A VLSI-CHIP-SET FOR A HARDWARE-ACCELERATOR FOR THE SIMPLEX-METHOD

Bernd Schütz, André Klindworth
University of Hamburg, Dept. of Computer Science
Troplowitzstr. 7, 2000 Hamburg 54, Germany

Abstract

We present a hardware realization of the Simplex Method, the central method of linear programming. We customized the algorithm regarding numerical stability (arithmetics) as well as hardware proximity. The resulting hardware is based on a parallel architecture with up to eight processing units, employing standard FPUs, RAMs and custom VLSI-chips. It has been designed for use in an IBM PC/AT environment.

Introduction

The Simplex Method, introduced in the late fourties by George B. Dantzig [3], is by far the most important algorithm within linear programming with many applications especially in the field of Operations Research [10]. The Simplex Method has a non-polynomial time-complexity. But experiences show that its average-complexity grows roughly linear with the problem size when applied to practical problems [3], hence its widespread use. Considering Borgwardt's statement in [2] that the Simplex Method "... is probably that mathematical algorithm, for which the most computation time on computers is spent" it is obvious that using an application-specific hardware to reduce computation time is highly desirable.

Besides just increasing the speed of computation, the limited precision of ordinary floating point arithmetic has also to be considered [1]. Commercial software for linear programming adapted to small computers [7] therefore usually restricts the problem size. But of course this is not an adequate way to deal with the problem of numerical stability. Based on this fact, alternative arithmetics like interval- [8], x/y-arithmetics or arithmetics with continued fractions were examined in preparation for the design of our hardware architecture. These arithmetics are very interesting from the scientific point of view. But they are not appropriate for practical applications. Therefore the implementation presented here uses a floating-point arithmetic that employs several carefully adjusted epsilontics to avoid the accumulation of round-off errors in the course of the computations.

Problem description

Fig.1 shows the mathematical model of the kind of problems solved with the Simplex Method: given non-negative variables $x_1, \ldots, x_n$ subject to a set of inequalities (constraints), and an objective function $f(x_1, \ldots, x_n)$. For example, the variables may represent different products to be fabricated and the objective function may allow to calculate the profit. The task is to find feasible values for $x_1, \ldots, x_n$ that satisfy all constraints and maximize the value of f.

$$\sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i, \quad i = 1, ..., m; \quad n \leq m,$$

$$\forall x_j : x_j \geq 0$$

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{n} c_j \cdot x_j \quad \rightarrow \quad Max,$$

Figure 1: mathematical model of a linear programming problem

When solved with the revised Simplex Method [5] (a special variant of the Simplex Method well-suited for a hardware-implementation), the data structures involved in the calculation are a matrix (containing the full inverse basis) and a few vectors. Thus, an application-specific computer architecture should allow to handle operations on the matrix and the vectors efficiently.

Parallelism

In order to get some important information about where and how calculations may be parallelized, we extensively analyzed the solution of a given set of examples we obtained from the Department of Ecomomics of the University of Hamburg. The results of this analysis showed, among other things, that for our class of problems a great many of the operands occuring in the course of the calculations are equal to zero (30 to 90 per cent). Considering that an arithmetic operation only needs to be initiated if both its operands are not zeroes, our objective has been the design of a computer architecture that avoids redundant operations whenever possible.

```
for i := 0 to n-1 do
  begin
    x := 0.0;
    for j := 0 to n-1 do
      if (A[i][j] <> 0.0) and (b[j] <> 0.0)
        then x := x + A[i][j] * b[j];
    c[j] := x;
  end;
```

Figure 2: Matrix-vector-multiplication, avoiding redundant computations

Fig.2 shows as an example a sequential program for multiplying a $n \times n$- matrix A by a vector b giving the vector c, which makes use of zero-operands to avoid performing redundant operations. Given a high rate of zeroes in A and b and an efficient mechanism for the detection of zero-operands, this program will probably run much faster than the trivial implementation that calculates $c_i = \sum_j A_{ij} b_j$, omitting the if-statement.

When the outer for-loop in Fig.2 is parallelized allowing concurrent computation of the c_i, still taking advantage of zero-operands in the depicted way implies that several processing units (PUs) are available, which may execute different instructions at the same time. The resulting scheme of parallel processing thus belongs to a restricted form of MIMD (Multiple Instruction, Multiple Data in the classification introduced by Flynn [4]), which is sometimes called SPMD (Single Program, Multiple Data) [9], indicating that the possibly distinct instructions executed by the PUs belong to the same program.

Of course, other schemes of parallel processing could have been chosen (e.g. solving several linear programming problems or computing different vectors in parallel), but regarding memory requirements, data-dependencies and efficiency of synchronization the SPMD-model has come out to be the most promising. Our computer architecture therefore allows efficient implementation of synchronization primitives, task-scheduling (which reduces to spreading different values of the loop-index i among the PUs) and detection of zero-operands.

Architecture

The overall architecture of our Linear Programming Accelerator (LPA) is depicted in Fig.3. From one to eight PUs share a common main memory. They are coordinated by a MCU (Master Control Unit) that provides task-scheduling and administrates the global resources shared by the PUs. Fig.4 shows the internal structure of one PU consisting of a LCU (Local Control Unit), an asynchronous FPU (Floating-Point Unit), and a BSW (Bus-SWitch), the latter realizing an interface from the LCU to the FPU and to the main memory interface. The LCU consists of a program-ROM, a register-file, a status-register, and an integer-unit. A fast application-specific address calculation unit is also provided, allowing all address calculations to be performed in at most two machine cycles. The instruction set executed by the PUs contains instructions for accessing the main memory, performing arithmetic operations, modifying the status-register, and initiating floating point operations in the FPUs. This standard instruction set has been expanded with commands to accomplish fast detection of zero-operands and to realize transitions between sequential and parallel phases of program execution. The latter is the most outstanding feature of our LPA and therefore is described in more detail in the following.

```
    SLIF(n);
10: Get_LI(i); if i < 0 then goto 20;
    <calculate c[i]>
    goto 10;
20: join;
    <sequential part>
    wait;
```

Figure 5: Parallelized outer for-loop from fig.2

While executing sequential parts of the program, only one PU is working. When entering a parallelized for-loop this PU executes a SLIF-command (Set Loop-Index and Fork), setting the loop-index register in the MCU to the upper bound of the loop-variable i and activating the other dormant PUs. All PUs resume program execution at the instruction following the SLIF-command. When entering the body of the parallelized for-loop, a PU needs to be sent a unique value for the loop-variable i. The PU requests this value executing a Get_LI- instruction (Get Loop-Index). The MCU contains a queue of all Get_LI-requests and serves them one at a time by first decrementing i and then delivering i to the requesting PU. When a PU is sent a negative value i, it recognizes the end of the loop and executes a join-command which makes all but the first of the PUs inactive. This way, the first PU leaving the parallel section immediately goes on with the next sequential part. At last, a wait-command is provided, allowing this PU to wait for the end of parallel instruction streams pending from the previous SLIF. As an illustration, Fig.5 shows the program from Fig.2 after parallelization using SLIF-, Get_LI-, join-, and wait-commands.

Performance

Tab.1 shows the computation time needed by our LPA for the solution of some typical linear programming problems. These results were obtained through computer-simulations on the register-transfer level for an LPA equipped with four PUs running at 12 MHz and using FPUs of type Motorola MC68881 running at 16 MHz.

Problemsize			**CPU-time/Pivotstep [ms]**		
(constr.	x	var.)	LPA-4	Apollo DN3000	SUN SS2
61	x	69	11.7	318	8.4
118	x	52	13.9	625	20.5
88	x	116	12.1	524	14.1
166	x	77	67.2	1899	59.3
292	x	251	201.7	4168	230.8
94	x	88	32.0	692	38.0

Table 1: Some performance analyses

For comparison, the computation time on a Sun SPARCstation 2 and an Apollo DN3000 running with our algorithm are included in Tab.1. It can be seen from Tab.1 that the LPA, although using the same type of FPU and running at the same clock rate, is more than twenty times faster than the Apollo DN3000. On the other hand, the comparison with a Sun SPARCstation 2 shows the progress in technology. Nevertheless, the comparison with the Apollo demonstrates the speed-up that can be achieved by an application-specific architecture.

Realization

The realization of the custom VLSI-chips (MCU, LCU, and BSW) is currently under work at the University of Hamburg using the SOLO 1400 [11] and SOLO 2030 [12] design systems from ES2 European Silicon Structures. Fig.6 shows the physical layout of the LCU-chip. The MCU contains most of the host interface (excluding line drivers) to allow the LPA being easily used as a co-processor in an IBM PC/AT environment.

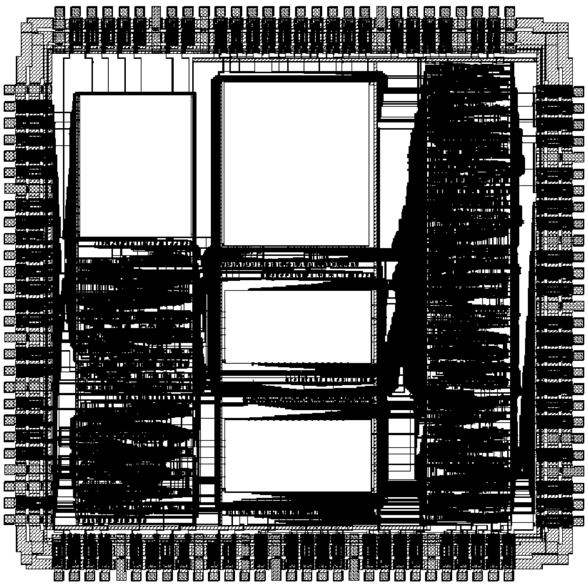

Figure 6: The LCU-chip (1.5μm CMOS, 42 mm^2)

References

[1] G. Bohlender. *What Do We Need beyond IEEE Arithmetic?*, in: Computer Arithmetic and Self-Validating Numerical Methods, Academic Press INC, San Diego, 1990.

[2] K. H. Borgward. *The Simplex Method - A Probabilistic Analysis.* Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1987.

[3] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, 1963.

[4] M. Flynn. *Very High Speed Computing Systems.* Proceedings of the IEEE, 1986.

[5] P. Haupt, D. Lohse. *Grundlagen und Anwendungen der Linearen Optimierung.* Giradet, Essen, 1975.

[6] A. Klindworth. *Entwurf, Simulation und Realisierung einer parallelen Rechnerarchitektur für ein revidiertes Simplex-Verfahren.* Diploma Thesis, Department of Computer Science, University of Hamburg, 1992

[7] N. Köckler. *Numerische Algorithmen in Softwaresystemen.* Teubner, 1990.

[8] U. W. Kulisch, W. L. Miranker. *Computer Arithmetic in Theorie and Practice.* Academic Press, New York, 1981.

[9] T. Lewis, H. El-Rewini. *Introduction to Parallel Computing.* Prentice Hall, London, 1992.

[10] H. Müller-Merbach. *Operations Research.* Vahlen, München, 1973.

[11] European Silicon Structures Ltd. *SOLO 1400 Release 3.1 Reference Manual.* Berkshire, 1991.

[12] European Silicon Structures Ltd. *SOLO 2030 Release 2.1 Reference Manual.* Berkshire, 1990.

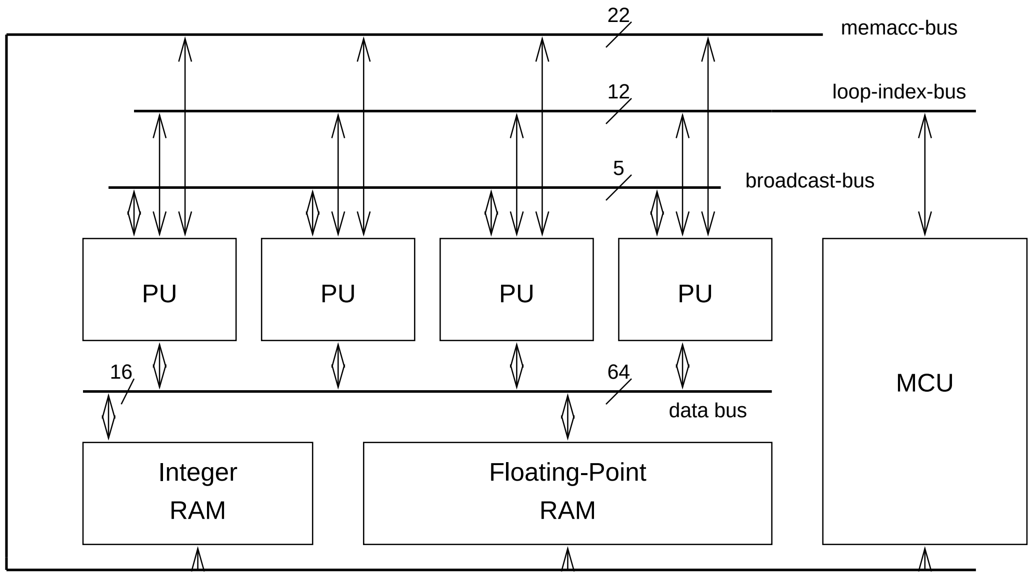

Figure 3: Overview of the accelerator architecture

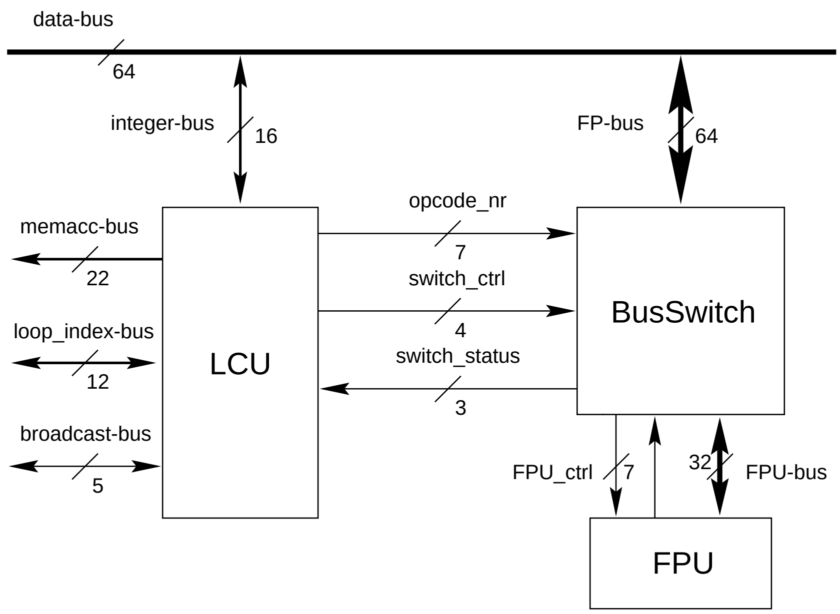

Figure 4: Architecture of one processing unit (PU)