The S-Machine, an Architecture for Symbolic Processing

Cristian Băleanu, Dan Tomescu

Ro-Micro, Brașov, 28 June 2024

Context

Computer architecture was a remarkably active field of research in the 1980s, primarily due to impressive advances in circuit technology. Not only was technology enabling the implementation of faster and more complex designs, but it was also lowering the cost barriers to building experimental models that played a major role in testing novel architectural concepts. Much of the flurry of research of that time was related to investigating the relationship between architecture and programming languages, i.e. identifying the kind of hardware/software interfaces that would optimize the execution of (compiled) code written in various high level languages [1]. RISC architectures [2], Lisp and Prolog machines [3, 4], tagged architectures [5], array processors [6], recursive architectures [7]: these are but a few of the multitude of designs that were being explored and experimented with. Some of those solutions were later incorporated into commercial computers, others proved to be unrealistic, but most of them helped shape the current landscape of practical computing.

Our interest in computer architecture dates from the late 1970s, when we were part of a Bucharest Politehnica University team that designed and built the production prototype of DIAGRAM, a graphics workstation produced and commercialized by FEPER. DIAGRAM was built around a Z80-based bi-processor machine featuring a hardware accelerator for graphics. Our responsibilities included, on the one hand, the design and implementation of the main (Z80-based) processing unit and associated hardware I/O drivers for devices such as floppy disks and, on the other hand, the design and implementation of an operating system kernel with support for pre-emptive multitasking.

Projects

After the DIAGRAM project we joined ICPE, a research institute for electrical engineering, to do applied research work on innovative computer architectures, with a particular emphasis on the potential of functional and logic programming languages to harness the power of the rapidly evolving circuit technology. Our goal was to build, from the ground up, a complete solution for a real world problem that would be intractable using stock hardware. The only way to do that was to find a customer who would have the resources to fund our work. Some local market research brought us in front of such a customer, Petromar, a Romanian company doing off-shore oil drilling and exploration work in the Black Sea. Petromar's CEO had been impressed by work done in the West with expert systems for oil drilling, but commercial embargoes in force at that time (1985) did not allow him to consider the purchase of a turnkey solution. He accepted our project proposal to build an original solution around a novel architecture, the *S-machin*e [8, 9, 10], that would be capable of running the expert system (Prolog) code within the real-time constraints imposed by their application. The work was done under the umbrella of ICPE, that allowed us to build the team needed to do all the design and implementation work.

The S-machine was designed to be used as an "intelligent" hardware accelerator connected to a general-purpose computer through a high-speed interface and fully integrated with the host operating system. DEC PDP 11 running Unix and IBM PC were the two types of computers that could be connected to the S-machine. Code libraries written in C were designed and implemented to make the S-machine transparent to programs running on host machines, i.e. the S-machine was seen as a regular API (application programming interface).

Most of the code for the Prolog environment (user interfaces, translator, code editor,

debugger) was run on the host machine and was based on work done before the S-machine had been designed [11]. Implementing the Prolog runtime engine on the S-machine itself turned a conventional Prolog implementation into a high-speed system (while running tests it was not unusual to notice speedups of more than an order of magnitude).

In 1989, while the Petromar project was in an advanced stage, our team started to work on a new job. ITCI, a Romanian research institute for computing technology, was interested to bid for a request for a Lisp machine coming through Comecon (the Eastern block trade organization) from a Soviet research institute for the aerospace industry. ITCI asked us to join forces with them to write a project proposal that was accepted by the customer. As a consequence, part of our team started to work with the ITCI team on this new project and a new version of the S-machine was about to be built. The Lisp programming environment was on a path very similar to that successfully taken to build the Prolog system: implementing a Lisp runtime environment on the S-machine and combining it with modules developed for conventional implementations [12, 13].

1990 was a very eventful year in Eastern Europe. As a consequence, progress on our projects slowed down considerably, before coming to a halt (e.g. Comecon was disbanded and took with it our contract with the Moscow institute, but not before successfully reaching the first milestone).

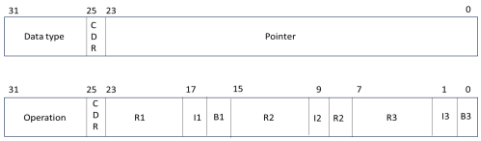
In conclusion, our S-machine projects were a success while they lasted. We managed to successfully build a hardware machine having a novel architecture together with its associated software and our work was fully funded by real customers. However, our goal to implement the machine in VLSI technology and turn it into a commercial product had to be abandoned.

The S-machine

The S-machine was a 32-bit computer with hardware support for symbolic languages like Lisp and Prolog, but able to run with no time penalty compiled code for programs written in procedural languages like C. Lisp implementations were usually inefficient because the control of program execution on conventional machines is implicitly sequential and memories are linearly organized, while in Lisp both data structures (lists) and evaluation procedures are recursive. We did not want to restrict our machine to Lisp, but similar problems were posed by Prolog, so we chose to design a general-purpose architecture with a reduced set of instructions and hardware support for recursive control strategies and compact list representation of both data and programs.

Given a list *l*, *car(l)* is a function that selects the first element of *l*; *cdr(l)* is a function whose value is *l* without its first element (e.g. if *l* = (a, b, (c, d)), *car(l)* = a, *cdr(l)* = (b, (c, d)). Lists are represented usually by means of cells made up of two pointers each, CAR and CDR. However, statistical studies carried out on large Lisp programs showed that if lists were linearized, more than 98% of list CDRs pointed to the next cell. As a consequence, a technique called *CDR-coding* [14] can be used to represent lists: every pointer P is represented in a separate memory word together with two additional bits encoding information about CDRs of list cells, i.e. “CAR is P, CDR is in the next word”, “CAR is P, CDR is nil (empty list)”, “CAR is P, CDR is the next word”, “the cell is relocated at P”.

In the S-machine lists are represented using CDR-coding, that unifies the management of lists and contiguous blocks of memory. Memory words are 32 bit long and contain a 24 bit pointer field (P), a two bit CDR field (used for CDR coding) and a six bit data type field identifying the type of data pointed to by P. When lists are used to store machine programs, the data type field is interpreted as an operation code, while the pointer field holds the operands. The representation of programs by means of lists implies that, unlike in the case of conventional computers, on the S-machine any program, regardless of the language it is written in, can be manipulated by any other program.



List representation of data/programs

A fixed value of the data type field is reserved for program lists: it indicates that the pointer field holds the address of a program list. Machine programs can therefore be nested lists, with instructions as "atomic" data. To execute such a program means to (recursively) traverse the list according to some discipline and execute individual instructions as they are encountered; the program list traversal strategy can be specified by the user (i.e language processor). Suppose that the S-machine has to execute a program list l. If *car(l)* is an instruction, it is executed and control is passed to *cdr(l)*, but if *car(l)* is a program list, the execution could follow either branch (note that enough information has to be saved about the branch that is not followed in order to resume its execution at a later stage). Let a register, *rl*, hold the starting address of a procedure to be invoked whenever the processor is faced with such a choice. Assuming that *car(l)* is a program list itself, a trapping mechanism saves both *car(l)* and *cdr(l)* in two registers, then control is passed to the address held in *rl*. The procedure stored at that address decides which program branch is to be followed and which one is to be saved; if the list under execution is empty, control is passed to a procedure pointed to by another register, *re*. Procedures pointed to by *rl* and *re* implement the traversal strategy of program lists; since both *rl* and *re* are general-purpose registers, their contents may be changed at will, hence the traversal strategy can be dynamically modified. Note that if the data type/operation code field contains the value reserved for the program list type, that value can also be seen as an operation code: it activates the trapping mechanism.

While the logical address space of the S-machine is continuous, the first 16 Kwords were implemented in fast memory. The machine had 128 general-purpose registers, out of which 64 were global and 64 local. The global register set takes up the first 64 words of fast memory, while the local register set specifies a window made up of any 64 consecutive memory words, with addresses calculated by adding an offset (derived from the register number) to the contents of a base register. The local register set can be used to hold local variables of functions and procedures: if the base register is appropriately set on procedure entry, then local variables are accessible via register references. For efficiency reasons, live contexts should reside in fast memory: it is up to compilers to generate code for dealing with fast memory management.

The S-machine has a set of 61 simple hardwired instructions with a uniform representation. Instructions are register-to-register (except for jumps and load/store instructions). Besides the usual data transfer, logical, and arithmetic instructions, two instructions implement the most frequently used list operations: *car ri, rj* and *cdr ri, rj*. A wide range of jump instructions, including 2^n jumps, some of them using masks to test selected bits of special purpose state registers, can be used for efficient dynamic data type checking, which is a vital part of the runtime support for symbolic languages.

At the hardware implementation level, most functional modules have a symmetric structure, resulting in increased efficiency of program execution. Main memory is interleaved while the fast memory is made up of two identical banks containing the same information (two simultaneous read operations per memory cycle can therefore be executed, with write operations altering corresponding addresses from both banks). Efficiency gains brought about by the dual hardware structure include simultaneous access to left and right operands residing in global registers and local registers allocated in fast memory, the execution of parallel operations in the two sections of the processor, and fast transfers of compact data blocks. Prefetching of instructions is

favoured by the compact list representation of programs.

The S-machine prototype was built in TTL-S and TTL-LS technology. The processor is sequentially controlled by a 100 ns single phase clock. The fast memory has a 100 ns cycle and is built out of TTL-S chips, while the main memory is implemented in MOS DRAM technology and has a 350 ns cycle.

References

1. K. Kavi et al, HLL Architectures: Pitfalls and Predilections, Proc. 9th Sym. Computer Architecture, 1982, 18-23.
2. D. A. Patterson, Reduced Instruction Set Computers, Comm. ACM 28 (1), 1985, 8-21..
3. D. A. Moon, Architecture of the Symbolics 3600, Proc. 12th Symp. On Computer Architecture, 1985, 76-83.
4. R. Nakazaki, Design of a High-Speed Prolog Machine (HPM), Proc. 12th Symp. On Computer Architecture, 1985, 191-197.
5. F. Gehringer, J.L. Keedy, Tagged Architectures: How Compelling Are Its Advantages, Proc. 12th Symp. On Computer Architecture, 1985, 162-170.
6. V. Zakharov, Parallelism and Array Processing, IEEE Trans. Comp. C-33 (1), 1984, 45-78
7. P.C. Treleaven, R. P. Hopkins, Recursive Computer Architecture for VLSI, Proc. 9th Symp. On Computer Architecture, 1982, 229-238.
8. C. Băleanu, D. Tomescu, An Architecture for Symbolic Processing, Information Processing Letters, 26 (4), 1987, 217-222.
9. C. Băleanu, D. Tomescu, A General-Purpose Machine with a List-Structured Memory, Preprint, ICPE, 1, 1987.
10. D. Tomescu, C. Băleanu, The S-Machine: Design and Applications, Computers and Artificial Intelligence, 7 (5), 1988, 461-470.
11. D. Tomescu, V. Secarin, The Pseudo-Prolog Programming Environment, 8 (2), 1989, 131-139.
12. D. Teodosiu, tLisp – A semicompiled Lisp on a Microcomputer, Computers and Artificial Intelligence, 7 (3), 1988, 193-202.
13. D. Teodosiu, HARE: An Optimizing Portable Compiler for Scheme, ACM SIGPLAN Notices, 26 (1), 109-120
14. W. J. Hansen, Compact List Representation: Definition, Garbage Collection, and System Implementation, Comm. ACM, 12 (9), 1969, 499-507.