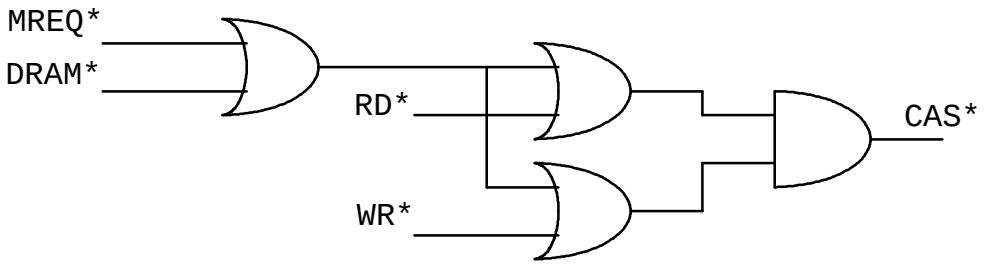


Fig 8. Circuit to generate CAS for Z80.

The circuit in figure 8 will generate CAS for the early write devices. The signal DRAM* comes from the address decoding logic. For read cycles CAS will be generated by the Z80 RD signal. For write cycles CAS will be held off until WR goes active. There will still be other things this circuit must do, so don't get out your wire wrap guns just yet.

What have we left out now? We know how to read and write our DRAM. What's left? Well, there is one more thing; REFRESH. Static memories are made from flip-flops. Flip-flops can remain in a state indefinately, as long as you keep power on them. The problem with static rams is that the die cells are rather large; each flip-flop being constructed with either 2 or 4 transistors.

In dynamic memories, the storage element is a capacitor. Just put a charge into the capacitor for a one, take it away for a zero. The problem with capacitors is that they won't hold their charge forever. At least not without some help they won't. The reason capacitors won't hold their charge is something called leakage. The charge is help on two plates, one with a positive charge, one with a negative charge. The plates are held apart with some kind of insulator, or dielectric. Charge leaks between the plates through the dielectric. Now, wouldn't it be great if we put our program in one of these capacitors, then came back a little later to run it, and it wasn't there anymore? That is exactly what DRAMs woudl do without refresh.

Someone smarter than me decided that if you were to periodicly go around to all of the capacitors and freshen up the charge, that this just might work. Well, it does. To refresh a DRAM you must reference every row address in the device within a specified amount of time.

As DRAM devices get denser, that is bigger, they have more rows in them. The 4164 we've been talking about has 256 rows; it uses 8 bits for the row address. A modern 4MB part has 2048 rows, using 11 bits for the row address. This is eight times as many rows. If we had to refresh all rows in any device in teh same amount of time, then with the 4MB part, we would need to run refresh eight times as fast as for the 4164, just to get through in time.

Fortunately, this is not true. Over the years chip manufacturers have gotten the leakage performance of each sucessive part a little better. Now we can basicly refresh each part at the same rate as the last one. This is good. If we had to keep refreshing faster and faster, we would

soon have no bandwidth left for the CPU to use the memory. We would be using all the available time to refresh it.

OK. How do we do this thing called refresh? Glad you asked. There are two ways of doing it; RAS only refresh, and CAS before RAS refresh. Let's excamine RAS only refresh first.

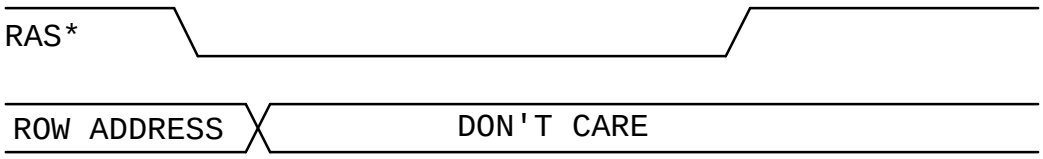


Fig 9. RAS only refresh cycle.

Examining figure 9 we see that a RAS only refresh consists of providing a row address, and strobing RAS. CAS and WE must be held high during this cycle. It is CAS remaining high that tells the device that this is a refresh cycle. In DRAMS it is CAS that controls the output drivers. By keeping CAS high, the output drivers remain off, abd the row which was accessed is refreshed.

Actualy, every read cycle is also a refresh cycle for the row accessed. The problem with normal reads is that they tend to be random. You cannot guarantee that all possible row addresses will be referemceed in the specified time just by executing programs. Therefore, we must refresh the device. The Z80 CPU provides a mechanism for refreshing DRAMs. Unfortunately for us, the Z80 was designed just before the last ice age; when 4116 (16K by 1) DRAMs were popular. Thus, they only furnish 7 bits of refresh address. The intent of this refresh mechanism was to support the RAS only refresh. At that time, that was all we had, and if you are going to work with the 4164, that is what you MUST impliment. CAS before RAS hadn't come along yet. This is a bummer, but we can still use teh Z80's refresh cycle to control refresh, we just have to furnish the address. A RAS only refresh DRAM subsystem may be implimented as shown in the following illustration.

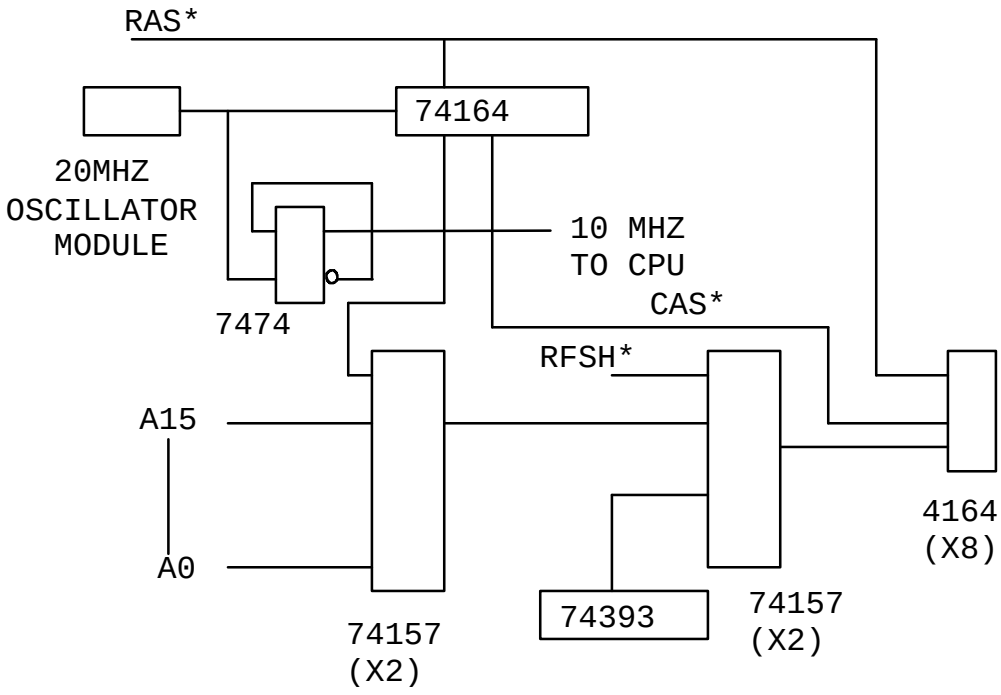


Fig 10. RAS only refresh implimentation.

We are rapidly approaching our promised design implimentation for the Z80. The circuit in figure 10 will impliment a single row of 4164, 64K by 1, DRAMs for the Z80. Don't worry, when we're done, we will draw a MUCH better diagram for you. There are a few control issues left out of figure 10 for the sake of simplifying the drawing.

RAS only refresh was the only thing we had to work with until the arrival of the 256K by 1 devices. With the 256K devices we got CAS before RAS refresh. and NOT ALL OF THEM HAD IT. If you are designing with 256K parts, you should consult the manufacturers data sheet for the parts you want to use to verify that they support CAS before RAS refresh. If not, you must either impliment RAS only refresh, or find some other parts.

Ok. What does CAS before RAS refresh look like?Let's see.

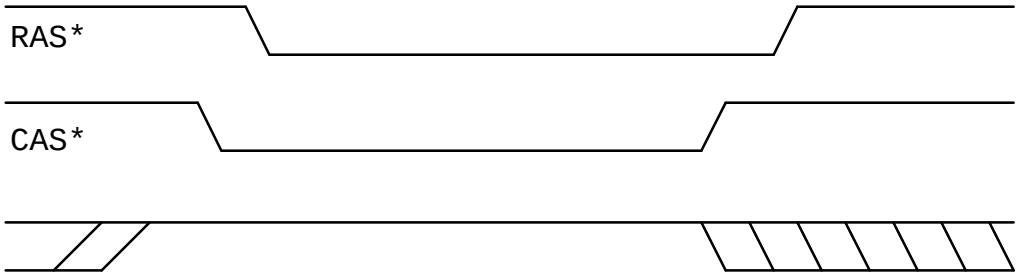


Fig 11. CAS before RAS refresh.

Oh boy. This looks different. We are used to seing RAS go active before CAS. Also, we now don't care about what is on the address lines. WE must be held high during the refresh cycle, and that's it. Done. This realy looks simple, but what does it do for us in hardware? Let's see.

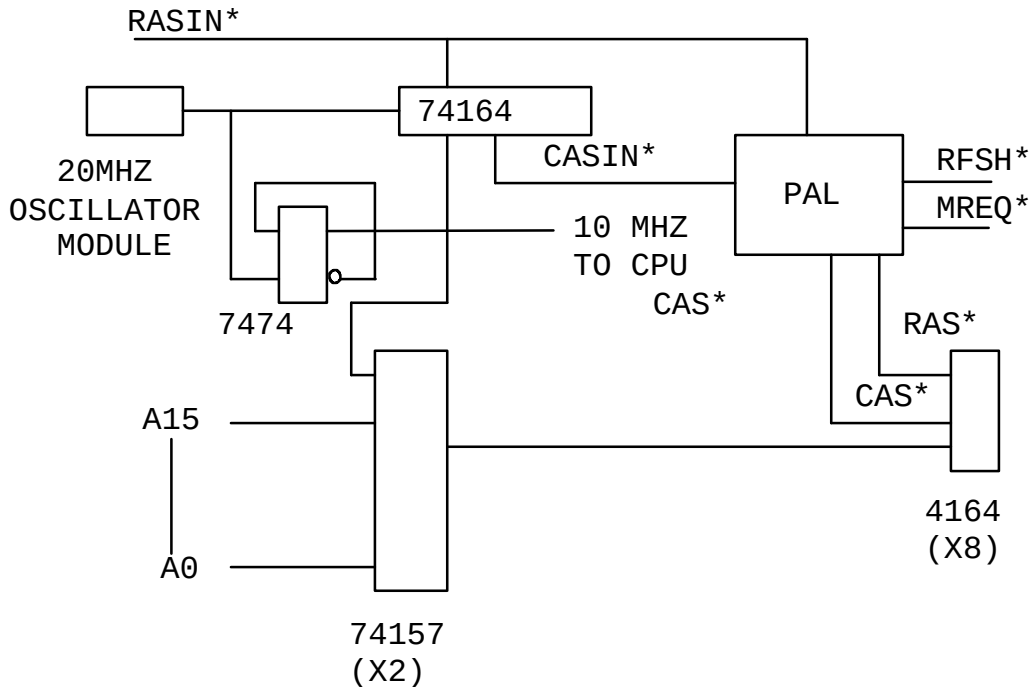


Fig 12. CAS before RAS refresh implimentation.

This looks suspiciously like figure 4. It is, with the addion a a PAL, or Programmable Array Logic, device. At this point, the PAL maked implimentation of this kind of logic MUCH easier. The equations for RAS and CAS in figure 12 would look something like this.

/RAS = /MREQ * RFSH * /RASIN ; NORMAL RAS

```
        + /MREQ * /RFSH * /CASIN   ; REFSRESH

/CAS = /MREQ * RFSH * /CASIN      ; NORMAL CAS
        + /MREQ * /RFSH * /RASIN  ; REFRESH
```

From the above equations it becomes quite clear how CAS before RAS refresh works. We still have our shift register generating the timing for us. For a normal mamory cycle, we pass this on through. But, for a refresh cycle, we swap the outputs. The signal that is normaly RAS goes to CAS, and the signal that is normaly CAS goes to RAS. This impliments the CAS before RAS function very nicely. The processor will hold WR high during a refresh cycle, so there we are. The only thing left for us to do is to add in RD and WR. You did remember that we have to hold off CAS for writes didn't you? Of course you did. The new equations would look like this.

```
/RAS = /MREQ * RFSH * /RASIN    ; NORMAL RAS
        + /MREQ * /RFSH * /CASIN   ; REFSRESH

/CAS = /MREQ * RFSH * /CASIN * /RD      ; NORMAL CAS FOR READ
        + /MREQ * RFSH * /CASIN * /WR   ; NORMAL CAS FOR WRITE
        + /MREQ * /RFSH * /RASIN  ; REFRESH
```

The memory subsystem shown in figure 12 may be implimented with any DRAM device that supports CAS before RAS refresh. With the equations above, you can also support early write and use devices with a bi-directional data pin. Before we move on, let's examine some of these devices that might be of interest.

When trying to build a project with the fewest components we might want to examine some of the denser parts. One such part is the 64K by 4 DRAM. It is/was available from several vendors. It may not be currently being made any more, but you may find them in the surplus channels. I have personaly removed several of then from old 286' machines. with 2 of these parts, you have 64K of memory for a Z80. They are new enough to support CAS before RAS refresh, and the use early write. The decvice looks like this.

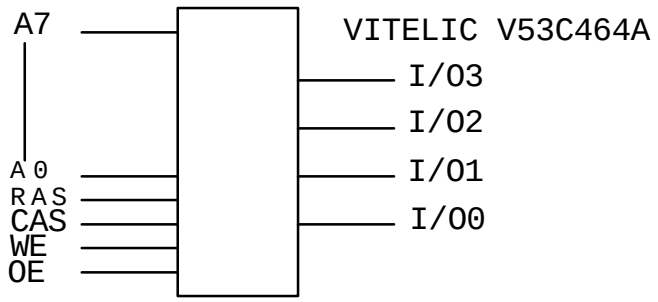


Fig 13. A 64K by 4 DRAM chip.

The chip shown in figure 13 has one pin we haven't discussed yet; OE. This pin may be tied to ground and ignored.This device is realy a 256K bit part internale. They just arranged it as four banks of 64K.

The move to make DRAMs wider than one bit is becoming a welcome trend. There are now parts that are 8, 9, 16, 18 bits wide. Let's look at another device that is 8 bits wide. Perfect for the Z80 except that it is greater than 64K. We will discuss memory management on the Z80 later. The device we will discuss next is the Vitelic V53C8256H.

NOTE : I am using a MOSEL/VITELIC data book for some of these parts because it is what I have handy. Most, or all. of these decices are manufactured by many different memory vendors. Consult the appropriate data book. I have especialy concentrated on the older devices as I fealt that they would be available on the surplus market at good prices. Or, turn over that pile of old XT and 286 motherboards, and see what gold lies there.

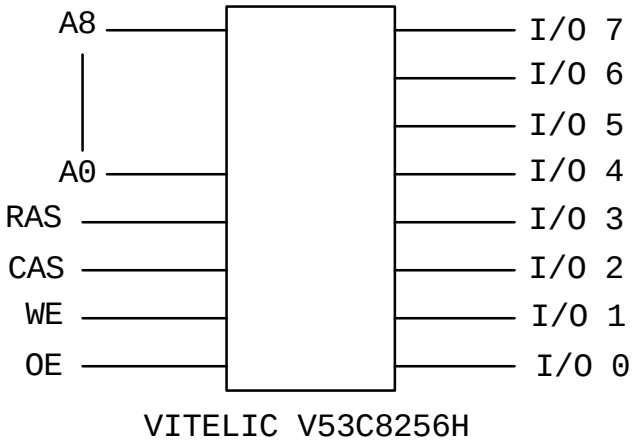


Fig 14. 256K by 8 DRAM chip.

With the chip in figure 14 you would have a 256KB memory system in one chip. This trend goes on with the current highest desnity device being 2M by 8, I believe; and in one chip. Of course these are the current state of the art devices, and you will have to pay real money for them. The older devices can be had for free, or very close to it.

Let's examine one more memory system design issue before we move on to memory management; parity. Should we or shouldn't we have parity? That is a question that only you can answer. It depends on the application. Most applications probably don't need parity, but some do. Medical applications, or anything that needs to be fail safe should have AT LEAST parity, if not ECC. All parity will do is tell you that something happened, not how to fix it.

Parity is a wonderful thing if you are a DRAM manufacturer. You just found a way to sell every customer more of your product. All you have to do is create a panic in the user comunity. Make them believe that their memory is so unreliable that they need this, then you will be able to sell

them more of it. But, if the manufacturers memory is that unreliable, why are we buying it in the first place? OK. I'll get down off my soapbox. If you think you realy need parity, then read on.

What is parity anyway. Well, put simply, it forces the number of buts set to a one across the stored word, including the parity bit, to be either even, or odd. For example, consider that the data on the CPU's data bus is 00001111. To impliment an even parity system, we would store a zero in the parity bit. The byte we are generating parity for is already even since it has four bits set to a one. By storing a zero in the parity bit, we still have an even number of bits set to a one. If we were implimenting an odd parity system, we would store a one in the parity bit for this example. We would then have odd parity across all nine bits of the stored data.

I prefer to impliment odd parity for DRAM systems. This ensures that there will be at least one bit in the group that is set to a one. Very often DRAM will come up with all zeroes in it after power up. If we implimented even parity we could read uninitialized memory, and not detect it.

To add parity to your system you need to add one more ram chip to each byte. Since we are talking about a Z80 processor, and it is only an 8 bit processor, we will add one ram chip to each row of memory. A special circuit manages that extra device. It gets the same RAS, CAS, and WE as the rest of that row of devices, but it's data doesn't come from the data bus. Consider the following.

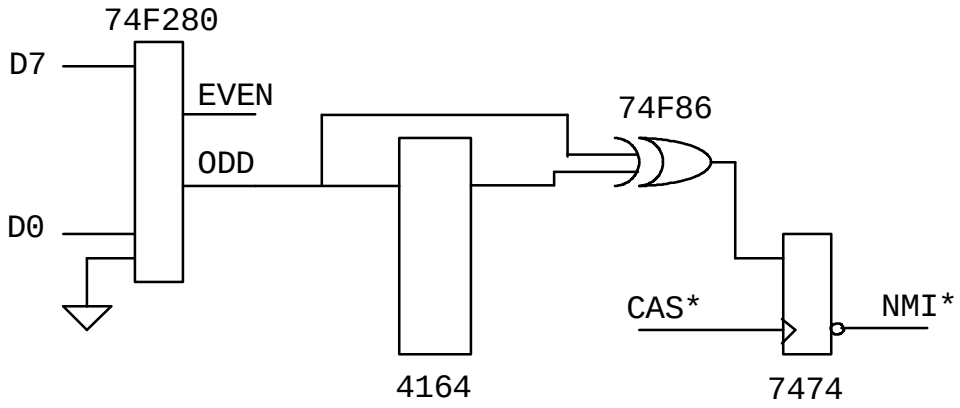


Fig 15. Parity implimentation.

The heart of the implimentation of parity is the 74280 device. It watches the data on the Z80's data bus and continously generates the parity of it. The 74F280 is very fast. It will typicly generate parity in 4 to 5ns. While this is fast we must remember to include this time in our speed calculations when we get to the application of all this theory.

The design in figure 15 uses a part with seperate I/O pins for the partiy device. If we didn't, we would have ti insert a tristate buffer between the memory and the 74F280, then add control logic to decide when to enable it. We would also have another delay between the output of the 74F280 and the memory.

During a write cycle the parity is written into the parity ram. When the data is read back out of memory and placed on the CPUs data bus, the 74F280 generates the parity on the data just read back. The results are fed to the 74F86 XOR gate along with the value read back from the parity ram. if they are both the same there will be a zero on the output of the XOR gate. This value is sampled at the end of the memory cycle when CAS goes back high. If the genereted parity does not agree with the parity stored in the extra ram an interrupt will be generated. System software will then have to figure out what to do about it.

The 30 pin SIMM modules were designed with parity in mind. And here you thought I was going to forget SIMM modules. Let's look at a 4MB by 9, 30 pin, SIMM module.

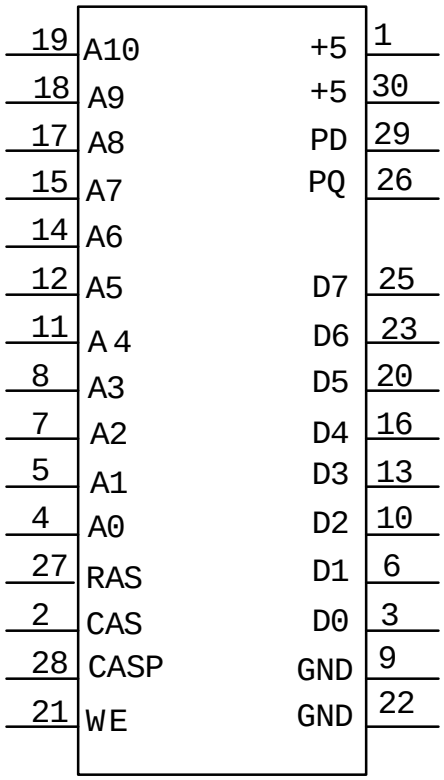


Fig 16. 4MB by 8 SIMM with parity.

Figure 16 is shown as a data sheet because I have seen repeated requests for the pinout of a SIMM on the internet. If you hold the module in your hand with the chips facing up,. and the edge connector facing you, then pin 1 in on the left end. You may treat this module just the same as you would the 256K by 8 device in figure 14.

Note that the 8 data lines are bi-directional, but the parity bit has seperate I/O pins. The parity bit also has a seperate CAS pin. This is usualy tied to the primary CAS pin for the module. If you wanted to delay the write to the parity chip, to allow ,more time for the parity to be valid, you could generate a seperate CAS signal for it. In practice this is usualy not necessary. The parity circuit in figure 15 will handle the parity bit quite nicely.

For a number of reasons 30 pin SIMMs should be seriously considered for any homebrew project. Using a SIMM module may spell the difference between success and not success for your project; especialy if it is hand wired. The SIMM module already hss a PCB with the DRAMs mounted on it. It also has the correct bypass capacitors mounted under the DRAM chips. This gives you a step up on the most difficult part of implimenting DRAMs in a prototype environment; power distribution.

Another reason for considering using 30 pin SIMM modules is that the industry is moving on to the 72 pin modules. it is now fairly easy to find 256K, 30 pin, SIMMs cheap. One surplus store near me has them for $3.95 each. The 1MB and up parts are still in demand, and the price on them is actualy going up. Oh well. That's what supply and demand will do for you.

We will not discuss the 72 pin modules here. They are 32 bits wide. Our stated goal was to interface memory to a Z80 which is 8 bits wide. While we could impliment the module as four banks of 8 bit memory this is kind of escteric and we won't do it. Should I get a flood of requests, we'll see.

APPLICATIONS

Oh boy. Now we get to the fun part. Here is where we try to make it work. We will consider several configurations of memory, but first it might be good to examine the environment we wish to impliment in; the Z80 CPU.

The Z80 is an 8 bit microprocessor. It uses 16 bits for memroy addressing giving it the ability to address 64K of memory. This is not much by todays standards. It is possible to make the Z80 address more memory by adding external circuitry. With this circuitry it would be possibke to make the Z80 address as much memory as we want; say 4GB. A Z80 addressing 4GB of memory might not be quite practical, after all, what would it do with it? However, something a little more down to earth might be useful; say 256K, or 1-4MB.

The first thing we must understand is this. No matter what external circuit we come up with, the Z80 will only address 64K at any given moment in time. What we need is a way to change where in the PHYSICAL address space the Z80 is working from moment to moment. This function is called memory management. THe circuit that performs the memory management is called an MMU, or Memory Management Unit.

Today everyone is probably experienced with running 386MAX, QEMM, or HIMEM on their pc's. This is the memory management software that runs the MMU in our 386/486/Pentium processors. The pc uses memory management for a different function than what we might use it for in the Z80, since the 386/486/Pentium processors are inheriently capable of directly addressing a full 4GB of memory. With the Z80, we need an MMU just to even get at all of the memory we may have in the system.
The basic idea of how a memory manager works is this. There is a large PHYSICAL memory space defined by the amount of memory plugged into the system. If you plugged in 256K of memory, then your physical address space is 256K, and so on. When a memory manager maps

memory for the Z80 processor, the 64K address space of the Z80 becomes the LOGICAL address space.

The logical address space is broken up, by the MMU, into small chunks. The next thing we must decide is how big the chunks. They can be as small as we want. For our Z80's 64K ogical address space we would need 128 pages in our MMU to impliment this.

If we are building a multi-tasking system some or most of these MMU pages may need to be rewritten each time we have a task switch. This greatly increases system overhead. We want the task switch to be accomplished as fast as possible. The code we execute during the task switch doesn't contribute to the running of our application task. It is just overhead.

We would also need to design hardware that could provide that many pages in our MMU. We could certainly do this, but it would increase our chip count, and the MMU may not be fast enough for our needs.

Ok, 512 bytes per page is too fine for our needs. Let's look at 4K pages. Again, for our Z80's 64K logical address space, we would now need 16 pages. This sounds a lot better. Very fast hardware register file chips are available with 16 registers, that will meet our needs; the 74xx189. The 74xx189 is a 16 by 4 register file chip. You can stack them to get any width you need.

As we said earlier, if we are using 4K pages, we will need 16 of them to accomodate the 64K logical address space of the Z80 CPU. To address 16 pages in our external MMU we will need four address line. We will use the uppermost address lines on the Z80. The block diagram of our MMU is shown in the following illustration.

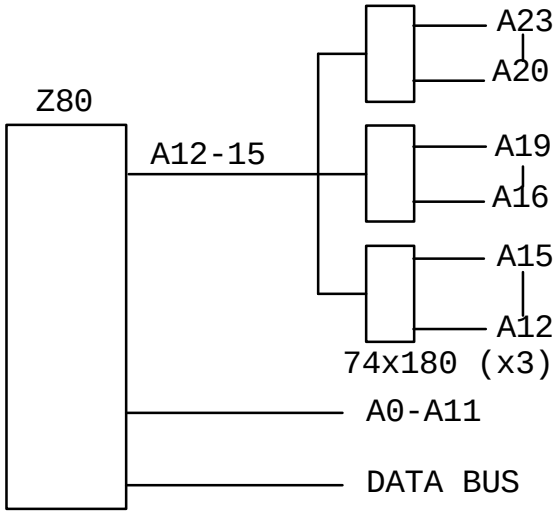


Fig 17. Z80 CPU with external MMU.

Figure 17 shows the basic Z80 CPU implimented with an MMU. The MMU is made from three 74x189 register files. These 3 parts develop 12 address lines. When put with the low order 12 address lines from the Z80, we have 24 address lines, enough to address 16MB of memory. If we limited our design to 1MB of ram we could eliminate one of the 189's and simplify the control software somewhat. For the rest of our discussion we will assume three 189's.

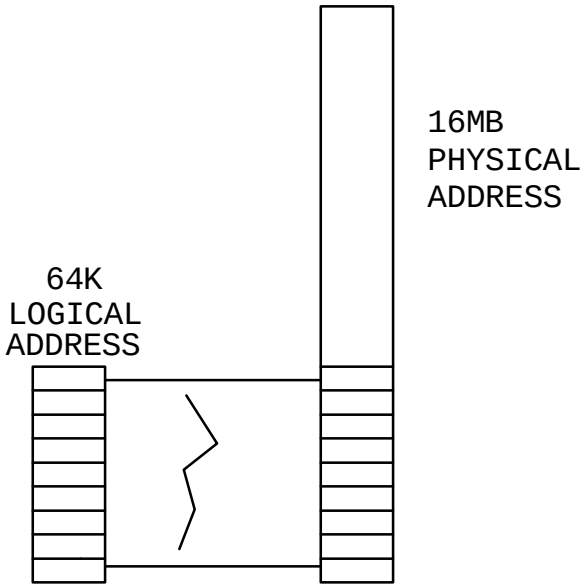


Fig 18. MMU mapping.

One of the first things we must deal with is initializing the MMU. If the MMU doesn't come up in a defined state at power up, and it soesn't, then we must somehow initialize it. This also means that our ROM accesses must not go through the MMU because we can't depend on it decoding the ROM addresses at power up. We'll look at how to get around this in a minute. For now, just assume that we can execute instructions from the ROM to initialize the MMU.

The first thing we must do to the MMU is to bring it to a known state. Figure 18 shows the MMU mapping we may wish to have at start up. This is simply a one to one mapping. The lower 64K of the physical address space is mapped onto the 64K logical address space. Now we can have a stack, make subroutine calls, service interrupts, etc. All the things a Z80 likes to do.

Afer we have initialized the MMU to its' default state we can start our application program running. For the sake of discussion let's say that we are designing a data logging system. Further, let's say that this system uses a BASIC interpreter in ROM, and that the application program is

also in ROM. The Z80's logical address space may look like this.

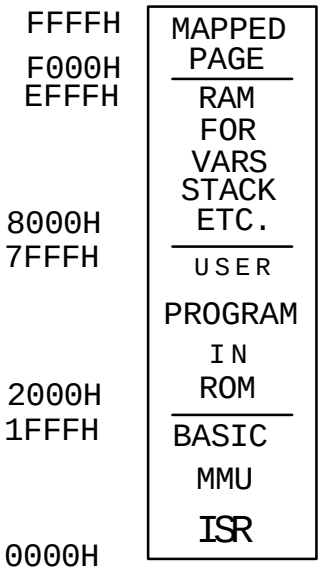


Fig 19. Possible magging for Z80 MMU

In figure 19 we see a possible layout for the Z80's 64K logical address space for our dtat logging application. We haven't said anything yet about how this is mapped to the physical memory. If we use the default mapping we set up in figure 18 we're almost there. We need to account for the ROM and reserve the last page for mapping. That's all there is to it. RIGGGHT!!! Wellllll. That's almost true.

OK. Let's deal with the ROM first. What I would do with it is get rid of it. We use it ust long enough to get the CPU up and then switch it out, never to use it again; until the next hardware reset, or power up. Once we have the MMU initialized, and the memory manager running, we realy don't want any memory active that is not going through the MMU. Remember that we said the ROM couldn't go through the MMU. This is one of the chicken/egg problems. We can't decode the ROM addresses from the MMU because it comes up in an unknown state. If we can't decode the ROM addresses from the MMU then we have no way to execute code so we can initialize the MMU so it can decode ROM addresses. Quite a mess huh? Well, there is a simple solution.

When the Z80 is reset we set a flip-flop that allows ALL memory reads, regardless of the address, to go to the ROM. The ROM has its address pins tied DIRECTLY to the Z80 CPU chip pins, not to the MMU. Now we can execute code at reset. After a quick thought you say "Hey now. If all reads go to the ROM, how do we access our stack?" The answer is "We don't." This is just a very temporary state we go through in bringing up the processor. The following code will establish the default state shown in figure 18.

```
; INITIALIZE THE MMU TO THE DEFAULT STATE
; THIS WILL ALLOW A ONE TO ONE MAPPING FROM
; PHYSICAL TO LOGICAL ADDRESSES. THE
```

```
; FIRST 64K OF DRAM IS MAPPED INTO THE Z80'S
; LOGICAL ADDRESS SPACE.
;
; THE FOLLOWING TABLE CONTAINS THE VALUES
; TO BE WRITTEN TO THE MMU ON STARTUP.
;
;
MMU.START:  DW   0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F ; DEFAULT MAPPING
;
MMU.LO:     EQU ##              ; I/O PORT ADDRESS FOR LOW TWO 189 CHIPS
MMU.HI:     EQU ##              ; HIGH 189 CHIP
;
KILL.ROM:   EQU ##              ; I/O DECODE THAT DISABLES ROM
;
;
;       NOTE : THIS CODE ASSUMES THAT THE TWO GROUPS OF 189 CHIPS
;       ARE DECODED AT SUCCESSIVE I/O PORTS.
;
;
SET.DEFAULT:
                LD      HL, MMU.START   ; POINT TO MMU TABLE
                LD      B, 0            ; ADDRESS FIRST ENTRY IN MMU
MMU.LOOP:       LD      C, MMU.LO       ; GET ADDRESS OF LOW 189 GROUP
                LD      A,(HL)                  ; GET TABLE ENTRY
                CPL     A               ; INVERT DATA
                OUT     (C), A          ; WIRTE TO LOW 189 GROUP
                INC     HL              ; POINT TO NEXT BYTE IN TABLE
                LD      A,(HL)                  ; GET IT
                CPL     A               ; INVERT IT
                INC     C               ; POINT TO HIGH GROUP 189
                OUT     (C), A          ; WRITE IT
                INC     HL              ; BUMP TABLE POINTER
                LD      A, B            ; GET MMU REG POINTER
                ADD     A,10H           ; BUMP IT IN THE HIGH 4 BITS
                LD      B, A            ; PUT IT BACK
                CP      A, 0            ; WAS THIS THE LAST ONE ?
                JR      NZ, MMU.LOOP    ; KEEP GOING IF NOT
;
;       WE NOW HAVE RAM MAPPED. WE CAN COPY THE ROM INTO RAM
;       AND SWITCH OUT THE ROM.
;
                LD      HL, 0           ; SET UP SORCE ADDRESS
                LD      DE, 0           ; SET UP DEST ADDRESS
                LD      BC, 8000H       ; GET LENGTH = 32K
                LDIR                    ; COPY ALL OF ROM TO RAM
                OUT     (KILL.ROM), A   ; SWITCH ROM OUT
```

```
;
;       FROM HERE ON, WE ARE RUNNING IN RAM.
;
                LD      SP, 7FFFH               ; SET STACK
                .
                .
```

The above code segment will handle MMU initialization. It first sets up the default mapping of one to one. The first 64K of the physical address space is mapped onto the Z80's logical address space. Then the contents of the ROM are copied into the DRAM. (I never said that writes couldn't go to the dram). The LDIR instruction very nicely copies the first 32K, which is all of the ROM, into the dram, at the same logical address. We couldn't have done this until after the MMU was programmed with it's default settings from the table MMU.START.

Now, if we just had a couple of variables we could write a routine that would step the page in the last MMU slot. If this routine were called repeatably it would result in :walking" a window through the entire address space. The window will appear in the last 4K of the Z80's logical address space, 0F000H to 0FFFFH.

```
;       THIS ROUTINE WILL STEP THE LAST PAGE OF THE MMU. SINCE WE
;       CAN'T READ THE MMU WITH AN I/O INSTRUCTION, WE MUST KEEP
;       AN IMAGE OF WHAT WE PUT IN IT. THIS ROUTINE WILL ALSO MAKE
;       IT CLEAR WHY WE COMPLIMENT THE DATA BEFORE WRITING IT TO THE
;       189'S. IT IS A LOT EASIER TO DO BINARY ARITHMETIC ON POSTIIVE
;       NUMBERS. SINCE THE 189'S INVERT THE OUTPUTS, WE INVERT, OR
;       COMPLIMENT, THE NUMBER WE PUT IN, SO WE WILL GET OUT WHAT
;       WE WANT.
;
;       IF THE MMU WRAPS AROUND 16MB, THEN THIS ROUTINE WILL RETURN
;       WITH "NZ", OR "Z" IF NO ERROR
;
LAST.PAGE: DW   0FH             ; INITIAL SETTING FOR LAST PAGE IN MMU
;
INC.MMU:    LD      HL, (LAST.PAGE)     ; GET LAST PAGE VALUE
            INC     HL                  ; BUMP IT
            BIT     4, H                ; DID WE WRAP AROUND 16MB?
            JR      NZ, MMU.ERR         ; ERROR IF SO
            LD      (LAST.PAGE), HL     ; SAVE NEW MMU VALUE
            LD      B, 0F0H             ; POINT TO LAST MMU PAGE
            LD      C, MMU.LO           ; GET POINTER TO WRITE TO MMU
            LD      A, L                ; GET LSB BYTE OF NEW MMU ENTRY
            CPL     A                   ; INVERT DATA
            OUT     (C), A              ; WIRTE TO LOW 189 GROUP
            LD      A, H                ; GET LSB BYTE OF NEW MMU ENTRY
            CPL     A                   ; INVERT IT
            INC     C                   ; POINT TO HIGH GROUP 189
```

```
                OUT   (C), A            ; WRITE IT
                XOR   A, A              ; CLEAR Z FLAG
                RET                     ; SEND BACK GOOD COMPLETION
;
MMU.ERR:        LD    A,0FFH            ; SEND BACK ERROR
                AND   A, A              ; TO CALLER BY SETTING
                RET                     ; NZ
```

If we want to bump the MMU page the above code will do the job for us. When we overflow the 16MB barrier we will get back an NZ status, and no change will be made in the MMU. I will leave it as an exercise for the student to figure out what would happen to the system if this test were not included. What would happen? Oh heck, I can't keep a secret. It would start writing over memory at physical location 00000H. Since we put our BASIC interpreter, and interrupt vectors there, the system would crash. All you would see of it is a little mushroom cloud over the CPU chip.

When setting up the system memory map we must be sure of a couple of things. Certain things must always be available. Some of these things are : Interrupt Service Routines, or ISRs, Interrupt/trap vector tables, and the MMU management code itself. For example, the routine shown above would need to be in common memory. At the least, it would not be good to load this routine anywhere in the range of 0F000H to 0FFFFH. If you did, the results would be a system crash very alike to the one in the previous paragraph. The Z80 would be executiln along until it hit the first I/O instruction which changed the MMU page. After the write the MMU would be pointing to a different place in memory and the next instruction fetched would not be likely to be what we want.

There is a way to make this work; you must make sure that the memory page you are switching to also has a copy of the same routine, in the same place in memory. Then it would work. Why would I ever want to do this? Well, let's consider another example application for our MMU circuit; multi-tasking. Let's say that we want to set up a system to watch four serial lines. When data is presented from the SIO, we will store it in memory. To make it easy we would like to write only one copy of the program, and let it multi-task to manage the four serial lines.

We will need to write a small multi-tasking kernel. It will handle setting up the four tasks, and any task switching we may need. We will assume a timer interrupt driven pre-emptive multi-tasking environment. Since the serial lines are using interrupts we must have the ISRs in common memory, or at least duplicated once per task. I will not concentrate on the application, but will look only at implimenting the multi-tasking.

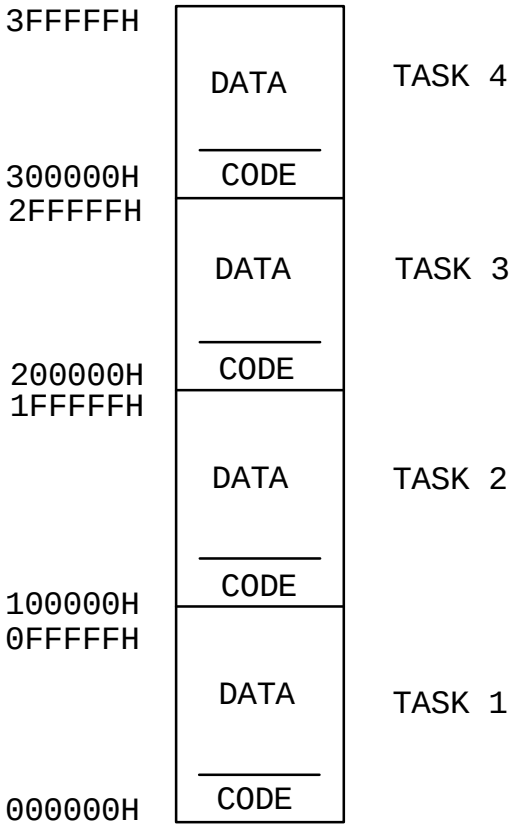


Fig 20. MULTI-TASKING memory model.

The memory model in figure 20 might suit our needs for the multi-tasking system. Notice that there is no space shown in the model for ISRs, kernel, etc. It is all lumped together and called "code". Also note that the code for each task is identical. When each task is started up it is given a task ID, or identifier. This may simple be a byte value in each tasks own memory. It will identify the task to the kernel.

Since the ISRs are actualy considered to be part of the kernel, incoming data from the serial lines would be placed in a buffer. The task may get the data a couple of ways. First, it could request a "wait for semaphore" from the kernel. In this case the task will be suspended until a byte is recieved. Whe this happens, the data is still placed in a buffer. The task is flaged as "ready to run" a,d started up the next time the kernel is looking for a task to run.

Another way to accumplish the same thing is to have each task periodicly poll the buffer to see if anything is in it. If so, the data is accessed and processed. If not, the process should make a kernel call to voluntarily surrender the CPU, assuming that it is stalled waiting for data.

The major difference between te two methods has to deal with the sophistication required in the real time kernel. For the first method the kernel must be able to handle semephores and connect them to events. It must also be able to suspend a process and restart it. These are common

features of commercial real time kernels. Once such kernel I have worked with is the USX80 kernel. It runs in a Z80 and provides all the features listed, and more.

In the second method, most of the "smarts" is moved to the application. A mechanism is required to switch tasks. This may be as simple as saving the machine state. I.E. : CPU registers and flags, to suspend a process. To restart the next process the kernel uses the task number to index into a table of MMU values and reprograms the MMU. If any MMU pages are allowed to be changed, then they will need to be restored from variables stored in each tasks memory, after the low MMU registers have been switched to point to the code space for the task. The tasks CPU registers are then loaded back into the CPU, and the process restarted. This is fairly simple code.

The call made by the process to give up the CPU only forced a task switch. When a task losses the CPU because of a timer interrupt it is called pre-emptuve multi-tasking. When a task voluntarily gives up the CPU it is called voluntary multi-tasking.

Both techniques may be combined in a system, and that is very appropriate for a Z80. In data logging applications it is not hard to over-run the CPU if the data comes in too fast. If you have one very high speed data channel, and the rest are of moderate data rate, the inclusion of the voluntary task switch call may speed the system up consisderably. The timer based task switch will guarantee that no task can hog the CPU, but it does not allow you to recover idle time from each task by itself. You need both methods together to do that.

If implimenting the simpler task switching system, my personal favorite, it would be a good idea to re-initialize the timer chip (within the kernel) when you execute the voluntary task switch. The timer interrupt will be asynchronous with respect to the task switch call so you don't know how much time remains before the timer will generate an interrupt. When you start the next task you would like it to have a full time slice to run before it is interrupted.

Ok, so how do we initialize the memory model in figure 20? I'll bet you thought I'd forgotten that, didn't you? If we're going to do this, let's do it right. Let's develop a multi-tasking kernel that we can use on our Z80.