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 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 headcount 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. Gary agreed, I was hired, and started at Intel on the first working day of 1973. Gary 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. 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 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.

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 Intellec 4 computer had quite a bit of logic in it to simulate 4004 roms in ram, needed for a development system. It was an interesting design, but not a computer that would support any kind of operating system.

I had to modify the assembler program, written to fit into 2 1K eproms, so that it could recognize ‘mark’ parity, 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 written self-modifying code, something very hard to do in a Harvard architecture. My proof was when I made sure the parity bit was on, the assembler died with errors. The consultant showed me where he had used parts a few bytes of executable code as storage, so that changing the parity changed his logic. He found a work-around and I applied it. End of 4004 work.

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 cpus 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. 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 Intellec 8 computer could not also be a debugger, which really limited what it could do.

Things like dumping a block of RAM to the teletype were supported, along with typing in code in hex and then jumping to the entry point at 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 link 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 usually meant 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, then the user could read in 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 timesharing system. We also sold the compiler in source form on magnetic tape for installation on chip 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 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 undo threat to the development time line. 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.

Intellec8 mod 80. The microprocessor group wanted to build a complete new development box for the 8080, but management insisted that we modify the Intellec 8 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 Intellec 8 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 return 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. 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 workspace 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.

The terminal used on Intellec 8 mod 80 systems was usually 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 developer’s box RAM and 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.