Ken Burgett
10 Nov 2017

1972: interest in 8008, interview

In the fall of 1972, a data sheet landed on my desk. That document described the Intel 8008 computer chip. I don't think the 'microprocessor' term was in use yet.

I was working as a Software Manager/Developer for a small electronics company, and we had been developing and selling systems based on a proprietary 8-bit MSI CPU design. The CPU was a single board of about 100 chips, and this 8008 was a single 16-pin DIP (dual inline package). I said to myself, "This is important! Who is this Intel Corp?" Another engineer panned the 8008 since it was so slow, and I agreed, but the form factor and cost aspects had me really intrigued. So, I made some calls.

I found out that a hardware engineer I knew well had left my company and had joined Intel, so I called him to check out this company. My friend Bob told me about Intel and about what the company was doing, that he was in the Microprocessor Group at Intel, and that they were currently at 19 people and looking for people with software skills. We arranged a visit in the next few days and I interviewed in the next week.

Hank, the manager of the team, was a pleasant guy, but made it clear that he wasn't up-to-speed about software people, so he said he had arranged a phone call with one of his consultants, a guy named Gary Kildall, who was an instructor at the Naval Postgraduate School in Monterey, and that he would do the interview.

The call went through and we made introductions. Among other things, Gary asked what I thought about possible programming languages for the 8008. I told him that I had taken several summer short courses from Bill McKeeman at UC Santa Cruz, and had learned about the XPL compiler writing system. I said I liked the block structure of the PL/I-like language, but didn't think the complex string processing data types used for compiler writing were appropriate for a small 8-bit processor, so I suggested a PL/I-like language with 2 simple data types: byte and word, with arrays of each, with a character string being an array of bytes. Gary agreed, I was hired, and started at Intel on the first working day of 1973.

1973: The Intel 4004, short and bittersweet.

Among my responsibilities, being the first full-time software person in the Intel Microprocessor Group, was care and feeding of all of the CPU products, which included the Intel 4004 processor, a 4-bit computer. I will say right now that any piece of hardware that has to treat an ASCII character as a double-precision integer is not a computer to me. I would have to manage and fix bugs in 4004 ROM code and an assembler, written by a consultant.

I had to modify the assembler program, written to fit into 2 1K EPROMS, so that it could recognize 'mark' parity, where the high order bit of an ASCII character is always '1', instead of '0' parity, which is the only input the assembler would presently accept. While working on this issue, it became clear to me that the consultant author of this software had used machine instructions for storage of constants. The consultant showed me where he had used a few bytes of executable code as storage, so that changing the parity changed his logic. He found a fix and I applied it.

The only good thing I can say about the 4004 is that it is an accurate implementation of a "Harvard Architecture", where storage of code and storage of data are distinct. The 4004 had a set of 16-digit registers, intended for BCD computation suitable for a calculator, and a separate code storage, normally a ROM. The Intellec4 computer had quite a bit of logic built into it to simulate 4004 ROMS in RAM, something needed for a

development system. It was an interesting design, but not a computer that would support any kind of operating system. The big contribution of the 4004 was in the field of integrated circuits, not computer science. End of 4004 work.

8008, 8008 ROM BIOS/monitor, PL/M Gary Kildall developed and wrote the Fortran cross-compiler for ‘PL/M’. We picked Fortran as the implementation language because Intel had a PDP-10 timesharing system for engineering use, and Fortran was the best compiler available. I tested the PL/M compiler and gave feedback. The compiler produced a loadable ASCII file of hexadecimal machine instructions, which was loadable via the Intellec8 monitor that I was building. I also started work on developing a text editor for the 8008, to be written in PL/M. This was the first program written in this new language.

The 8008 was what brought me to Intel, and I set about learning the architecture and assembly language. Not much to learn. 48 instructions, about 4 registers, 16k address space. No interrupt system to speak of. The world needed a bit more from a single chip computer, I thought.

Many of the problems associated with the 8008 were that it was shoe-horned into a very limited LSI technology, called ‘P-channel’, and this made it slow and limited the space necessary for a decent instruction set.

The 8008 CPU had no DEBUG instruction. Many computers have a single byte instruction, called DEBUG, which when executed, would do an indirect call through a fixed address. This allows people who write debuggers to swap out the one-byte opcode out of any 1, 2, or 3-byte instruction, replace it with a DEBUG instruction, thereby setting a breakpoint. You had to save the opcode somewhere and you had to be able to support at least 2 breakpoints, for both sides of a branch. The 8008 had none of this, and so it was impossible to write a debugger for it. So, the ROM monitor that I needed to build for the upcoming Intellec8 computer could not also be a debugger, which really limited what it could do. It could do things like loading a hex file produced by the cross-assembler, displaying and setting the registers, dumping a block of RAM to the teletype printer, along with typing in code in hex and then jumping to the entry point and letting it run, hoping to get some response. Programming an 8008 at the assembly level was not easy.

It was easy to agree on the general direction for the PL/M programming language and the compiler should take. We wanted to reduce the runtime support code for the compiler to a bare minimum. We eventually decided to add a 16-bit multiply and a 16-bit divide as library functions that got emitted by the compiler on first use and then point back to the function for later calls. The PL/M language had no explicit I/O operations or libraries. This was a low-level language for a low-level computer, and the I/O would have to be added by the application developers as needed. A cross compiler can rely on I/O derived from the host OS, but couldn’t guess what would be needed in applications. We kicked the can down the road.

The PL/M (and later PL/M-80) cross compilers also had no support for any kind of linkage editor or loader, nor was the code relocatable. Everything in the source file was compiled directly into an output hex file that was a complete memory map, designed to be dropped into a fixed location in the 8008’s small 16kb RAM space, which often was just a 4k RAM board and 3 empty slots.

Within the constraints of the CPU, the PL/M compiler worked quite well. I wrote a text editor, using syntax taken from the DEC PDP-10 ‘teco’ editor. It loaded from paper tape, so the user could read in the paper tape of a source module under construction, edit or add to the source in RAM, and then dump the code out to the paper tape punch. Primitive, at best.

We used the PL/M compiler extensively, as hosted on our PDP-10 time-sharing system. We also sold the compiler in source form on

magnetic tape for installation on customers' in-house development environments. The 8080, something to build on.

I learned that the 8080 was in development, and that implementation technology was going to be Intel's new "N-channel' fabrication process, which was much denser, and a lot faster, than the 8008's old 'P-channel' fab process. I also learned that there was a good possibility that we could update the 8008 architecture considerably, which is what I set out to do. We (the software folks) didn't get all we wanted, They (the process folks) had the last word whether a feature could be added without threatening the development timeline. Nevertheless, the 8080 had a greatly improved instruction set, more working registers, a real stack pointer, a much bigger address space, a real interrupt system, and a DEBUG instruction. And it was a lot faster. The 8080 was also packaged in a 40-pin Dual Inline Package. 40 pins meant that the address could present 16 bits at a time, which meant that board developers did not have to add outboard registers and multiplexers that the 8008 needed.

Hopes were sky high for the 8080. We heard that the chip designers had completed a chip layout, and we could stop by and see it. What I saw was a complete room, empty except for Mylar sheets taped to the floor, with a maze of black Mylar tape lines, running all across the room. I thought at the time it looked like the street map of a medieval European city. I called the layout "Downtown Warsaw". I could see blocks of logic grouped together, but a lot of it looked very hand-built. Later chip designs were completely built on CAD systems, but this was not. I think pieces were built in 1973-vintage CAD systems, but integration was done on the floor, with tape.

When release time came in April 1974, a Component Sales guy came down the hall carrying a large plastic tub, filled with little cardboard matchboxes. Each box contained an 8080A CPU, stuck on a piece of conductive foam. Each box was going to a customer who had pre-ordered an evaluation chip at $540 each. The salesman commented that he had nearly 'a quarter of a million dollars' in the tub. That translates to about 500 chips, so he might have been exaggerating a bit, but the excitement was palpable.

Intellec8 mod 80. The microprocessor group wanted to build a complete new development box for the 8080, but management insisted that we modify the Intellec8 to support the new 8080, so that's what we did. A year later, we got the chance to build the MDS development system, with a very cool Multibus architecture, which allowed multiple masters to share the bus. That was a feature left out of the IBM PC, which came along almost a decade later.

The Intellec8 mod 80 did give me a place to develop a much improved "BIOS". That term did not yet exist, but the ROM monitor I was building would be the loader, debugger, and stream I/O subsystem that my later disk operating systems would rely upon for character I/O, so it was a true "BIOS".

The 8080 had some badly needed architectural features over the 8008, most notably, the Stack Register. This was a 16-bit register which pointed to a section of RAM memory. A CALL instruction would push the return address onto the stack, decrement the register by two, and then jump to a function's entry point. A RET instruction would pop the return address off the stack and jump to it. The 8080 stack could also push parameter addresses or data before the call and a called function could access those parameters using stack-relative addressing, which is a really useful bit of hardware for a programmer, and something that a compiler could really leverage in a programming language implementation.

By comparison, the 8008 stack implementation was an 8 address internal array. When you went beyond 8 levels deep in a call sequence, you never came back.

I worked on variations of the 8080 ROM monitor for over two years, squeezing more and more features into the same 2k bytes of ROM, situated

at the top of the 16-bit address space. I didn’t work on it full time, but kept coming back to it and folding in a good suggestion when one came by.

One of those was to use stack-based addressing for all of the internal state used in the ROM. That way, the monitor would determine how much RAM was installed, up to 64k, allocate the necessary work-space at the top, and set up the stack pointer below it. The ROM monitor did not have to interfere with the rest of the RAM, leaving that to the application programmer’s choices.

Another feature we added was an auto-baud-rate detector for the input terminal. We could program the UART to a maximum baud-rate, like 9600 bps, and then type spaces on the keyboard. The algorithm could detect the spaces between bits in the 0x0a bit pattern and determine the actual input baud rate, and then generate an input prompt. This was a small feature, but very useful as developers moved away from Model 33 teletypes to 2400 and 9600 baud CRT terminals.

The terminal used on Intellec8 mod 80 systems was often a model 33 teletype, usually with a paper tape reader and punch, along with keyboard and console printer. That was 4 different types of devices. I split a single byte up into 4 2-bit nibbles and that gave me 4 choices for each device type. This made it possible to extend tape input to a high-speed paper tape reader, user input and output to a CRT, and add a high-speed line printer. In addition, if there was another UART, we could add remote connections. This one little feature extended the utility of the monitor greatly and made it a big part of the character I/O subsystem of future disk operating systems. While developing ISIS later on, we used this feature to allow us to redirect the output of the PL/M compiler hosted on the corporate PDP-10, so that the hex code got downloaded directly into a development systems' RAM and then break into the debugger once it was loaded, ready to be debugged. This eliminated all use of time-consuming and error-prone paper tape generation from the time-sharing system. ISIS was developed using this system.

ISIS, a real operating system

In 1974, I started on the design of a disk operating system, intended for the upcoming MDS 800 multibus chassis. I had already designed and built a DOS at my previous company, and I had some idea what I wanted to do with the new DOS, and what I wanted to avoid.

My original plan was to have an internal hard drive of maybe 5 mb, along with a removable floppy disk or cassette tape for offline storage and distribution. I didn’t think a floppy was sufficient to do the job of supporting a full microprocessor development cycle. But, Marketing had different ideas. A hard disk was costly, it would require two different controllers, yadda, yadda, yadda.

The floppy disk had been developed by IBM as a distribution media for their big mainframe controllers. Their model was that an IBM tech would visit a customer site, use the floppy disk to install a patch or an upgrade to an installed controller, and then restart the controller with the upgrades. As such, it was a punch card replacement. At no time was a floppy disk to be relied upon for secure data storage. If one failed, make a new one.

IBM competitors scrambled to find out how the floppy disk worked, and IBM made that as hard as possible. I was told by the Shugart salesman that the first IBM design info they managed to get had the drive rotating counter-clockwise, and they started to design something similar. When the IBM production units showed up, they spun clockwise, thus setting the competition back by months, since they would have to redo the entire drive chassis. I may have the drive directions reversed, but you can get my point.

This was the device selected by Intel to use to create a microprocessor development system, over my objections.

When we started the design, we looked at the track and sector layout: 77 tracks, with 26 128-byte sectors per track, about 256k of storage. Note that the track count is a product of 2 prime numbers (11 and 7), and that the sector count is also a product of 2 prime numbers (13 and 2), not a power of 2 to be seen anywhere. This weird feature would cause untold complexity in any sector allocation algorithm, since it required 16 bit multiplies and divides to map that sector/track map into a continuous string of sectors, the kind of thing a file system wishes to work with. If you do the math, you can quickly determine that those multiplies and divides will always take a maximum amount of time, since you can’t do any clever shifts and adds based on the data, because the data is always ugly, due to the prime numbers.

We (Jim and I, the entire ISIS project team) had long discussions trying to come up with an elegant solution, and we failed. Table look-ups took up too much space, and calculations took too much time. We finally just used a 16 bit track/sector pair and created an allocation bitmap that padded out the odd sector count to 32 (I think), and filled in the dummy sectors bits as already used. This had the undesirable effect of storing the IBM mistakes directly in the file system, and made it difficult to add a hard drive to later versions of ISIS. I was no longer with Intel at that time, but came back as a contractor to work on that issue.

PIFS

Ridiculous drive layout aside, we took delivery of several Shugart drives and a tech went to work hooking one of them up on his bench. Once he got to the point where he could see data coming off the drive on his scope, he called me over to take a look at what he had accomplished. He had built a small wire-wrap board that used a UART chip to read the stream of bits off of a floppy track, block it into 8-bit bytes, and then generates an interrupt, where his test program could grab the byte and stuff it into an array in memory, and then did it again. What he had was a primitive disk controller. With some software, you could set up a write command and it would write out a sector of bytes in the same manner. This was not a sophisticated disk controller by any stretch of the imagination, far from it. Basically, you could start writing a sector 0, track 0, and it would keep writing and incrementing until if reached the end of storage at track 76, sector 25. It did not support any kind of random access.

At the same time, there were many contentious meetings with Marketing over the amount of effort and budget needed to produce a ‘real’ disk operating system, versus what a ‘microprocessor development engineer really needed’. One of the Marketing people remarked that the tech already had the disk controller working. Jim and I decided we would test that idea.

We used the ROM BIOS that I had built over the past year, adding an EPROM of code that mapped the tech’s crude disk controller writes into one of paper tape punch channels and reads into one of one of the paper tape reader channels. When hooked up to the text editor I had written in PL/M, I could write a small program in memory, save it by ‘punching’ it to floppy disk ‘tape’, close the file, and then read it back.

We called this “Papertape Information on Floppy Storage”, or PIFS, and presented it to Marketing. They asked about more than 1 file on the disk, our answer was “No, can’t do that”, it was really just a mapping of a string of bytes to/from a floppy. Marketing reluctantly agreed that this was shooting too low, and that Intel could be embarrassed if they attempted to market this solution. We obviously agreed. They asked what we called this piece of work, and we told them “PIFS”, which stood for the “Pie In The Face System”, since it would look like something the 3 Stooges had brought to market. Marketing decided to let us go ahead and design something more substantial.

Despite all the aggravation provided by Marketing, I really enjoyed writing ISIS. We decided early on that we would write the entire system

as a single PL/M program, since all we had to work with was the PLM8 compiler (The PL/M80 version would become available before product release), and neither of us relished writing and debugging 8kb of assembler code. So, we were very careful with resources. We dropped all character I/O out of the DOS, and used the ROM monitor to handle keyboard and display chores. We did not use the 8080 interrupt system, because using it caused a code explosion due to the requirement for multiple buffers and other resources like low memory interrupt vectors. We ended up with a single sector buffer which handled all I/O, to save memory.

The final ISIS memory footprint was 12 kb, which made running an 8k assembler in a 16k ram system an impossibility. Marketing had a melt-down. After a great display of angst, Marketing finally agreed to ship a 32k RAM minimum ISIS dual-floppy system. For this, they demanded that I produce an assembly language version of the OS, which could fit in 8kb (because PL/M was so ‘inefficient’). This version would be sold with a single floppy disk (still in the dual chassis, but with a blank plate covering the missing second drive).

The Marketing team presented this dual strategy solution to the sales force, and someone (my hero), said “But what happens if your dog pisses on your boot disk? You can’t make a copy!” The 16k RAM requirement stayed, but the single drive idea was dropped. I didn’t see that error repeated until the Macintosh in 1984. But, the Mac was marketed as a toy, and they got away with it.

After ISIS was released, written in PL/M, consuming 12k of ram and supporting an assembler and text editor (also written in PL/M), all fitting into a 32kb ram configuration, I did produce a hand-built assembly language version. I found that the PDP-10 timesharing system also supported a Simula compiler, which is a nice Algol-like language, so I used that to build a dis-assembler to convert the generated PL/M code back to assembler, which I then hand-tuned to squeeze 12k bytes into an 8k bag. I had to drop a random access ‘seek’ feature from the file system, but neither the assembler or text editor depended on that feature. It turned out that the compiler was not that inefficient, it just used a few instruction patterns and did not do much optimization. I got the space back because I removed features in addition to hand turning the code. We sold two of those systems.

ISIS did very well in the market and had a stunning 0 bug reports over the first year.

For reasons unknown to me, Intel had reduced the work being done by Gary Kildall, the developer of the PL/M cross-compiler. If I had to make a supposition, I would say the new software manager wanted tight control over all software work being done by and for Intel Corp, and independent consultants made him experience loss of control. That manager was “wound a bit tight”.

Gary did manage to get one of the early production MDS 800 systems, with the disk controller we built for the ISIS project, trading for something Marketing wanted done, and the development of CP/M proceeded in parallel with our work on ISIS. Gary and I discussed possible solutions for doing file allocation and I/O, and how to manage the disk controller, and such like, but each DOS was developed separately.

That done, it was time to make the PL/M compiler run on ISIS.

But first, the 8080 vs. The Motorola 6800.

Intel Marketing was worried about the 6800, because it presented a much more sane instruction set than the 8080, and that was hard to argue against. So, Intel started touting the 8080’s speed relative to the 6800.

There was a particular free-lance technical writer who wrote articles showing how the 6800 could beat an 8080 in an assembly language

face-off. My second level manager was livid about this guy, and challenged him to a code-off. They agreed to a small set of computing tasks that each CPU would have to do, and then they would tote up the cycles used. The loser would have to buy dinner for the winner at the Blue Fox, an expensive French restaurant in San Francisco. This manager came out of his office down the hall, shouting about this tech writer and said that he needed someone to write the winning code for the 8080, and looked directly at me. Cripes.

To make a long story shorter, I would point out that the fastest way to move 16 bits in an 8080 is to push it on the stack. The stack pointer is a 16-bit register, whereas all the other general registers were 8-bit pairs (BC, DE, HL). Since one of the algorithms in the contest was a string copy, I simply pushed 2 bytes at a time onto the stack, ignoring the fact that you can’t actually use a trick like that if you also want to have subroutines. But, we won, and the tech-writer had to take my wife and I to a nice dinner.

The competition continued over time, with the 8085 being developed as basically an 8080 with some process improvements that gave it a higher clock speed. When the design was presented to the software group, it turned out that the chip guys had added a couple of instructions to the chipset, and wanted us to add those instructions to the assembler. The instructions had little real value, and no one could come up with a solution for backward compatibility, since 8085 code would not work in an 8080, so we absolutely refused to support the new instructions, and they still remain buried in the old 8085 chips. I felt we got some revenge for the ‘dog pee DOS’ of the past.

And now, ISIS-II, the whole shebang

The Software Group at Intel really wanted to take programming languages further to support the microprocessor chip war, so first, we had to have a real, native 8080-based compiler. In addition to the significant effort to build the compiler, that also implied a lot of other pieces, like a linker, an object module definition, and a new assembler that could generate the same kind of object module as the compiler. Add to that the need to support the In-Circuit Emulator under development, and we had a big, complex project.

I figured I had it quite easy, since all I would have to do was add support to the ISIS loader to handle new object formats, and also recompile all of ISIS with the new compiler. No big deal. Unfortunately, the management decided they needed a single Release Chief to make sure all the parts got done on time and integrated properly into the whole. That turned out to be my job.

We had contracted with an outside software house to build a new compiler, and that was going well. These guys had a good compiler-writing system and methodology, and they proved that they could make a compiler run successfully in the 64k address space of an 8080. The new compiler was a joy to work with, and Jim and I made short work of the port of ISIS to the new compiler system.

A good friend of mine, Bruce, got the job of developing the Object Module Format (OMF) for the 8080, and the upcoming 8086. This piece of work would live on for years, since MS-DOS used it with little or no modifications as the EXE format. It was a good piece of work and it was updated in 1985 to support the 80386 32-bit flat address mode. It probably was updated again when the 64-bit architecture came along, but I wasn’t in the loop at the time.

The OMF gave us a good roadmap for the design of the linker, which knitted object modules together, and the locator, which processed the relocatable code into a fixed memory image, with a defined entry point, since that was how ISIS handled load modules. The locator was also built to enable mapping of a linked module into a set of EPROM images, so they could be programmed by a prom-burner.

The assembler proved to be more of a headache than some of the other more complex programs. It became the single biggest risk to meeting our delivery dates. I checked with the 2 person team working on the assembler, and one of them started complaining about floppy disk errors causing loss of data and therefore causing the time slip. We had probably a dozen engineers writing code all day long, and these two were having by far the most problems. I looked around their shared office, and saw diskettes, out of their storage sleeves, spread all over any horizontal surface. Sitting nearby was an ash-tray, full of cigarette butts (this is 1975). I pointed out that KEEPING THE DISKS IN THEIR SLEEVES was a good idea, and that SMOKING NEARBY AND DROPPING ASHES ON THE DISK SURFACE, was the probable cause of their error rate issues. I let it be known that that would stop now and that they had better clean up their office. They did bring the assembler in on time, just, but it continued to produce more bug reports than the rest of the release combined.

The In-Circuit-Emulator (ICE) was probably the most complex software code in the ISIS-II release. It was all new, and handling issues of high-speed demands and complex timing. I felt very lucky that the development team was a seasoned group of developers who knew their jobs, and they delivered on time.

Being a Release Team Leader was not always fun.

Other Intel CPUs

The Software group at Intel was always keen on using silicon to simplify complex operating system and language issues, so there was always another CPU design in the works. Everyone agreed that the 8-bit, 64k world of the 8080 wasn’t up to solving all the world’s software problems, but we didn’t agree on what the 8080 successors should be.

The first new architecture proposed was the “PL/M Machine”, a CPU which could execute the pseudo-code emitted by the compiler. The initial design was going to be implemented with the 3000 bipolar chips, so it would be a board level product, not a chip. The 3000 chipset was weird, with instruction decoders that had the side effect that there was no program counter, each instruction pointed to a successor or set of successors. This design created one very complex cross-assembler, and the 3000 chipsets did not do well in the market being gobbled up by the AMD 2900 chipset, so the project was dropped.

There was a lot of interest in the Plessey 250 computer, a ‘capabilities’ architecture. It had been developed in England as a highly-reliable telephone switch. To people intrigued by computer architectures, it was a thing of beauty. But, it was very complex, and many wondered if something more like a general register architecture, with more and larger registers, and a bigger address space, wouldn’t be a better product for Intel to build. The idea of a capabilities architecture did not go away, it continued in kind of a stealth mode, and eventually became known as the iAPX 432. Years later, I was talking to my friend Bruce, who had been negotiating a compiler contract with the folks in Oregon, building the new ‘432’ chip. He told me that they had situations where the CPU would go into its internal microcode to determine what to do next, and leave the bus idle for periods of over 9 ms. In early ‘80s microprocessors, that was forever. I figured that Intel had ‘bitten off more than it could chew’ on that processor design. A CPU that ‘contemplates its navel’ for that long is not going to be a winner.

What was finally approved was the 8086. It would have a 16-bit instruction set, more registers, and a larger address space. But, Intel management required it to fit into a 40-pin DIP, same as the 8080. To many of us, that was beginning to look like a straightjacket, but the decision had been made. What’s more, the staff assigned to do the design and implementation was two people, a good software guy, and a good chip guy. And, they had 15 months to finish the job.

The 8086 did come in on time, and it fit the management criteria. It had some very weird multiple-register addressing techniques that allowed addressing beyond 64k bytes. It would prove to be the powerhouse that Intel needed to dominate the industry, and the very good fortune to become the basis of the IBM PC architecture. But the 8086 was not a thing of beauty.

As the 8086 work commenced, the ISIS-II system was completed and I looked around for a new project. I found that the new Single Board Computer division was cranking out 8080-based single-board computers, using the Multibus architecture of the MDS-800 system as the basis of the board designs. I looked at the SBC 80/10, the first product, and was not impressed. It was a simple design, but it didn't have much of an interrupt system, which I considered a necessity, and it would require another whole board of peripheral components to get any useful work done.

The SBC 80/20, however, was another matter. This board had several new peripheral chips: an 8251 UART, 8253 interval timer, and an 8259 interrupt controller.

The single-board-computer business did not have any clear software development needs that weren't already covered by ISIS-II, with one major exception: there was a need for a small real-time executive that could coordinate interrupts and time-based scheduling. This was something that most of our surveys about the SBC market identified as needed.

So, how do we get one of those?

I spent time with one of the smarter people in the latest crop of Ph.D. software engineers we had just hired, and we talked about Dijkstra's THE operating system, which used semaphores to control access between independent processes in a multiprogramming environment. Maybe we could do something like that for the SBC 80/20.

I took on this task and decided to focus on the two basic primitives, 'send' and 'wait'. If a process issued a wait function on a control structure, it would block and give up control until some other process performed a send function on the same control structure.

I called the control structure an 'exchange' instead of semaphore and added the ability to provide an optional timeout value so that a process would not get blocked forever, but it could get control back after the prescribed 'wait-time' expired. The 8253 counter-time chip gave us the timer we needed.

The exchange could enqueue an incoming wait operation on two lists. The first was a simple FIFO, to make sure that no process could gobble all of the processing time. The second list was ordered by the respective wait-time values, which got inserted into the list by subtracting time off the wait-time until it fits between other pending wait-time values. This meant that the clock only had to decrement one counter at the top of the list, and when the counter went to 0, it would dispatch the process at the top of the stack. If a message was received for a given process, that process would be removed from the timer list by adding its remaining time to the counter below it in the stack. This made for a very efficient context switch that didn't spend very much time inside the kernel and gave us uniform dispatching whether the cause was a message arrival or a timeout.

In addition to standard send and wait operations, we wanted to tie the 8259 interrupt controller into the system, so that a software developer could have an interrupt mapped into a send operation, thus making the interrupt system a part of the multiprogramming environment, instead of an add-on. I called the project RMX-80. This work was done in a few months and it fit in 2 kilobytes of EPROM on an SBC 80/20 motherboard.

RMX-80 was my last project at Intel. I resigned in 1977 to pursue other

interests, including microprocessor software consulting.