Fairchild Symbol Computer

Stanley Mazor
IEEE, Sr. Member

Editor: Anne Fitzpatrick

In 1965, Gordon Moore of Fairchild Semiconductor published what is known as Moore's law: his observation and prediction about the growing density of IC chips, and the precursor to large-scale integration.[1] Moore's predictions about LSI chips raised questions about how complex chips would be designed and used.[2–6] As the Fairchild R&D director, he initiated several research projects addressing the following LSI design and use issues:

- computer-aided design tools (CAD) for LSI chips
- cellular logic chips using standard cells
- gate array logic chips
- packaging for LSI chips
- semiconductor memory chips
- computer design with LSI chips

As testimony to Moore's vision, most of these Fairchild projects subsequently spawned new industries and companies, such as LSI Logic and several CAD companies. And, in 1968, Moore and Robert Noyce started Intel.[7,8] They reasoned that chip density was increasing geometrically, but that IC package pins were growing less than linearly: IC pin limitations could be an obstacle. Memory chips required few pins, were regular in design, and the market for computer memory systems was growing.[9] The advent of LSI chips and memory later fostered the development of the single-chip CPU at Intel.[10–12]

The earlier computer development project in Fairchild's R&D, however, did not succeed and is therefore not well known. I worked as a designer on both the Fairchild and Intel projects, and I share some insights about them here.

Programming at Fairchild

In 1964, I joined Fairchild Semiconductor in Mountain View as a Data Processing Department programmer using several machines—the IBM 1620, 1401, and 360, and the SDS 930—in assembly language, Cobol, and Fortran. I wrote software that processed sales order records with a varying number of variable-length fields. Customer orders contained an arbitrary list of products, each with a varying number of requested delivery dates. Conventional programming languages couldn't handle variable-length data, and few could handle data arrays of varying size. Most systems required fixed-dimension arrays. Cobol barely supported varying sized records, with the "occurs depending on" option, and few systems supported nested variable-length fields. These limitations forced us to write the sales order processing software in assembly language, so the code wasn't machine independent.

As it happened, Gordon Moore's R&D computer development group wanted a practicing programmer to join their staff, so I transferred to R&D in 1966.[13] Our project team developed a new language, Symbol, and the hardware for *directly* executing it. The Symbol computer overcame the data handling problems that I'd encountered with the sales order entry software. At the expense of extra hardware, the Symbol computer removed "artifacts"—introduced to improve execution speed—of conventional computing languages. Undesirable programming artifacts, for example, included static data types and type declaration.

Computer design project

One of Moore's R&D groups, Digital Systems Research (see Figure 1) under the direction of Rex Rice, focused on chip and system packaging. This group had developed the original Dual In-line Package (DIP). Earlier IC packages had flexible (flimsy) wires, so mounting these IC chips on printed circuit boards (PCBs) was difficult. The DIP, however, had two rows of rigid pins in two regularly spaced lines, which facilitated automatic handling of ICs and PCB insertion equipment.

Our computer design project tackled the next level of packaging, using tens of PCBs with hundreds of IC chips per board. But what kind of experimental computer should we build? We soon embarked on the design of Symbol, a "radical" large-scale, high-level language, time-sharing computer with a virtual memory.[14] One objective was to use an order of magnitude more hardware than conventional computers did. Most college engineering courses stress logic minimization, so *maximizing hardware* may sound absurd. But in 1966 we assumed that LSI chip costs would fall dramatically, and that by using more hardware we might overcome some limitations of conventional computers—including variable-length data handling.[15–18] The Symbol's logic design was hardwired, with no control store or microprogramming of any kind. Another objective was to partition the CPU into several functional units that we could build as LSI chips.[19]

Although we thought about eventually using LSI chips, in the Symbol prototype each of 100-plus two-sided PCBs (16″ × 20″) had about 200 conventional gates and flip-flops (Fairchild CTL). Since an LSI chip would have few pins, all of Symbol's functional units were constrained to have a small number of interfacing signals to meet the LSI chips' ultimate packaging constraints. In retrospect, these LSI considerations were appropriate; they were the compelling factors for the microcomputer's development at Intel three years later in 1969. However, a few key differences distinguish the assump-

Figure 1. Fairchild Digital Systems Research Group, Symbol IIR computer prototype's chassis and design team (1968). (Courtesy of Fairchild Digital Systems Research Dept.)

tions underlying the Symbol and the Intel microcomputer. First, the Symbol pin limit was arbitrarily set at 100, whereas the first Intel microprocessor, the 4004, had only 16 pins. Next, the Intel microcomputer had only a 4-bit data word and a dramatically scaled-down instruction set in order to squeeze the entire CPU onto a single chip.[20–22] Finally, the Fairchild Symbol project was a "super smart" 64-bit computer that would have required a large number of specialized LSI chips. (I found the contrast stark when I left my work on a "smart" large computer at Fairchild in 1969 to work on the 4004—a small, dumb microcomputer—at Intel.[12,23,24])

Symbol computer overview

Fairchild's Symbol computer had both the compiler and operating system built with hardware, not software. Several of us on the Symbol design team were experienced programmers, familiar with the high-level languages Cobol, Fortran, and Algol, and with the inner workings of compilers and operating system software. Accordingly, many hardware mechanisms we built were based on known software techniques. For example, the one-pass (hardware) translator generated a symbol table and reverse Polish code as in conventional software interpretive languages. The translator hardware (compiler) operated at disk transfer speeds and was so fast there was no need to keep and store object code, since it could be quickly regenerated on-the-fly. The hardware-implemented job controller performed conventional operating system functions. The memory controller provided a virtual memory for variable-length strings and is described herein.[23]

Dynamically varying variables

Symbol data variables were of arbitrary size and could change type and size during program execution. The following Symbol code fragment emphasizes this point—the hardware executes three consecutive assignment statements:

```
x <= "a";
x <= 1.3333333333333333333333333333356;
x <= "now is the time for all good men to
come to the aid";
```

In the first case, the variable *x* is assigned a simple string character. Thereafter, the *x* value is replaced by a long number, and finally by a long string. The Symbol hardware provided for dynamic allocation and deallocation of storage for variables like *x*, as well as marking the data type in the symbol table during program execution. Most conventional software languages don't allow variables to change type or size during program execution.

Symbol floating-point arithmetic hardware permitted up to 99-decimal digits of mantissa, and the precision was dynamically user-determined. Many computers only offer either single or double precision; the IBM System/360 Model 44 users controlled floating-point arithmetic precision via a front panel dial.

The Symbol hardware supported arrays of dynamically varying sized data, unlike most environments wherein arrays hold identically structured data (e.g., a linear list of floating-point numbers). In Symbol, each vector of an array could be a different length, and each data entry could be a different size. The following assignment statement changes the scalar variable *x* into a three-element linear array:

```
x <= < 1.2 | "now is the time for all" |
1.333333333333356 >;
```

This array has 3 fields; in this external representation special field markers '|' separate the three data values. Each array element is a different size and can be of a different data type. Indexing or referencing into an array in the Symbol language is conventional, x[2], but locating variable length data in real time is difficult and linear searching is slow.

Machine organization

Consistent with the design objectives, we built Symbol with multiple (hierarchical) functional units suitable for LSI implementation. The principal instruction fetch and interpreter that executed the object code was the instruction sequencer. The IS in turn called on several other CPU functional units: FP, for floating-point arithmetic; SP, for string processing; RP, for variable reference processing; and MC, for memory reading and writing and management.

Communication between these units was facilitated via separate buses using control response codes. Typical service requests and communication were between the IS and MC; the IS and the FP and SP; and the IS and RP, but these units also made requests of the MC for memory operations.

Memory organization

Recall that Symbol's memory controller hardware implemented a virtual memory system that supported variable-length data. The MC allocated virtual memory pages from a free-page list to a user's memory pool, assigned data storage within a job's pages, and did garbage collection. Each memory page held twenty-eight 64-byte groups (with additional group-link words in each page). Except for simple scalar variables stored within the symbol table, the smallest unit of user data was a single group. Larger data used multiple groups that used forward and backward links. The MC could traverse data in either direction. Similarly, arrays utilized as many linked groups as needed. The MC provided a high-level string storage memory system and served other processors by executing "primitive" operations. Here are four of the interesting memory control operations:

- *Assign group—AG*. This operation allocated a new group from the storage free list, (allocated a new page if needed), and returned the address of the first word in the group.
- *Store and assign—SA*. This operation stored the data word at the address sent, and returned the next sequential address for the following word in the same group. If the group was full, it allocated another group from the free space pool, and linked the groups together, and returned the address of the next free location in the group.
- *Fetch and follow—FF*. This operation fetched the data word at the address given and returned the next sequential address of data, if in another group, it followed the link. If this was the end, it returned a null address.
- *Delete string—DS*. Using the address sent, this operation linked all the groups in the string into the free space list for this job.

The Symbol computer was complex, but the hierarchy layering of functional hardware units concealed low-level details from higher-level units. Accordingly, for the floating-point (FP) unit, memory could be treated as holding arbitrary long data. The MC handled the overhead of following links. As an example in the FP arithmetic unit (most significant digit handled first) that I designed, the output result, up to 99 digits could occupy multiple data words. The FP started with a request to the MC to do an assign group, and as each output word was developed by the FP, the FP called the MC to store and assign each successive data word into memory. Temporary work spaces for partial products or quotients were similarly requested from the MC; afterward, the FP called the MC for a delete string for the temporary variable-length string fields that were no longer needed.

Summary

The overly ambitious Symbol computer project delivered a prototype (to Iowa State) without using LSI but was commercially uninteresting, chiefly because LSI's impact on computer CPUs was reflected in dramatically lowered hardware costs.[25] In a direction completely opposite that of the Symbol computer's hardwired logic design, the industry turned to microprogramming to implement complex logic functions—a more practical, cost-effective approach. More recently we have seen LSI used for graphics, but chip designers are challenged on how to best use more hardware. Hierarchical hardware systems, like the neglected Symbol computer, aren't often considered yet are worth further analysis in order to simplify complex design.

Intel's single-chip CPU was the result of scaling down a computer's architecture, not scaling it up.[25,26] Intel later also put operating system functions into a CPU, but abandoned that effort.[27] Moore's LSI passion affected computer design, not through Symbol, but ultimately via the microcomputer.

Acknowledgments

Key contributors were Russell Briggs, Bill Smith, Gil Chesley, Larry Gerould, George Powers, Ted Laliotis, Brooks Cowart, Tom Cook, Steve Lundstrom, and Rex Rice.

References and notes

1. G.E. Moore, ''Cramming More Components onto Integrated Circuits,'' *Electronics*, 19 Apr. 1965, pp. 114-117.
2. F.G. Heath, ''Large Scale Integration in Electronics,'' *Scientific Am*, Feb. 1970, pp. 22-31.
3. R. Petritz, ''Technological Foundations and Future Directions of Large-Scale Integrated Electronics,'' *Proc. Fall Joint Computer Conf*, (FJCC), AFIPS Press, 1966, p. 65.
4. P.E. Haggerty, ''Integrated Electronics—A Perspective,'' *Proc. IEEE*, vol. 52, no. 12, 1964, pp. 1400-1405.
5. H.G. Rudenberg, ''Large Scale Integration: Promises versus Accomplishments—The Dilemma of Our Industry,'' *Proc. FJCC*, AFIPS Press, 1969, vol. 35, p. 359.
6. A.W. Lo, ''High-Speed Logic and Memory—Past, Present, and Future,'' *Proc. FJCC*, AFIPS Press, 1968, vol. 33, pp. 1459-1465.
7. G. Bylinsky, ''Little Chips Invade the Memory Market,'' *Fortune*, April 1971, pp. 100-104.
8. L. Vasdasz et al., ''Silicon Gate Technology,'' *IEEE Spectrum*, Oct. 1969, pp. 27-35.
9. *Intel 3101 64-bit Static RAM Data Sheet*, Intel Corp., 1970.
10. G. Bylinsky, ''Here Comes the Second Computer Revolution,'' *Fortune*, Nov. 1975, pp. 134-138.
11. R. Noyce and M. Hoff, ''A History of Microprocessor Development at Intel,'' *IEEE Micro*, vol. 1, no. 1, 1981, pp. 8-21.
12. S. Mazor, ''The History of the Microcomputer—Invention and Evolution,'' *Readings in Computer Architecture*, M.D. Hill, N.P. Jouppi, and G.S. Sohi, eds., Morgan Kaufmann, 2000, p. 60.
13. S. Mazor, ''Programming and/or Logic Design,'' *Proc. IEEE Computer Group Conf.*, IEEE Press, 1968, pp. 69-71.
14. W. Smith et al., ''The Symbol Computer,'' *Computer Structures: Principles and Examples*, D. Siewiorek, G. Bell, and A. Newell, eds., McGraw Hill, sect. 7, ch. 30, pp. 489-507.
15. J. Holland, ''A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously,'' *Proc. Eastern Joint Computer Conf.*, 1959, pp. 108-112.
16. R. Noyce, ''A Look at Future Costs of Large Integrated Arrays,'' *Proc. FJCC*, AFIPS Press, 1966, p. 111.
17. M.E. Conway and L.M. Spandorfer, ''A Computer Designer's View of Large-Scale Integration,'' *Proc. FJCC*, AFIPS Press, 1968, p. 835.
18. L.C. Hobbs, ''Effects of Large Arrays on Machine Organization and Hardware/Software Trade-offs,'' *Proc. FJCC*, vol. 29, AFIPS Press, 1966, p. 89.
19. N. Cserhalmi et al., ''Efficient Partitioning for the Batch-Fabricated Fourth Generation Computer,'' *Proc. FJCC*, vol. 33, AFIPS Press, 1968, pp. 857-866.
20. *MCS-4 Micro Computer Set*, data sheet # 7144, Intel Corp., 1971.
21. M.E. Hoff and S. Mazor, ''Standard LSI for a Micro Programmed Processor,'' *IEEE NEREM '70 Record*, Nov. 1970, pp. 92-93.
22. S. Mazor, ''A New Single Chip CPU,'' *Proc. Compcon*, IEEE CS Press, 1974, pp. 177-180.
23. W. Smith, R. Rice, and S. Mazor, *Hardware-Oriented Paging Control System*, US patent 3,647,348, to Fairchild Camera and Instrument Corp., Patent and Trademark Office, 1972.
24. M. Hoff, S. Mazor, and F. Faggin, *Memory System for a Multi-Chip Digital Computer*, US patent 3,821,715, to Intel Corp., Patent and Trademark Office, 1974.
25. S. Mazor, ''VLSI Computer Architecture Issues,'' *Process and Devices Symposium*, Electron Devices Group, 1981.
26. S. Morse et al., ''Intel Microprocessors 8008 to 8086,'' *Computer*, Oct. 1980, pp. 42-60.
27. S. Mazor and S. Wharton, ''Compact Code—IAPX 432 Addressing Techniques,'' *Computer Design*, May 1982, p. 249.

Readers may contact Stan Mazor at stanmazor@sbcglobal.net.

Readers may contact department editor Anne Fitzpatrick at annals-anecdotes@computer.org.