A Report On The NEC V20 Microprocessor Chip For The "Z" Machines Part I

Richard L. Mueller, Ph.D.
11890-65th Avenue North
Maple Grove, MN 55369

Introduction

Some time last Fall, articles in magazines and notes on some of the Bulletin Boards started to talk about the replacement microprocessor chip for the 8088 chip from NEC. This chip is the V20, one in a series of advanced microprocessor chips in the NEC "V" series. For those micros that may be using the 8086 chip, the V30 is the replacement for that one.

In this article, I would like to relate my experiences in the short time that I have had the V20 chip and in a second article give a brief overview/background/description of the V20 itself with a mention of other "V" chips coming in the future. The information that I have seen on the V20 state that one can get anywhere from 5% to 100% or more improvement in execution of programs. Well, needless to say, I decided to get one this past December and test it out on both my Z-160 and H-100.

The V20 microprocessor chip comes in either a 5 MHz (uPD70108-5) version or an 8 MHz (uPD70108-8) version. In one of the notes that I saw recently, it talked about a 10 MHz version. However, I did not see that in the preliminary information that I received from NEC. Since I have not upgraded my H-100 to run at the 8 MHz speed, which means both of my machines run at 4.77 MHz, I ordered the 5 MHz version.

Benchmark Background

While waiting for my V20 chip to arrive (I ordered it from a hardware/software mail-order house), I decided to write a test program (my "benchmark") in assembly language and use MS-DOS to time a variety of instruction sequences. First, I would run my benchmark program on both machines using the 8088 chip that came with my machines, then rerun the program after installing the V20 chip. My intention was not to get an absolute precise timing of each instruction (although one could certainly do that with a lot of work), but basically to compare the timings of the various instruction sequences or operations of the two chips. I wanted to get a feeling on where the V20 chip was faster and where it was about the same as the 8088 microprocessor chip.

Before discussing the actual instruction sequences or operations that were used, I first want to describe how the timing was done. Anytime one wants to 'time' a particular operation, a time stamp must be taken immediately before the operation to be tested, and again a time stamp must be taken immediately upon completion of the operation. If precise timing is a requirement of your benchmark, then using the Operating System for getting the "times" may not be accurate enough. Some "overhead" would be introduced into your results.

In my case, I was not concerned with precise timings. I just wanted to get a "ballpark" comparison of several operations using the two microprocessor chips. Using the MS-DOS "GETTIME" function was just fine for me. One must remember, this MS-DOS time function only returns the time to the nearest hundreth of a second, which was fine for my testing.

For curiosity's sake, I called the MS-DOS "GETTIME" function twice in succession, saved the results from the first reading and compared them with the second reading. To my surprise, the readings were the same which tells me that the Operating System takes less than one hundreth of a second to process the time request. This does not mean there is no overhead involved with getting the time, it just means that the overhead is less than one hundreth of a second. Anyway, reading the time via the MS-DOS Operating System was negligible for my benchmark.

Just as the MS-DOS time function took what appeared to be "no time at all", since the time returned from the function itself is only carried to the nearest hundreth of a second, I expected my instruction sequences or operations to also take "no time" to execute. That's exactly what happened. To get around this situation, an instruction sequence or operation must be repeated a very large number of times to make sure the elapsed time is much greater than one hundreth of a second. I just used 1,000 or 1,000,000 depending on the operation; more on the operations in the next section.

List Of Operations Tested

The following is a list of the various instruction sequences or operations that were used in this comparison study. The results of the testing are contained in the next section. Although each instruction sequence or operation is intended to test out a particular instruction as its primary purpose, it takes many other instructions to accomplish the task. For example, a loop is needed to control the number of times a particular operation is executed. This means setting up counters, decrementing them, branching depending on the status of the counters, etc. So each instruction sequence or operation is more than just testing one instruction. But that's okay since the purpose of this exercise was just to compare various operations, not individual instructions.

1. 1000-Byte-Clear Test (1000 times). This sequence/operation clears (sets to zero) 1000 "Bytes" of memory using the 'REP STOSB' instruction. This 1000-byte buffer clearing operation is repeated 1000 times. Look at the results in the next section.
2. 1000-Word-Clear Test (1000 times). This sequence/operation is the same as the preceding test except that it clears 1000 "Words" of memory using the 'REP STOSW' instruction. This operation is also executed 1000 times to get a meaningful result.
3. 1000-Byte-Load Test (1000 times). This sequence/operation uses the 'REP LODSB' instruction. This sequence is not very meaningful as far as a production program is concerned, since it loads the same byte into the AL register 1000 times. However, from a benchmarking point of view, the operation makes sense. The 1000-Byte-Load test is executed 1000 times.
4. 1000-Word-Load Test (1000 times). This is the same as the preceding test except that a "Word" is loaded into the AX register instead of a "Byte".
5. 1000-Byte-Move Test (1000 times). Just as the name implies; 1000 "Bytes" are moved from one section of memory to another section of memory using the 'REP MOVSB' instruction. This Move operation is repeated 1000 times.
6. 1000-Word-Move Test (1000 times). This test is the same as the Byte Test above except "Words" are moved using the 'REP MOVSW' instruction.
7. Byte-Add Test (1,000,000 adds). Although this test would not be meaningful in a production program, it does, however, provide a good test for the ADD instruction. This test simply executes the following ADD instruction 1,000,000 times along with the accompanying loop instructions: 'ADD AL,BL'.
8. Word-Add Test (1,000,000 adds). Same as the previous test except that "Words" are added together instead of "Bytes": 'ADD AX,BX'.
9. Byte-Subtract Test (1,000,000 subtractions). This test simply subtracts one register from another 1,000,000 times using the following: 'SUB AL,BL'. Similar to the ADD tests.
10. Word-Subtract Test (1,000,000 subtractions). Same as above except "Words" are subtracted instead of "Bytes": 'SUB AX,BX'.
11. Byte-Divide Test (1,000,000 divides). Basically the same as the other arithmetic tests above: one word register is divided by a byte register. This operation is repeated 1,000,000 times using the following sequence: 'MOV AX,n' 'DIV BL' where n is any unsigned integer, and likewise, BL contains an unsigned integer. 'DIV' performs "unsigned" divisions.
12. Word-Divide Test (1,000,000 divides). Same as the preceding test except that this operation involves dividing a word register pair by a word register: 'MOV AX,n' 'MOV DX,m' 'DIV BX', where m is the most significant portion of a 32-bit unsigned integer and n is the least significant portion of the 32-bit unsigned integer. BX contains a 16-bit unsigned integer. Again, 'DIV' performs "unsigned" divisions.
13. Byte-Integer-Divide Test (1,000,000 divides). This is the same as the 'DIV' tests except that signed integers are used by the 'IDIV' instruction (i.e., signed divisions).
14. Word-Integer-Divide Test (1,000,000 divides). Same as the preceding tests except that a 32-bit signed integer is divided by a 16-bit signed integer. The DX AX register pair is divided by the BX register.
15. Byte-Multiply Test (1,000,000 multiplications). In this operation, the 8-bit unsigned integer in the AL register is multiplied by the unsigned integer in the BL register 1,000,000 times. This operation uses the 'MUL' instruction along with the accompanying loop instructions.
16. Word-Multiply Test (1,000,000 multiplications). Same as the Byte-Multiply test above except that the 16-bit unsigned integer in the AX register is multiplied by the 16-bit unsigned integer in the BX register.
17. Byte-Integer-Multiply Test (1,000,000 multiplications). This is the same as the Byte-Multiply test above except that the 8-bit signed integer in the AL register is multiplied by the signed integer in the BL register (i.e., signed multiplication operations). The 'IMUL' instruction was used here.
18. Word-Integer-Multiply Test (1,000,000 multiplications). Same as the preceding Byte-Integer test except that the 16-bit signed integer in the AX register is multiplied by the signed integer in the BX register.
19. Write 640 512-Byte Blocks Test. This operation writes 640 blocks of 512-bytes long on a floppy disk that was formatted as 9-sector per track disk. In other words, 320K of information was written on a 360K capacity disk. The purpose of this test was to see if the NEC V20 chip had any affect on floppy disk operations. See the results in the next section.
20. Read 640 512-Byte Blocks Test. This operation is similar to the preceding test except the blocks that were written above are now read. Nothing is done with the data that is read; each block read overwrites the previous block read into memory. Again, the purpose of this operation is just to see if the V20 chip has any affect on floppy disk reads.

Benchmark Results

Test	Z-160 (secs) 8088	Z-160 (secs) V20	H-100 (secs) 8088	H-100 (secs) V20
1. Byte-Clear	2.14	.88	2.16	.90
2. Word-Clear	3.07	1.87	3.06	1.79
3. Byte-Load	2.80	1.93	2.87	1.95
4. Word-Load	3.79	2.91	3.70	2.86
5. Byte-Move	3.79	1.81	3.69	1.79
6. Word-Move	5.49	3.62	5.48	3.55
7. Byte-Add	6.76	6.26	6.56	6.11
8. Word-Add	6.70	6.26	6.56	6.11
9. Byte-Subtract	6.75	6.26	6.56	6.11
10. Word-Subtract	6.70	6.26	6.56	6.10
11. Byte-Divide (unsigned)	25.71	11.37	25.74	11.26
12. Word-Divide (unsigned)	42.02	15.10	42.31	15.19
13. Byte-Divide (signed)	30.26	15.16	30.50	15.09
14. Word-Divide (signed)	45.43	19.22	46.68	19.20
15. Byte-Multiply (unsigned)	22.69	10.82	22.94	10.79
16. Word-Multiply (unsigned)	34.05	13.24	34.24	13.19
17. Byte-Multiply (signed)	25.21	12.97	25.13	12.93
18. Word-Multiply (signed)	36.03	15.10	36.46	15.35
19. Floppy Disk Write	142.48	142.47	135.06	135.06
20. Floppy Disk Read	142.70	142.69	135.23	135.24

Conclusions

Except for the floppy disk operation tests, all the other tests showed an improvement, some very significantly, when the NEC V20 microprocessor chip was used. The reason why the floppy disk operations remained the same is that the operations were very dependent on the physical disk drives themselves, rather than on any specific instruction or set of instructions. The differences between the Z-160 runs and the H-100 runs are due to different physical disk drives, and different BIOS/Disk Drivers being used.

As you can see from the results, the Divide and Multiply operations showed tremendous improvement with the NEC V20 chip, while other operations, such as the Add and Subtract operations, showed only very little improvement. The Load, Store, and Move operations also showed very significant improvement with the NEC V20 chip.

What all this means is that depending on the application that is being used, you could see little or no improvement over the 8088 chip or see a significant improvement. Those applications which depend heavily on disk activity will most likely show no improvement. However, those applications which have very little disk activity and have very high usage of the Multiply and Divide instructions, along with Loads, Stores, and Moves, will show a significant improvement. If your spreadsheet application has many formulas that perform multiplications and divisions, you will notice an improvement.

The application being used with its unique activity will determine whether there is any improvement or not with the NEC V20 chip. In most cases, you won't notice an improvement but it will be there. As I said earlier, I used the 5 MHz version of the V20. If you have upgraded your Z-Machine to run at 8 MHz, you have undoubtedly seen an improvement just running your 8088 chip at that speed. Adding the 8 MHz V20 chip should give you yet another improvement.

The purpose of this article was not to promote or sell you the NEC V20 microprocessor chip, but to give you some idea of some of the differences between it and the 8088 chip in terms of execution speed. The choice is yours. However, I can tell you that I am sold and I went out and purchased a second V20 chip so I could have a V20 chip in each of my Z-Machines. In my second article on the NEC V20 chip, I will discuss the additional instructions, the enhanced instructions, and the 8080 emulation mode that are all part of the NEC V20 (or V30) chip. I did write a test program that does switch between 8088 native mode and the 8080 emulation mode to perform a task, and I will discuss this in my next article.

Serious Business Applications?

CCT,® the Nation's leading S-100 OEM, now offers products and support for the Z100 system. We have compatible 256K and 512K static memory boards, and the CCT Printerfacer 1™ The Printerfacer 1 is an intelligent I/O board with 256K of on-board memory, expandable to 1 megabyte, for printer/plotter intensive application

Increases system speed and throughput tremendously! Runs two devices simultaneously! Fast!

bufferring. The board comes standard with 1 serial and 1 parallel port, residing at the same physical address, which are software and keyboard selectable. Increases system speed and throughput tremendously! Runs two devices simultaneously! Fast! Printerfacer 1 with cable set and CCT 12 month warranty.

Liberal Dealer Package Available—Inquire with CCT Marketing.

Zenith Data Systems

Z-138 PC Transportable .... CALL Z-148 PC Desktop .... CALL Z-158 PC Desktop .... CALL
Z-171 Lap Top Portable...... CALL Z-200 PC-AT System...... CALL

Hard disk upgrade kits are available for the Zenith System

CCT Implements Tomorrow's Technology Today™

CCT and Printerfacer 1 are trademarks of CCT.

CCT / 1 CCT Plaza • Box 4160 / West Sedona, AZ 86340 / 800-222-8686 / In AZ: 602-282-6299

A Report On The NEC V20 Microprocessor Chip For The "Z" Machines Part II

Richard L. Mueller, Ph.D.
11890–65th Avenue North
Maple Grove, MN 55369

Overview

In my last article on the NEC V20 microprocessor chip, I covered a comparison of some operations using the 8088 chip and V20 chip. In this article, I will cover the enhanced instructions, the additional instructions, and the 8080 emulation mode of the V20 and V30 chips.

Some time ago, NEC came out with a new family of high–performance, low power CMOS, microprocessor chips, the "V" series. The ones that are interesting today are the V20 and V30 chips which are the replacements for the 8088 and 8086 chips, respectively. The instruction sets for the V20 and V30 are supersets of the 8088 and 8086, and are compatible with those earlier micros in terms of pins, functions, and object code. I'm not a "hardware" person, so I won't go into details of the chip hardware itself. I will discuss the software aspect of the chips; that is, the instructions and modes.

Modes

First, let me discuss the various modes of the V20/V30 chips. There are three modes: Native, 8080 Emulation, and Standby modes. All three modes are mutually exclusive; that is, the microprocessor can be in only one of three states at a time. In the Native mode (8088/8086), all the instructions of the 8088/8086, as well as the enhanced and additional instructions of the V20/V30, can be executed. In 8080 Emulation mode, the microprocessor executes 8080 code (programs). What this means is that CP/M–80 programs based on the 8080 instruction set can be executed. More on this later.

Standby Mode

The third mode (state) of the V20/V30 is the Standby mode. This places the V20/V30 in an idle state where the microprocessor consumes only 10% of its normal operating current while retaining all data necessary to keep the microprocessor operative. The only way to enter Standby mode is to execute an HLT instruction in either the Native mode or 8080 Emulation mode. The processor exits Standby mode in response to a RESET, NMI (non–maskable interrupt), or an external INT (interrupt).

8080 Emulation Mode

Before discussing the Native mode new instructions, let me spend some time discussing the 8080 emulation mode. The V20/V30 chips have a special Mode Flag (MD) to select between the two operating modes. When initialized, the V20/V30 microprocessors are in Native mode and the Mode Flag is set to 1. In 8080 Emulation mode, the Mode Flag is set to 0. The Mode Flag is set and reset, directly and indirectly, by executing the mode manipulation instructions, three of which are new instructions.

Two of the new instructions are provided to allow one to switch the operating state from Native mode to 8080 Emulation mode and back again. The instructions are BRKEM (Break for Emulation) and RETEM (Return from Emulation). BRKEM is the basic instruction used to start 8080 Emulation mode. It operates basically the same as an INT instruction, except that the BRKEM sets the Mode Flag to 0. The Flags, Code Segment, and Instruction Pointer are all saved on the stack just as for an INT. The Interrupt Vector specified by the operand of the BRKEM instruction is loaded into the Code Segment and Instruction Pointer. The code pointed to by the Interrupt Vector is 8080 code and the CPU starts executing this code.

In 8080 Emulation mode, the registers and flags used are as follows:

	8080	8088/8086
Registers:	A	AL
	B	CH
	C	CL
	D	DH
	E	DL
	H	BH
	L	BL
	SP	BP
	IP	IP
Flags:	C	CY
	Z	Z
	S	S
	P	P
	AC	AC

In the Native mode, SP is used for the stack pointer, while the BP register is used for this function in 8080 Emulation mode. The SP, SI, DI, and AH registers along with the segment registers CS, SS, and ES are not affected by 8080 emulation mode operations. The Data Segment (DS) register is used in the Emulation mode for data and must be set to the Code Segment value by the user before entry is made to the 8080 Emulation mode.

When finished executing 8080 code, the execution of the RETEM instruction returns the operating mode to Native mode and sets the Mode Flag to 1. The microprocessor can now continue with the execution of 8088/8086 instructions.

NEC literature states that the V20/V30 microprocessors will run CP/M-80 (the version written in 8080 code and not in Z-80 code) operating system and programs. This is true, but it doesn't happen by magic. Since you are in the Native mode when the system is initialized and you are running MS-DOS (PC-DOS), you need a way to load CP/M-80 and 8080 programs into memory before you can execute the code. A small load program may be all that is necessary. However, there still is the question of disk files.

I haven't really looked into this very closely yet, but it may be necessary to convert CP/M-80 files to MS-DOS formats and utilize the second part of the 8080 Emulation mode which I will talk about shortly. Two companies have looked into this very closely and have products available for sale that will allow you to execute your CP/M-80 programs. Both of these companies have ADs in the February 1986 issue of REMark.

What I like about this 8080 Emulation mode is that while you are executing 8080 code, you can call an 8088/8086 routine by switching back to the native mode temporarily, execute the code there, and then return to 8080 Emulation mode. This is accomplished with a new instruction which is similar to the BRKEM and an existing instruction. The new instruction is the CALLN instruction which makes it possible to call Native mode subroutines. When the CALLN is executed, the Mode Flag is set to 1 and the Interrupt Vector pointed to by the operand of this instruction is loaded into the Code segment and Instruction Pointer registers. The microprocessor can now execute 8088/8086 code. Return back to 8080 Emulation mode by executing an existing instruction, the IRET (Interrupt Return) instruction.

To get a feeling how this switching between Native and 8080 Emulation modes works, I wrote a small test program in 8086/8088 assembly langauge that switched to 8080 Emulation to start processing a request, the 8080 code called a Native subroutine to complete the request, return to 8080 Emulation mode, and finally back to my main program in Native mode. The object of the program was to convert a series of binary numbers to ASCII decimal and display the numbers on the CRT.

The program sequence went like this:

- Set up Interrupt Vectors at 200 and at 201 to point to the start of 8080 Emulation code and Native mode subroutines, respectively.
- Place an 8-bit binary number in the AL register.
- BRKEM 200 — This causes the switch to 8080 Emulation code.
- The 8080 code starts the conversion process by determining what the hundreds digit is, converts that to ASCII, and saves it in the C register (CL). The remaining portion of the original number is left in the A register (AL).
- CALLN 201 — This causes a switch to Native mode to take place and the conversion processes continue. The tens digit is converted to ASCII and stored in the AL register (A) and the units digit converted to ASCII is stored in the CH register (B).
- IRET is executed by my Native mode subroutine to return to 8080 Emulation mode with the original number now fully converted to ASCII decimals stored in the registers indicated above.
- RETEM is executed by the 8080 Emulation mode to switch the state back to Native mode.
- The result is now displayed on the console CRT.
- The above sequence is repeated a number of times, each time using a different 8-bit integer to convert to ASCII decimals.

Now you are going to ask how I entered 8080 code into my program. Well, the way I did it, because the number of 8080 instructions were but a few, was by using 'DB' statements containing the HEX code of the 8080 instructions. I first wrote a small subroutine in 8080 code, used the 8080 assembler on CP/M-85 to assemble the code, got the HEX code from the listing, and put that into my 8086/8088 assembly program using the 'DB' statements. It's a way, but not necessarily the best way. However, it works. Another way would be to write a small disk read subroutine into your Native program that would read a file containing 8080 code and store it into your code segment where you want it. I'm sure there are other ways, as well. This little test program was tried on both of my Z-Machines and worked fine in both cases.

Enhanced Instructions

In addition to the "standard" set of 8086/8088 instructions, the V20/V30 have the following "enhanced" instructions:

- PUSH im — Pushes immediate data onto the stack.
- PUSH R — Pushes the contents of the four 16-bit general registers onto the stack (AX, BX, CX, DX).
- POP R — Pops the four 16-bit general purpose registers from the stack.
- MUL reg16,im16 — Multiplies the contents of a 16-bit register by 16-bit immediate data.
- MUL mem16,im16 — Multiplies the 16-bit contents of a memory location by 16-bit immediate data.
- SHL reg,im — Shifts specified register left by immediate value.
- SHR reg,im — Shifts specified register right by immediate value.
- SAR reg,im — Shifts specified register 'arithmetic right' by immediate value.

- ROL reg,im — Rotate specified register left by immediate value.
- ROR reg,im — Rotate specified register right by immediate value.
- RCL reg,im — Rotate specified register left through carry by immediate value.
- RCR reg,im — Rotate specified register right through carry by immediate value.
- CHKIND reg16,mem32 — Checks array index against designated boundaries. This is added for support for high-level languages. The index value is in reg16 and the lower limit is in location mem32 and the upper limit is in mem32+2.
- INM — Used to input a string into memory when preceded by a repeat prefix (REP). The address for storing the string is contained in the DI register and the I/O port is specified in DX.
- OUTM — Used to output a string from memory when preceded by a repeat prefix (REP). The address of the string is contained in the SI register and the I/O port is specified in DX.
- PREPARE im16,im8 — Generate stack frames required by high-level languages such as PASCAL and ADA that use block structures. This instruction provides the facilities of creating and linking the stack frames. The stack frame is pointed to by the BP register. The stack framer is composed of a local variable area (im8) and a copy area for frame pointers (im16).
- DISPOSE — This instruction releases the last stack frame generated by the PREPARE instruction. It returns the stack and base pointers to the values they had before the PREPARE instruction was used to call a procedure.

Additional Instructions (Unique)

In addition to all the "standard" 8086/8088 instructions and the enhanced instructions above, there are a number of unique instructions in the V20/V30 microprocessor chips. A number of these unique instructions have been added to support advanced business applications (packed BCD string operations); and to support engineering work stations, computer graphics, higher-language computing environments, and record-type data structures used in high-level languages (bit field manipulations).

- INS reg8,reg8 — This instruction transfers low bits from the AX register (the number of bits specified by the second operand) to the memory location specified by the ES Segment Register, plus the byte offset specified by the DI register with the bit offset specified by the low 4-bits of the first operand.
- INS reg8,imm4 — Same as the preceding instruction.
- EXT reg8,reg8 — This instruction loads to the AX register the bit field data whose bit length is specified by the second operand from the memory location specified by the Data Segment Register DS with the byte offset specified by the SI register and the bit offset by the lower 4-bits of the first operand.
- EXT reg8,im4 — Same as the preceding instruction.
- ADD4S — This instruction adds the packed BCD string addressed by the SI register to the packed BCD string addressed by the DI register, and stores the result in the string addressed by the DI Register. The length of the string, number of BCD digits) is specified by the CL register.
- SUB4S — This instruction subtracts the packed BCD string addressed by the SI register from the packed BCD string addressed by the DI register, and stores the result in the string addresses by the DI register. The length of the string in BCD digits is specified by the CL register.
- CMP4S — This instruction performs the same operation as SUB4S, except that the result is not stored and only the Overflow Flag, Carry Flag, and Zero Flag are affected.
- ROL4 — This instruction treats the byte data of a register or memory location specified by the instruction byte as BCD data and uses the lower 4-bits of the AL register to rotate that data one BCD digit to the left.
- ROR4 — this is the same as the preceding instruction except that it rotates the data one BCD digit to the right.
- TEST1 — Tests a specified bit in a register or memory location and resets the Zero Flag to 0 if the bit is a 1, and sets the Zero Flag to a 1 if the bit is a 0.
- NOT1 — This instruction inverts a specified bit in a register or memory location.
- CLR1 — Clears a specified bit in a register or memory location.
- SET1 — Sets a specified bit in a register or memory location.
- REPC — Repeats the next instruction until the Carry Flag becomes cleared or the CX register becomes zero.
- REPNC — Repeats the next instruction until the Carry Flag is set.
- FPO2 — Currently performs the same function as the ESC instruction used in conjunction with other processors, such as the 8087 coprocessor.

Other V-Series Chips

The only thing left to cover in this article is to say a few words about some of the other chips in the "V" series that are either available or will be in the future. At the time of writing this article, I did not have the current status of the other chips. However, I will talk about them briefly based on the literature that I received from NEC.

The V25 is a single-chip microcomputer aimed at portable microcomputer applications. The processing unit is the V20 chip with on-chip ROM and RAM for the application programs. It also has on-chip timers and DMA and Serial Interface Channels.

The V40 and V50 microprocessors integrate four independent DMA channels, three programmable timers, programmable interrupt controller, and on-chip clock generator along with the V20 and V30, respectively.

The only other chips in this series, that I am aware of, are the V60 and V70. The information that I have at this time is a little sketchy since I was not able to get much information on these from NEC. However, I do have some general information on them. These are 32-bit general-purpose microprocessors that realize mainframe functions on a single chip. The performance estimation at this time is from 1 million to 3 million instructions per second. It will support up to 4 gigabytes of virtual memory space with fast translation from virtual addresses to physical (or real) addresses. More information, I'm sure will be available later this year.

Conclusion

Hopefully, this article and the previous one will give you some idea of what the NEC V20 (and other "V" series chips) microprocessor chip is, how it performs with relation to the 8088 chip, what are some of the new instructions or enhanced instructions, and how the 8080 Emulation mode works.

Installing And Programming The NEC V20

Richard L. Ferch
1267 Marygrove Circle
Ottawa, ON K2C 2E1

In recent months, there has been considerable interest in the NEC V20 CPU chip. For example, a recent pair of articles in REMark discussed its speed improvements (April 1986, p.63), and its additional features (May 1986, p.45) relative to the 8088. However, neither explained how to use these new features in an actual program. This article is an attempt to provide such an explanation. It begins with some general background information on the 8088, V20 and related CPU chips, follows with detailed explanations of the new machine instructions the V20 offers, and ends with a discussion of my experience with the hardware aspects of installing a V20 (which wasn't quite as simple as I had expected, thanks to other modifications I had made previously).

Background

To begin with, it will be useful to describe briefly the differences between members of the Intel 8086 family. The 8086 itself is a true 16-bit CPU chip, which can transfer data to/from memory and I/O devices in 16-bit words. To reduce the pin count, the first 16 address lines (of 20) are multiplexed (shared) with data lines. The 8088, which is used in the H/Z-100, H/Z-100 PCs and "clone" PCs, is identical to the 8086 from the software point of view (except for timing). It is also very similar electrically, except that only 8 of the address pins are multiplexed with data. This allows the 8088 to interface readily with 8-bit memory and peripherals (and makes the H/Z-100's dual processor design feasible). As a result, though, the 8088 has to transfer 16-bit data words in two separate consecutive 8-bit bytes. Therefore, even at the same clock speed, an 8088 will be considerably slower than an 8086. Unfortunately, you cannot speed up an 8088-based system by substituting an 8086, without major redesign of the rest of the system.

The next members of the 8086 family are the 80186 and 80188. Electrically, these are completely incompatible with their predecessors, since many support functions were formerly done by auxiliary chips and are included on one chip. To the programmer, they are "upward-compatible", meaning that they execute all of the 8086/8088 instructions, plus several "enhanced" instructions. There are also a number of speed improvements: The calculation of effective addresses, which takes 5 to 12 clock cycles on the 8086/8088, takes only 2 cycles; iterative instructions (string moves and multi-bit shifts/rotates) are a lot faster; and multiply and divide speeds are much improved.

The H/Z-200 or "AT"-type computers use the 80286 CPU. This chip, which only comes in a 16-bit version, includes all of the software-related 80186 improvements, plus further hardware changes and additional instructions intended to support multi-user virtual-memory operating systems. However, MS-DOS does not make use of these "protected mode" instructions. Instead, it treats the 80286 as if it were an 80186. For example, recent versions of MASM have a ".286C" pseudo-operation to support 80286 instructions, and all of these new instructions are available on the 80186/80188 as well.

Now for the NEC V20: The V20, or uPD70108, is intended as a direct plug-in replacement for the 8088. (There is also a V30, or uPD70116, which replaces the 8086, and more advanced members of the "V" series may be able to replace other Intel CPUs.) Electrically, the V20 is the same as the 8088, except that it draws less power, and hence runs cooler. To the programmer, however, it looks like an 80188 with 8080 emulation added, and it also has several other new "unique" instructions. That is, when compared to the original 8088, the V20 does effective address calculations in only 2 clock cycles; string, shift/rotate, multiply and divide instructions run faster; the "enhanced" 80186/80286 instructions can be executed; there are several brand new instructions for packed BCD arithmetic and bit manipulation; and the V20 can be switched to 8080 mode to run 8-bit 8080 code.

Much of the initial interest in the V20 was based on its 8080 emulation capability. This is of particular interest to H/Z-100 PC

Table 1
"Enhanced" V20 Instructions (Intel Mnemonics)

BOUND	reg16,mem32	= 62 (mod reg16 r/m) [disp-low] [disp-high]
ENTER	aabb,cc	= C8 bb aa cc
IMUL	reg16,aabb	= 69 (11 reg16 reg16) bb aa
IMUL	reg16,cc	= 6B (11 reg16 reg16) cc
IMUL	reg16,mem16/reg16,aabb	= 69 (mod reg16 r/m) [disp-low] [disp-high] bb aa
IMUL	reg16,mem16/reg16,cc	= 6B (mod reg16 r/m) [disp-low] [disp-high] cc
INS	string,DX	= 6C/6D (for byte/word "string" type)
INSB		= 6C
INSW		= 6D
LEAVE		= C9
OUTS	DX,string	= 6E/6F (for byte/word "string" type)
OUTSB		= 6E
OUTSW		= 6F
PUSH	aabb	= 68 bb aa
PUSH	cc	= 6A cc
PUSHA		= 60
POPA		= 61
RCL	mem8/reg8,cc	= C0 (mod 010 r/m) [disp-low] [disp-high] cc
RCL	mem16/reg16,cc	= C1 (mod 010 r/m) [disp-low] [disp-high] cc
RCR	mem8/reg8,cc	= C0 (mod 011 r/m) [disp-low] [disp-high] cc
RCR	mem16/reg16,cc	= C1 (mod 011 r/m) [disp-low] [disp-high] cc
ROL	mem8/reg8,cc	= C0 (mod 000 r/m) [disp-low] [disp-high] cc
ROL	mem16/reg16,cc	= C1 (mod 000 r/m) [disp-low] [disp-high] cc
ROR	mem8/reg8,cc	= C0 (mod 001 r/m) [disp-low] [disp-high] cc
ROR	mem16/reg16,cc	= C1 (mod 001 r/m) [disp-low] [disp-high] cc
SAR	mem8/reg8,cc	= C0 (mod 111 r/m) [disp-low] [disp-high] cc
SAR	mem16/reg16,cc	= C1 (mod 111 r/m) [disp-low] [disp-high] cc
SHL	mem8/reg8,cc	= C0 (mod 100 r/m) [disp-low] [disp-high] cc
SHL	mem16/reg16,cc	= C1 (mod 100 r/m) [disp-low] [disp-high] cc
SHR	mem8/reg8,cc	= C0 (mod 101 r/m) [disp-low] [disp-high] cc
SHR	mem16/reg16,cc	= C1 (mod 101 r/m) [disp-low] [disp-high] cc

Note: The (mod reg r/m) byte and "disp" bytes are explained in Table 2.

(and "clone") owners, since they don't have the 8085 processor that H/Z-100 owners do. Commercial software is available for the H/Z-100 PC computers to run CP/M using the V20, giving them similar capabilities to the H/Z-100's CP/M-85. The V20 appears to be somewhat slower than the 8085 at the same clock speed. On the other hand, it has the advantage that improvements to the V20's clock speed will also speed up its 8080 emulation, whereas the 8085 on the H/Z-100 always runs at 5 MHz, regardless of the 8088's clock speed. The one simple benchmark 8080 program I tried took 103 seconds on the 8085 at 5 MHz, 123 seconds on the V20 at 5 MHz, and 82 seconds on the V20 at 7.5 MHz. The exact speed ratio probably depends on the instruction mix.

The speed improvements of some individual 8088 instructions on the V20 have been described elsewhere. In my experience, using typical higher-level language code, the V20 is about 5 to 10 percent faster overall than the 8088 at the same clock speed, although certain specialized applications may be improved slightly more. This speed-up is almost all due to the faster effective address calculations, since the other improvements affect only a small proportion of the typical instruction mix (even when doing number-crunching jobs, the CPU spends much of its time just moving data around and performing simple logical operations).

Without software changes, the other capabilities offered by the V20 (apart from this slight speed improvement) will go to waste. If your compiler (or assembler) has an 80186 or 80286 switch, you can take advantage of the "enhanced" instructions simply by telling the compiler you have an 80186 or 80286. Without such a switch, the only way to use these instructions is to hand-code them in Assembly language (or to write macros to do the encoding). The "unique" V20 instructions, including those related to 8080 emulation, always have to be hand-coded.

The following information is based on the NEC specification sheets, IBM documentation for MASM, and experiments with my V20. You will need experience with 8086 Assembly language to understand the explanations below.

"Enhanced" Instructions

The actual hexadecimal machine codes for these instructions are given in Table 1. The NEC mnemonics for many of these instructions differ from the Intel 80186/80286 mnemonics. I have used the Intel mnemonics, since they are more widely known, and are supported by recent versions of MASM. If your assembler has the ".286C" or equivalent pseudo-operation, you will be able to use all addressing modes with these instructions. Otherwise, you will not be able to use relocatable (i.e. assembler- or linker-resolved) addresses without resorting to undesirable techniques. Only addressing modes based on registers and fixed offsets (e.g., 4[BX + SI]) can be readily hand-encoded.

ENTER (NEC: PREPARE): This instruction is used to prepare the stack frame for subroutines in higher-level languages. The first argument, "aabb", is a word containing the number of bytes of local storage to reserve. The second argument, "cc", is a byte. In the most common case, when "cc" is zero, "ENTER aabb,0" is equivalent to:

```
PUSH BP
MOV  BP,SP
SUB  SP,aabb
```

The "aabb" bytes of local storage are addressed using negative offsets from BP, while the subroutine's arguments are addressed using positive offsets from BP. It is also possible to specify a non-zero value for "cc", in which case "cc" words of previous frame pointers are also saved:

```
PUSH BP
MOV  FP,SP
REPT cc-1
SUB  BP,2
PUSH BP
ENDM
MOV  BP,FP
PUSH BP
SUB  SP,aabb
```

(FP is an inaccessible hardware register.) Regardless of which form of ENTER is used, LEAVE should be used before every RET in the subroutine.

LEAVE (NEC: DISPOSE): This releases the frame set up by ENTER, and is equivalent to:

```
MOV  SP,BP
POP  BP
```

It is normally followed immediately by a RET.

PUSH immediate: An immediate word "aabb", or byte "cc" sign-extended to a word, is pushed onto the stack:

```
MOV  FP,aabb
PUSH FP
```

or:

```
MOV  FP,cc
PUSH FP
```

PUSHA (NEC: PUSH R): Registers AX, CX, DX, BX, the original SP, BP, SI and DI are pushed onto the stack:

```
MOV  FP,SP
PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH FP
PUSH BP
PUSH SI
PUSH DI
```

POPA (NEC: POP R): Registers DI, SI, BP, SP (discarded), BX, DX, CX and AX are popped from the stack:

```
POP  DI
POP  SI
POP  BP
POP  FP
POP  BX
POP  DX
POP  CX
POP  AX
```

IMUL immediate (NEC: MUL immediate): The first operand is always a 16-bit destination register, and the last operand is an immediate word "aabb" or byte "cc". If three operands are specified, the signed word addressed by the second operand ("mod", "r/m" and "disp" — for a summary of the "mod", "reg" and "r/m" bits, see Table 2) is multiplied by the immediate quantity "aabb", or "cc" sign-extended to 16 bits, and the lower 16 bits of the result are placed in the first operand register. If the result is longer than 16 bits, the carry and overflow flags are set; AF, PF, SF and ZF are undefined. If only two operands are specified, the source and destination are the same register.

Table 2
Explanation of Effective Address Calculations

r/m bits	mod = 00	mod = 01	mod = 10	mod = 11 (byte)	mod = 11 (word)
000	[BX+SI]	[BX+SI+disp8]	[BX+SI+disp16]	AL	AX
001	[BX+DI]	[BX+DI+disp8]	[BX+DI+disp16]	CL	CX
010	[BP+SI]	[BP+SI+disp8]	[BP+SI+disp16]	DL	DX
011	[BP+DI]	[BP+DI+disp8]	[BP+DI+disp16]	BL	BX
100	[SI]	[SI+disp8]	[SI+disp16]	AH	SP
101	[DI]	[DI+disp8]	[DI+disp16]	CH	BP
110	[disp16]	[BP+disp8]	[BP+disp16]	DH	SI
111	[BX]	[BX+disp8]	[BX+disp16]	BH	DI

Notes:
1. "reg" bits are the same as "r/m" bits (for the case mod = 11).
2. "disp8" is sign-extended to 16 bits for effective address calc.
3. "disp" byte(s) follow(s) (mod reg r/m) byte, lower byte first.
4. The byte/word choice depends on the preceding byte.

ROL immediate, ROR immediate, RCL immediate (NEC: ROLC immediate), RCR immediate (NEC: RORC immediate), SAL/SHL immediate (NEC: SHL immediate), SHR immediate, SAR immediate (NEC: SHRA immediate): All of these instructions are exactly like their counterparts which use CL for the shift count, except that the shift count "cc" is an immediate byte quantity. Note that, like the 8088 but unlike the 80286, the V20 allows shift/rotation counts greater than 31. (Of course, large shift counts don't do anything small shifts can't do — they just take longer.)

INS/INSB/INSW (NEC: INM): This instruction works the same way as STOS, except that the data comes from the I/O port addressed by DX, instead of from AL or AX. The destination string is addressed by ES:[DI], and DI is modified after every transfer, depending on the data type of the string and on DF. If this instruction is preceded by a repeat prefix, the input device must be fast enough to supply a new data value every 8 clock cycles; if INSW is used, the port must transfer 16 bits at a time. These conditions are unlikely to be met in most systems.

OUTS/OUTSB/OUTSW (NEC: OUTM): This instruction works like LODS, except the data is sent to the I/O port addressed by DX. The source string is addressed by DS:[SI]. If a repeat prefix is used, the output device must be fast enough to accept a new data item every 8 cycles; if OUTSW is used, the port must transfer 16 bits at a time. These conditions are unlikely to be met in most systems.

BOUND (NEC: CHKIND): The second operand must be a doubleword memory location (mod=11 is not allowed). If the signed value of the first operand is either less than the first word or greater than the second word, an INT 5 interrupt occurs. If your system uses INT 5 for the Print Screen interrupt (as Z-DOS/MS-DOS/PC-DOS do), you will be unable to use this instruction (unless your desired response to an out-of-range value happens to be a print screen operation!).

"Unique" Instructions

The following NEC-only instructions are not supported by widely-available software. Therefore, they will have to be hand-coded, and relocatable addresses cannot be used. They can be divided into three groups: Packed BCD instructions, bit manipulation instructions, and processor control (including 8080 emulation). NEC mnemonics are used. The actual hexadecimal machine codes are given in Table 3.

ADD4S: This adds the packed BCD string at DS:[SI] to the packed BCD string at ES:[DI], and stores the result at ES:[DI]. The length of the strings (1-254) is specified by CL. Carry and zero flags are affected, but they will only be as expected if CL is even. If CL is odd, the upper 4 bits of the highest byte may also be affected by this instruction, and the flag values will depend on their contents. The AF, OF, PF and SF flags are undefined after this operation.

SUB4S: The same as ADD4S, except subtracts instead of adding.

CMP4S: The same as SUB4S, except that the result of the subtraction is not stored, so ES:[DI] is unchanged. Only the zero and carry flags are affected; if CL is odd, the flag values will not be as expected, but will depend on the contents of the upper 4 bits of the highest byte.

ROL4: The single byte addressed by "mod", "r/m" and "disp" is rotated left by 4 bits through the lower 4 bits of AL. The flag bits and the upper 4 bits of AL are unaffected.

ROR4: The same as ROL4, except the rotation is to the right.

INS: (Warning: note the conflict in mnemonics between this instruction and the "enhanced" Intel input string instruction.) The lower 4 bits of the second operand, which may be either an 8-bit register or an immediate byte, are used as a bit count. The specified bits are moved from the low end of AX to the memory location specified by ES:[DI] at the bit offset specified by the lower 4 bits of the first operand, which must be an 8-bit register. The first operand register, and DI if necessary, are updated to point to the next bit field in memory.

EXT: This instruction is the inverse of INS. The bit field at DS:[SI], bit offset in the lower 4 bits of the first operand, length in the lower 4 bits of the second operand, is transferred to the low end of AX. The first operand register, and possibly SI, is/are updated to point to the next bit field.

Table 3
"Unique" Instructions (NEC Mnemonics)

```
ADD4S                  = 0F 20
BRKEM  cc              = 0F FF cc
CLR1   mem8/reg8,CL    = 0F 12 (mod 000 r/m) [disp-low] [disp-high]
CLR1   mem16/reg16,CL  = 0F 13 (mod 000 r/m) [disp-low] [disp-high]
CLR1   mem8/reg8,cc    = 0F 1A (mod 000 r/m) [disp-low] [disp-high] cc
CLR1   mem16/reg16,cc  = 0F 1B (mod 000 r/m) [disp-low] [disp-high] cc
CMP4S                  = 0F 26
EXT    reg8a,reg8b     = 0F 33 (11 reg8b reg8a)
EXT    reg8a,bb        = 0F 3B (11 000 reg8a) bb
FP02   fpop            = 66 (11 yyy zzz)  OR
                         67 (11 yyy zzz)
FP02   fpop,mem       = 66 (mod yyy r/m) [disp-low] [disp-high]  OR
                         67 (mod yyy r/m) [disp-low] [disp-high]
INS    reg8a,reg8b     = 0F 31 (11 reg8b reg8a)
INS    reg8a,bb        = 0F 39 (11 000 reg8a) bb
NOT1   mem8/reg8,CL    = 0F 16 (mod 000 r/m) [disp-low] [disp-high]
NOT1   mem16/reg16,CL  = 0F 17 (mod 000 r/m) [disp-low] [disp-high]
NOT1   mem8/reg8,cc    = 0F 1E (mod 000 r/m) [disp-low] [disp-high] cc
NOT1   mem16/reg16,cc  = 0F 1F (mod 000 r/m) [disp-low] [disp-high] cc
REPC                   = 65
REPNC                  = 64
ROL4   mem8/reg8       = 0F 28 (mod 000 r/m) [disp-low] [disp-high]
ROR4   mem8/reg8       = 0F 2A (mod 000 r/m) [disp-low] [disp-high]
SET1   mem8/reg8,CL    = 0F 14 (mod 000 r/m) [disp-low] [disp-high]
SET1   mem16/reg16,CL  = 0F 15 (mod 000 r/m) [disp-low] [disp-high]
SET1   mem8/reg8,cc    = 0F 1C (mod 000 r/m) [disp-low] [disp-high] cc
SET1   mem16/reg16,cc  = 0F 1D (mod 000 r/m) [disp-low] [disp-high] cc
SUB4S                  = 0F 22
TEST1  mem8/reg8,CL    = 0F 10 (mod 000 r/m) [disp-low] [disp-high]
TEST1  mem16/reg16,CL  = 0F 11 (mod 000 r/m) [disp-low] [disp-high]
TEST1  mem8/reg8,cc    = 0F 18 (mod 000 r/m) [disp-low] [disp-high] cc
TEST1  mem16/reg16,cc  = 0F 19 (mod 000 r/m) [disp-low] [disp-high] cc

CALLN  cc              = ED ED cc  (WHILE IN 8080 MODE)
RETEM                  = ED FD  (WHILE IN 8080 MODE)
```

Note: The (mod reg r/m) byte and "disp" bytes are explained in Table 2.

TEST1: The second operand is either CL or an immediate byte. The first operand may be either a byte or a word, either in memory or a register. The bit specified by the bit offset in the lower 3 (byte) or 4 (word) bits of the second operand, at the address specified by the first operand, is tested. If the bit is zero, ZF is set to 1, and if the bit is 1, ZF is reset to 0. The carry and overflow flags are cleared, and AF, PF and SF are undefined.

CLR1: The bit specified by the bit offset in the second operand, at the address specified by the first operand, is cleared; the flags are unaffected.

SET1: Like CLR1, except the specified bit is set to one.

NOT1: Like CLR1 or SET1, except that the specified bit is inverted.

REPC/REPNC: These repeat prefixes are similar to REPZ/REPNZ, except that CF is used instead of ZF. That is, the following string operation is repeated until CF is cleared (REPC) or set (REPNC), or until CX becomes zero.

FPO2: This instruction is similar to ESC. Its existence leads me to speculate that the NEC replacement for the Intel 8087 may have additional new instructions (above and beyond the 8087 instruction set).

BRKEM: This instruction is used to enter 8080 emulation mode. It works the same way as "INT cc", except that the CPU is placed in 8080 mode (by clearing the Mode Flag, which is bit 15 of the Flag Word). Only the 8080 instruction set is supported (neither the Z-80 nor the 8085 enhancements are implemented). While in 8080 mode, instruction addresses are calculated relative to CS (set by the interrupt vector), and data addresses are relative to DS (normally set to the same value as CS by the calling program, immediately before executing the BRKEM). The following 8088 registers are used as 8080 registers: AL as A, CH as B, CL as C, DH as D, DL as E, BH as H, BL as L, BP as SP, and IP as PC. The SP, SI, DI, AH and segment registers are unaffected. While the CPU is in 8080 mode, external interrupts are handled in "native" 8088 mode as usual, but IRET causes a return to 8080 mode. The RETEM instruction is used to return to "native" mode.

RETEM (8080 MODE ONLY): This instruction plays the same role after a BRKEM that IRET does after an INT. It causes a return to "native" mode, at the instruction immediately following the BRKEM.

CALLN (8080 MODE ONLY): While in 8080 mode, this instruction can be used in exactly the same way that "INT cc" would be used in "native" mode. The interrupt routine it invokes will be in "native" 8088 mode, and must not include BRKEM. The IRET at the end of the interrupt routine will cause a return to 8080 mode at the instruction following the CALLN.

Hardware Notes

The V20 is supposed to be a plug-in replacement for the 8088, and many users have had success with a simple direct substitution. However, there are some possible pitfalls, especially when sped-up older systems are involved, as my experience shows.

I have an old H-100 (85-2653-1 motherboard), to which I have added the CDR ZS100 7.5 MHz speed-up and the FBE Research ZMF100 768k memory modification. This system was working fine at 7.5 MHz with the original ICs (except for the new 256k RAM chips, which are rated at 150 ns). When I substituted an 8 MHz-rated V20 (uPD70108D-8) for the 8088, however, I started experiencing intermittent system crashes. These went away at 5 MHz, or at either speed when the 8088 was re-installed.

My first reaction was to try upgrading some of the support chips, as suggested in various letters and articles in REMark. Far from fixing the problem, however, the new chips made it much worse. The more high-speed chips there were in the system, the shorter the interval between crashes. Evidently, there is a timing glitch on the old board when fast parts with rapid switching times are used.

My next step was to replace the original support chips and the 8088 CPU, while I studied the wiring changes suggested for installing the HA-108 upgrade kit on older H/Z-100s (REMark, July 1985, p.22). The memory modification and 256k RAM modification were not relevant, since the ZMF-100 did the same job and I knew it worked at 7.5 MHz. (Actually, the ZMF100 upsets the H/Z-100's memory map options, but none of my operating systems or other software uses this feature anyway.) The wait state modification was likewise not needed, and the IC replacement had made things worse, while I preferred the ZS100 speed-up to the crystal replacement (if something goes wrong, I can try it again at the lower speed to make sure it isn't clock-related). That left the refresh clock, ready logic and swap logic modifications to be considered. The ready logic change was an obvious candidate if my hypothesis about a timing glitch was right, but I decided to make all three changes while I was at it.

Being very reluctant to cut traces on the motherboard, I used the prepared IC socket technique for all wiring changes. That is, whenever the connections to an IC were to be modified, I removed the IC, installed a "prepared" IC socket with some pins bent up, wired the required jumpers to the bent-up pins, and then inserted the IC into the new socket. All jumpers were wired either to bent-up IC socket pins, circuit board feedthroughs, or resistor or capacitor leads using wire wrapping type wire (for a detailed description, see the Appendix). In order to accommodate the swap logic change, I also had to replace the smaller ZMF100 board using a similar technique. The resulting changes are completely reversible from the component side of the motherboard, but luckily they didn't need to be reversed — the system now works perfectly with either CPU chip at either speed, with all the original support chips.

The lesson here is that the relationship between clock speed changes and IC speed ratings can be surprising. I now suspect that I might have had the same problems with an 8 MHz 8088-2 that I had with the V20. On the other hand, if I'd bought a 5 MHz V20 (uPD70108D-5), there's a chance it might have worked at 7.5 MHz without problems, just the way my original 8088 did (although I understand that NEC may not be quite as conservative with their speed ratings as Intel is, so I wouldn't recommend you try this unless you can exchange your slow V20 for a faster one if need be).

If you have an old H/Z-100 motherboard and are planning to experiment with clock speed and CPU chip changes, you should consider making the wiring changes I made (which are also all described on p.26 of the July 1985 REMark). If you have a different speed-up modification than the ZS100, you should first check how it works. The ZS100 module simply replaces the oscillator, and doesn't affect other aspects of system timing (for more details, see REMark, April 1985, p.20). It is possible that other third-party speed-up modules are wired differently and might not be subject to the same problems when "fast" chips are used on the old motherboard.

For the price of the V20, a few IC sockets, the ZMF100 and ZS100 kits and 27 256k RAM chips, my old H-100 is now a 768k machine which runs about as fast as the 8 MHz H/Z-108, and it also has the "enhanced" and "unique" V20 instruction set improvements. If only I hadn't bought those unneeded fast support ICs!

Appendix
Recommended Wiring Changes For Old (85-2653-1) H/Z-100 Motherboards

Ready Logic Modification

U205 - 14-pin IC socket, pins 12 and 13 bent up, plugged into old socket
- connect pin 13 to new U206 socket pin 1 (bent up)
- connect pin 12 to new U206 socket pin 13 (bent up)
- plug U205 into new socket

U206 - 14-pin IC socket, pins 1 and 13 bent up, plugged into old socket
- connect pin 1 to U205 socket pin 13, U220 socket pin 5 (both bent up)
- connect pin 13 to U205 socket pin 12, U236 socket pin 4 (both bent up)
- plug U206 into new socket

U220 - 14-pin IC socket, pins 5 and 6 bent up, plugged into old socket
- connect pin 5 to U206 socket pin 1 (bent up)
- connect pin 6 to U236 socket pin
- 3 (bent up)
- plug U220 into new socket

U236 - 18-pin IC socket, pins 3, 4 and 15 bent up, plugged into old socket
- connect pin 4 to U206 socket pin 13 (bent up)
- connect pin 3 to U220 socket pin 6 (bent up)
- connect pin 15 to keyboard (GND) end of R123
- if you are using a ZS100 or similar speed-up module, plug it into this prepared socket; otherwise, plug U236 into new socket

Refresh Clock Modification

First, check that there are no jumpers connected to U130 pins 8, 9 and 10 on the back of the motherboard; if there are, this change has already been made.

U130 - 14-pin IC socket, pins 8, 9 and 10 bent up, plugged into old socket
- connect pin 10 to keyboard end of R104
- connect pin 8 to feedthrough beside (connected to) U152 pin 1
- connect pin 9 to feedthrough 1/8" out from inner end of U243, just left of centerline of U243 (connected to U225 pin 3)
- plug U130 into new socket

U168 - 14-pin IC socket, pin 11 bent up (N.C.), plugged into old socket
- plug U168 into new socket

Swap Logic Modification

If you have a ZMF100 kit installed, you will have to replace the small piggyback board at U173 to make room for the U156 socket:

U173 - 20-pin IC socket, pins 1 and 2 bent up, plugged into old socket
- connect pins 1 and 2 together, and to the keyboard (GND) end
- of C168
- connect the P1 pin nearest the keyboard on the ZMF100 large piggyback board to the feedthrough between U156 pin 7 and C169 (formerly connected to U173 pin 1)
- connect the other P1 pin to the feedthrough nearest U173 pin 4 (formerly connected to U173 pin 2)
- plug U173 into new socket and set ZMF100 small board aside

U155 - 14-pin IC socket, pin 8 bent up, plugged into old socket
- connect pin 8 to U156 pin 13 (bent up)
- plug U155 into new socket

U156 - 14-pin IC socket, pins 11, 12 and 13 bent up, plugged into old socket
- connect pin 13 to U155 pin 8 (bent up)
- connect pin 11 to U171 pin 10 (bent up)
- connect pin 12 to feedthrough next to (connected to) U171 pin 1
- plug U156 into new socket

U171 - 14-pin IC socket, pin 10 bent up, plugged into old socket
- connect pin 10 to U156 pin 11 (bent up)
- plug U171 into new socket