# High Performance Numerical Calculation in Prolog Execution

*Robert Yung* †‡, *Alvin M. Despain* †, *Peter Van Roy* †, *Bruce K. Holmer* †

† Computer Science Division, University of California, Berkeley, CA 94720
‡ Xenologic Inc., 39899 Balentine Dr. Suite 145, Newark, CA 94560

4 February 88

## ABSTRACT

Numerically intensive calculations are not well supported by Prolog, yet there are important applications that require tightly coupled symbolic and numeric calculations. We identify some additional built-in predicates and macros for Prolog to support numeric calculations. These predicates are implemented in several layers of software and hardware, including a specially designed high performance numeric coprocessor. Simulated performance results indicate the system will achieve about 4 MFLOPS on the Prolog version of some Whetstone benchmarks (in double precision).

## 1. Introduction

Contemporary Prolog execution systems provide excellent support for symbolic calculations, but are generally quite weak in their support of numeric and linear algebra calculations. Yet some of the most interesting and challenging applications of logic programming require high performance execution of tightly coupled symbolic and numeric calculations. Examples include computer-aided design/engineering/manufacturing, sensor fusion, robotics, constraint logic programming, geometric modeling and reasoning with probabilistic evidence.

In our Aquarius project [6], one of the main applications is design automation [3] and it requires extensive numeric calculations as well as symbolic manipulations. We are investigating additional built-in predicates and macros for the Prolog language to better support numeric operations. The predicates have a semantic interpretation in a kernel subset of Prolog, but can be efficiently and directly compiled into powerful machine instructions. At execution time, most of the machine instructions are executed by a symbolic processor, the PLM [8,9]. When the special numeric instructions are fetched by a pre-fetch unit, they are ignored by the symbolic processor and are acted upon by the Aquarius Numeric Processor (ANP) [15].

The ANP is a high performance vector numeric processor especially designed to support numeric operations that occur in the context of logic programming. Figure 1 shows a block diagram of this integrated ANP/PLM architecture. The ANP coprocessor of figure 1 is currently under construction using TTL and ECL parts and will be inserted into our current experimental system[1] in the near future.

We are struggling with the many conflicting issues that develop when all the complexities of logic programming, floating point calculations, and linear algebra interact with the problems of exceptions, side-effects, efficiency of execution, and 'beauty' of language expression. It is our desire not to further burden the semantics of Prolog with any additional non-logical complications, but at the same time we must provide for efficient numeric

[1]Our current experimental system is a Xenologic model X-1 [7] co-processor with a Sun 3/160 host. The X-1 is an improved, commercial version of the PLM.

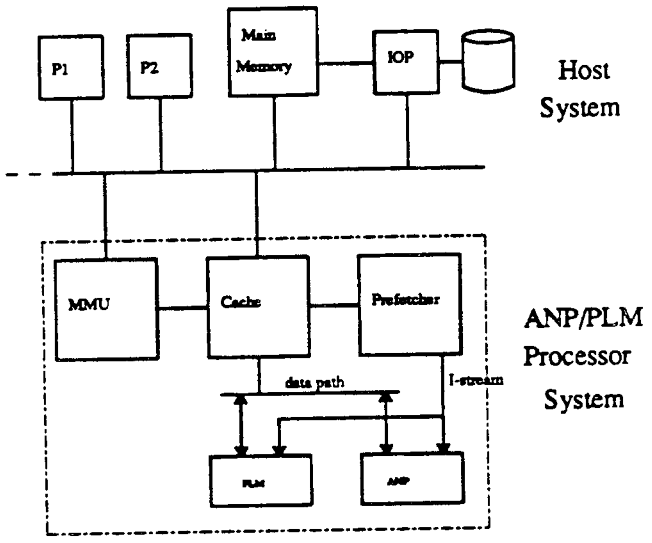

**Figure 1: ANP/PLM System Block Diagram**

calculations if the Aquarius system is to be useful for our applications. In the sections below we explain our current choices and compromises. We fully expect that our system will evolve as we discover and solve problems and gain experience in debugging and analyzing the new ANP/PLM System.

## 2. An Example of a Numeric Program

The numerically intensive calculations in science and engineering that are not well supported by Prolog include the heavy use of floating point, destructive assignment, arrays, and iteration (loops). A simple example from linear algebra (the solution of tridiagonal systems [12]) illustrates how we intend to address these points. The original Fortran code (slightly modified for illustration purposes) for this calculation is:

```
        SUBROUTINE TRIDAG(A, B, C, R, U, N)
        PARAMETER (NMAX=100)
        DIMENSION G(NMAX), A(N), B(N), C(N), R(N), U(N)
        H = 1
        IF (B(1) .EQ. 0.) PAUSE
        E = B(1)
        U(1) = R(1) / E
        DO 11 J = 2, N
                G(J) = C(J-1) / E
                E = B(J) - A(J) * G(J)
                IF (E .EQ. 0.) PAUSE
                U(J) = (R(J) - A(J) * U(J-1)) / E
11      CONTINUE
        DO 12 J = N-1, 1, -1
                U(J) = (U(J) - G(J+1) * U(J+1)) * H
12      CONTINUE
        RETURN
        END
```

When this code is directly translated into Prolog, one obtains the following executable code.

```
tridag(A, B, C, R, NNNU, N) :-
        H is 1,
        new_array(G),
        new_array(U),
        aref(1, B, E),
        aref(1, R, R1),
        test_bet(E),
        U1 is R1 / E,
        aset(1, U, U1, NU),
        xtridag(2, N, A, B, C, R, NU, G,
                           E, NNU, NG),

        N1 is N-1,
        ytridag(N1, NNU, NG, NNNU, H).

ytridag(J, U, _, U, _) :- J < 1.
ytridag(J, U, G, NNU, H) :- J >= 1,
        J1 is J+1,
        aref(J, U, UJ),
        aref(J1, U, UJ1),
        aref(J1, G, J1),
        NUJ is (UJ - J1 * UJ1) * H,
        aset(J, U, NUJ, NU),
        NJ is J-1,
        ytridag(NJ, NU, G, NNU, H).
```

```
test_bet(E) :-
        E =:= 0.0, !,
        write('E is zero in tridag'), nl,
        trace, fail.
test_bet(_).

xtridag(J, N, _, _, _, _, U, G,
                    _, _, U, G) :- J > N.
xtridag(J, N, A, B, C, R, U, G,
              E, H, NNU, NNG) :- J =< N,
        J1 is J-1,
        aref(J, A, AJ),
        aref(J, B, BJ),
        aref(J1, C, CJ1),
        aref(J, R, RJ),
        aref(J1, U, UJ1),
        GJ is CJ1 / E,
        aset(J, G, GJ, NG),
        NE is BJ - AJ * GJ,
        test_bet(NE),
        UJ is ((RJ - AJ * UJ1) / NE) * H,
        aset(J, U, UJ, NU),
        NJ is J+1,
        xtridag(NJ, N, A, B, C, R, NU,
                NG, NE, NNU, NNG).
```

The predicates `aref`, `aset`, and `new_array` are library routines for the support of extendible arrays [13]. In this implementation arrays are represented by balanced 4-way trees. Thus, the access time is logarithmic in the size of the array. Although this time may not seem large, an 1000 element array could require more than 20 words to be written for the modification of a single element. This overhead in both time and memory is not tolerable in high performance systems. For these reasons, we have introduced new predicates for destructive (yet backtrackable) assignment.

It is apparent that the readability of the Prolog code is much less than that of the Fortran code. Much of the clutter is caused by not being able to compute array elements directly in the assignment (`is`) statements. This problem is easily corrected by the use of a macro (called `*=`) which allows user defined functions to be placed in assignment statements. We use square brackets to denote array indexing. This does not conflict with list notation because they always follow a variable.

The Fortran `DO` loop must be replaced with recursion in Prolog. This in itself is fine, but often a large number of active variables must be passed to the recursive predicate. Lengthy argument lists introduce unnecessary tedium for the programmer and sources of error. The use of an iteration macro (called `do`) will alleviate this problem. The two macros `*=` and `do` are described in the next section. The Prolog code can be rewritten as follows.

```
tridag(A, B, C, R, U) :-
    H is 1,
    E[1] *= B[1],                  % boundary
    V[1] *= R[1] / B[1],
    do(J, 2, v_length(A),          % from 2 to length of A
      (G[J] *= C[J-1] / E[J-1],    % induction
      E[J] *= B[J] - A[J] * G[J],
      Y[J] *= (R[J] - A[J] * V[J-1]))),
    U[N] *= V[N],
    do(J, (N-1), 1, -1,            % from M to 1 step -1
      (U[J] *= (V[J] - G[J+1] * U[J+1]) * H)).
```

It can be seen that the use of the new notation restores the clarity of the algorithm.

## 3. Prolog Language Issues

We now discuss some important syntactic and semantic issues in providing a clean interface between the ANP hardware and the Prolog language. There are two general approaches to combine numeric computation with Prolog: 1) Include a second language which allows efficient implementation (but with procedural semantics) and an external language interface to Prolog. In this scheme the declarative semantics are lost for the system as a whole. 2) Extend Prolog to allow numeric computation. With care, the semantics of Prolog will be retained while allowing an efficient implementation. This is the approach we take in this design.

Our extension to the language is guided by three principles. First, it must allow an efficient implementation. Second, the logical semantics of Prolog should be kept. And third, it must be clean for the application programmer. Our approach has two facets: 1) Introduce new scalar and vector numeric types and operations which are then directly supported by the ANP, and extend the semantics of Prolog primitives for the new types. 2) Introduce a simple but powerful macro facility and a data typing scheme to allow concise scientific programming. We describe a set of suggested macros and built-in predicates which allow this while retaining the logical semantics.

### 3.1. The Macro Facility

Our macro facility is similar to that of SB-Prolog [2]. The scheme used in ESP [4] was rejected because of its complexity. The SB-Prolog macro facility will expand a goal inline and partially evaluate it. This simple idea is quite powerful. It will suffice to implement our ideas if the partial evaluator is general enough, and if `assert` is given the proper interpretation.

#### 3.1.1. Assignment and User-defined Functions

In order to denote array assignment and user-defined functions in a concise manner we introduce the `:=` and `*=` macros. These expand array references into explicit calls of new built-ins which access the array elements. They also expand function calls into goals with an additional argument that will contain the function value. The `*=` macro unifies its arguments (thus keeping the logical semantics), while the `:=` macro destructively assigns (with restoration of the value on backtracking). For example, the macro call `A[3]:=f(X)` will be expanded into the two goals `f(X,T), rplacarg(3,A,T)`. Part of the definition of `:=` is:

```
(L := R) :- right_eval(L, X), left_eval(R, X).

right_eval(X, X) :- (number(X); var(X)), !.
right_eval(Aref, X) :- array_ref(Aref), !,
       Aref =.. [Aname|Args],
       eval_index(Args, Index),
       aref(Aname, Index, X).
                                        % similar clauses for other operations
right_eval(A+B, X) :-
       right_eval(A, VA),
       right_eval(B, VB),
       X is VA+VB.
right_eval(Func, X) :- function_call(Func), !,
       Func =.. [Fname|Args],
       append(Args, [X], AllArgs),
       Call =.. [Fname|AllArgs],
       call(Call).
```

The `*=` macro is similarly defined.

### 3.1.2. Denoting Iteration

We suggest a `do` macro to denote iteration in a clean way similar to a Fortran `DO` loop:

```
    do(Index_name, Start_index, End_index, Optional_increment, (Body))
```

All objects mentioned in the body of the `do` are global in the scope of the clause in which the `do` construct appears. This macro allows the use of destructive assignment in the body. However, if all assignments are done with `*=` then the logical semantics are kept. The macro expands into a recursive predicate of the following form which is put in the global data base:

```
    x_do(Index_name, End, ...BodyVars... ) :- Index_name>End.
    x_do(Index_name, End, ...BodyVars... ) :- Index_name=<End,
         (Body),
         New_index is Index_name+Optional_increment,
         x_do(New_index, End, ...BodyVars... ).
```

The inline code is a sequence of goals initializing the body's variables (as denoted by `BodyVars`), followed by the call `x_do(Start_index, End_index, ...BodyVars...)`.

## 3.2. Backtracking Semantics

Ideally, the semantics of the numeric operations would fit into pure Prolog, with single assignment vectors and full state restoration on backtracking. This can be achieved sometimes, for example in the `do` predicate as used in the tridag example. We do not presently see how it can be achieved in general while keeping the highest performance. As a compromise, we will present a design which achieves efficiency with no greater harm to the logical semantics than the `var` predicate.

Prolog together with a backtrackable destructive assignment (which we call `rplacarg`) is no less logical than Prolog with `var` because `rplacarg` can be implemented with `var` (albeit inefficiently) [11]. When `rplacarg` is implemented directly in the underlying architecture it can execute in constant time. We conjecture that Prolog with this implementation of `rplacarg` can achieve the same time bound as a procedural language on any problem.

A block of floating point operations is implemented by loading the ANP registers (see below) from the heap, doing the calculations, and finally storing the results. Trailing of the ANP registers is never done; only the heap is trailed.

As a result of the above reasoning, we require that vectors must be restored on backtracking just like other Prolog terms. Destructive assignment is allowed as long as the old value can be restored. There are two methods to achieve this. The first way is to trail all floating point stores to the heap. Note that loads and numeric operations do not need to be trailed. The second way is to trail before the first assignment after choice point creations, and then trail only those vectors which will be changed. The choice of which of these methods to use is up to the compiler. For efficiency it will attempt to keep all trail checking out of the inner loops. One possible optimization is to recognize that if multiple assignments are done between choice point creations then only the first needs to be trailed.

## 3.3. Exception Handling

Floating point exceptions are handled by means of failure. The existing failure mechanism in Prolog is already set up to handle state restoration and continued execution. In order to use this we propose the addition of two global facts which are always accessible to the program: `exception_enable(ExceptionList)`, `exception_occurrence(Exception)` where `exception_enable` is given a list of flags telling what exception condition(s) will cause failure, and `exception_occurrence` will unify with the last exception which has actually caused failure. The programmer is not obliged to use these two facts as long as he realizes what the cause of a failure is. We implement and support the standard IEEE exceptions and proposed handling scheme [1,15] and some ANP-specific exceptions (such as bounds violation) in this design. However, we do not support user-defined exceptions.

As an example of the use of these predicates, consider the following numeric code which can fail both through an exception and through design (i.e. choosing an alternate algorithm if the first one is inadequate):

```
routine(...) :- algorithm1, !.                                % 1st algorithm
routine(...) :- exception_occurrence(none), algorithm2, !.    % 2nd algorithm
routine(...) :- exception_occurrence(E), not(E=none), exception_handler.
```

The nesting of exception handlers is provided naturally through the failure mechanism. A failure caused by an exception will restore the vectors at the most recent choice point, and go to the next clause, which could contain an exception handler. Consider this example:

```
Sequence of goals:    ..., algo(...), ...
                                 ^execution continues here
algo(...) :- code1...<exception occurs here>...code2.
algo(...) :- exception_handler.
```

If the handler succeeds then execution continues at the deepest goal containing the exception which has created a choice point, in this case `algo`. It does not continue at the exact point of occurrence of the exception. If the handler is not able to continue then it also will fail and execution will continue at the next higher handler in the hierarchy. In the example this happens when the call `algo(...)` fails.

The addition of two global facts which change during execution harms the logical semantics. We feel quite strongly that this should be rectified but we have not yet been able to invent a satisfactory solution that is sufficiently efficient. Thus we merely make visible the hardware exception registers to the Prolog programmer [10, 14].

## 4. Machine Programming Model

Because the Aquarius Numeric Processor (ANP) is a coprocessor to the Programmed Logic Machine (PLM), it inherits the data types and programming model from the PLM [8, 9]. It adds new data types to the programming model including, in both scalar and vector forms, integer, single and double precision floating point numbers in IEEE standard (754) form [1]. An extended numeric register set and a large repertoire of integer and floating point operations are provided for these new data types.

### 4.1. Representation of the Numeric Data Types

Data in the PLM programming model is represented by 32-bit tagged words. There are four primary types: list, structure, variable and constant, which are distinguished by bit<31:30>. These are shown in figure 2. Bit<29> is a cdr bit which is used for compact list representation, and bit<28> is a garbage collection bit. This bit is reserved for data marking during garbage collection. Bit<27:26> of a constant data type further differentiate between a 26-bit small integer (00), other-numeric header (01), an atom (10) and a nil (11). This tagging information allows efficient manipulation of data by applying different strategies to operate on each class of data. Although data typing benefits from efficient execution, it decreases the amount of information that can be stored within each data word.

Several new data types are added to the ANP for numeric computations. The fundamental numeric data types are 32- and 64-bit integer and single and double precision floating point numbers. Arrays based on these fundamental data types can be constructed in single and multi-dimensional forms. Integers and floating point numbers for computation in the ANP conforms to the IEEE Standard P754 [1].

#### 4.1.1. Structure Numeric Representation

The IEEE Standard for binary floating-point specifies numeric operands to be a multiple of a 32-bit word except for the recommended extended format, which is 80-bits long. To maintain compatibility with this standard as well as the PLM execution model, an additional 32-bit word is needed to store data type information. The Structure Numeric Representation (SNR), figure 2) utilizes a structure pointer to the numeric operand it is representing. The structure pointer has a 28-bit address pointing to the location of the numeric operand on the heap. The first entry of the numeric operand is a header which has a constant primary tag, garbage collection and cdr bits, an other-numeric secondary tag (bit<28:27> = 01), four bits of numeric tags and a 16-bit vector length. The numeric tags specify the extended data types which include vector/scalar (V), double-/single-precision (D), floating-point/integer (F) and unsigned/signed (S) of the operand. Tag space is also provided for additional numeric types such as infinite precision integers, (multi-words) bit vectors, decimal, and complex numbers that may be added in the future. Since the PLM data path cannot directly operate on the 32-bit numeric operands, the entire numeric structure addressed by an indirect pointer will not be transferred into the PLM register set, but into the ANP instead.

This encoding scheme is compatible with IEEE standard at the expense of less efficient execution and more memory storage for numeric operands.

## 4.2. Dynamic Operand Coercion

Many numeric operations generally appear in the instruction sets of scientific processors. Often a subset of equivalent scalar opcodes appear in vectorized forms as well. Normally the programmer (or the compiler) chooses the correct opcodes for the data types used in each program. For general programs, code for testing the input data types must be added to accommodate the dynamic nature of the input. There are two undesired side-effects in this method: 1) The extra code increases the size of the program, thus increasing the demand on a generally critical system resource, input/output to main memory. 2) The added test and branch opcodes decrease the efficiency in the processor's (pre-)fetching mechanism. The second side-effect is greatly magnified in a vector processing system in which the functional units are pipelined. We thus choose to support Dynamic Operand Coercion (DOC) [15]. Programmers can describe the numeric operations that are required to accomplish a goal without consideration of the input data types involved. The ANP will do dynamic type checking and coerce the arguments if necessary. The implementation is such that there is no overhead when no coercion is done (i.e. if the types are identical).

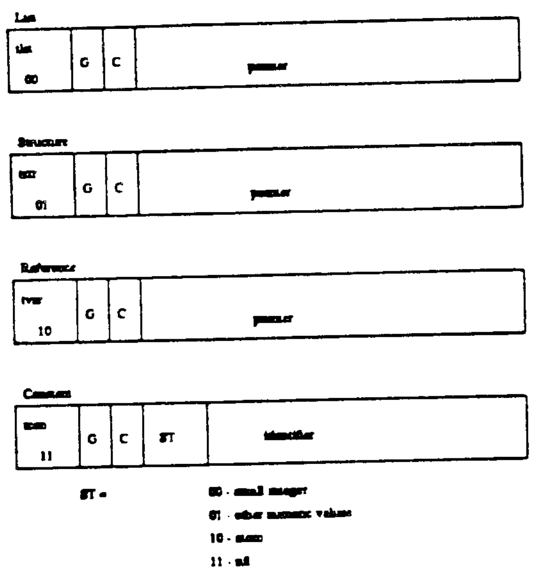

Figure 2: PLM Data tagging representation

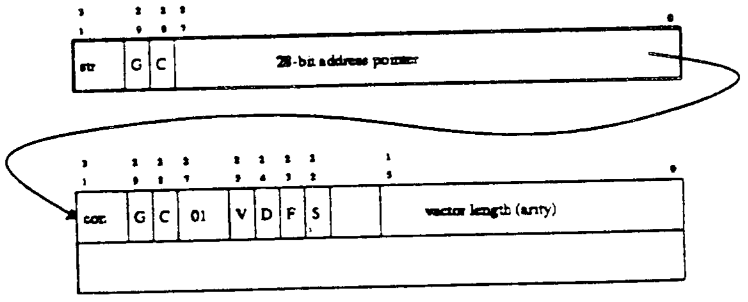

Figure 3: Structure Numeric Representation

## 4.3. Extended Numeric Register Set

The architecture of the ANP adds a number of data and state registers to the PLM programming model. There are an additional eight general purpose data registers that can be configured for scalar ($F_0$ - $F_7$) or vector ($B_0$ - $B_7$) storage. In addition to these eight data registers, there are 40 scalar registers ($F_8$ - $F_{47}$), 16 scratch pad registers reserved for internal use ($F_{48}$ - $F_{63}$), 128 predefined constants, and two control and status registers for system control. Each data register can store a 32-bit integer or a single or double precision floating point number. The data type and vector length of each vector is stored in the corresponding 32-bit header register ($H_x$). Figure 4 shows the combined ANP/PLM register set.

There are two registers in the ANP for status and control communication between the ANP, the PLM and the memory unit. System parameters can be written to the control register (CR) for initialization of the ANP. Status and flags can be read from the status register (SR) for debugging.

## 4.4. The ANP Instruction Set

Data movement instructions provide a means to load or store programmer visible registers in the ANP. Address calculation for these instructions is done in the PLM. The PLM fetches the numeric data structure from the heap addressed by an $A_x$ register or writes data provided by the ANP to the top of the heap.

FMOVxx instructions move data between an element of a $B_x$ and a $F_x$ register. This allows efficient access to individual elements of an array. FMOVL/FMOVSF uses an 8-bit immediate index for $B_x$ access; FMOVLF/FMOVSF uses the modulo 256 of a $F_y$ register value as an index for $B_x$ access.

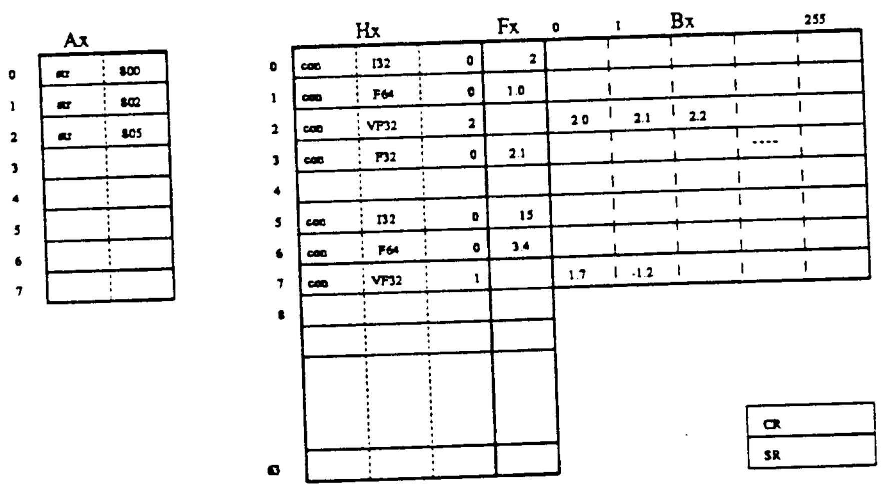

Figure 4: Extended PLM/ANP register set

Table 2: The ANP Instruction Set

| Data Movement | Arithmetic | Logical | Bit | Conversion |
|---|---|---|---|---|
| FLOAD | ADD | NAND | LS | USP |
| FLOAD_CR | ADDA | ANDNX | AS | UDP |
| FSTORE | SUB | ANDNY | ROT | ISP |
| FSTORE_SR | SUBA | AND | | IDP |
| FMOVL_INDEX | SUBX | ORNX | | SPU |
| FMOVLF_INDEX | SUBXA | ORNY | | SPI |
| FMOVS_INDEX | MULT | OR | | SPDP |
| FMOVSF_INDEX | MULTA | NOR | | DPU |
| FMOVE | DIV | XNOR | | DPI |
| FLOAD_MINDEX | XOR | | DPSP | |
| FLOAD_IMINDEX | NOTX | | | |
| FSTORE_MINDEX | NOTY | | | |
| FSTORE_IMINDEX | PASSX | | | |
| | | PASSY | | |
| | | SET | | |
| | | CLR | | |

| Monadic | Compare | Compound | Misc. |
|---|---|---|---|
| ABS | CMP | SQRT | CLF |
| NEG | MAX | MAC | NOP |
| | MIN | SMAC | |
| | | MACS | |

ABSolute and NEGate operations can be applied to all numeric data types in scalar and vector forms. The dyadic instructions include several classes of numeric functions. Arithmetic functions such as add, subtract, multiply and divide are supported in both normal form and absolute forms (e.g. Y is |A + B|). A full set of logical and bit manipulation operations are included for signed and unsigned integer data types. Shift and rotate instructions accept a shift count as an immediate value or from a numeric register.

Comparison instructions are used to test conditions for branching instructions and to select the maximum or minimum value from a set of numbers. Compound instructions are microcoded sequences of the basic operations. For example, MAC/SMAC/MACS calculates the inner product of two vectors. Conversion instructions provide a means to change between data formats.

## 5. ANP Architecture

The purpose of the ANP is to supplement the PLM symbolic processor with high performance numeric operations while maintaining upward compatibility with the existing PLM's Instruction Set Architecture. This is accomplished with an extension of numeric data types and instructions, as described in the previous sections, and an architecture that efficiently supports these new extensions. The ANP functions as a slave coprocessor to the PLM. The programmer perceives the PLM/ANP execution model as if all numeric instructions are executed in the PLM. In systems where an ANP is not present, numeric operations are emulated in software via traps to the host processor.

A Private Memory Bus (PMB) connects the PLM to its memory system. The ANP utilizes the PMB to provide a logical extension of the PLM registers and instructions in a manner which is transparent to the programmer. The ANP consists of five independent functional units operating concurrently to achieve high performance in numeric computations [14, 15]. The **Bus Interface Unit** (BIU) is responsible for all communications between the

ANP, PLM and memory system. The **Operand Coercion Unit** (OCU) provides operand type checking, coercion (DOC) and vector length management for the Execution Unit. The **Storage Unit** (SU) consists of 64 header registers, 64 scalar registers, eight 256-element vector registers, and 128 predefined constants. The **Execution Unit** (EU) contains the data path for integer and floating point operations. The heart of the ANP is a **Micro Control Unit** (MCU) which consists of a microprogram sequencer, a 96-bit horizontal writable control store, and other circuitry that handles exception processing and initialization of microcode. A block diagram of the ANP is shown in figure 5.

## 6. Performance Measurements

Evaluation of the ANP is done in two steps. First, a register transfer level simulator provides a means for the evaluation of the microarchitecture of the ANP. Second, a hardware implementation will be constructed and tested with calculations that are too large to be simulated. Preliminary performance measurements were obtained from simulation of the design using a set of benchmark programs written in Prolog.

### 6.1. Measurement Results

A set of Prolog programs translated from selected (double precision) Whetstone benchmark modules [5] is used to verify the correctness and measure the performance of the PLM/ANP system. The second of the Whetstone programs 'wh2' is shown below to illustrate the style of some selected Whetstone benchmark modules written in Prolog with our new array notations.

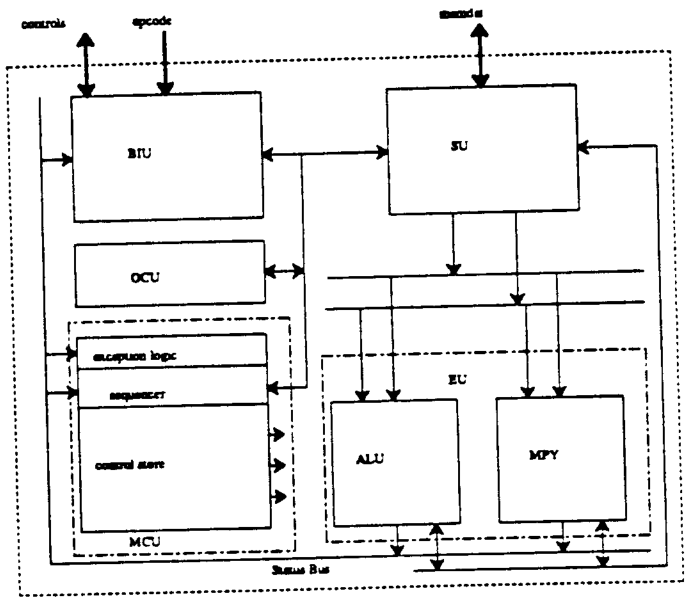

Figure 5: ANP Simplified Block Diagram

```
wh2(0,A,_,A)  :- !.
wh2(N,E,T,Y)  :-
    A[0] =: ( E[0]  + E[1]  + E[2]  - E[3])  * T,
    A[1] =: ( A[0]  + E[1]  - E[2]  + E[3])  * T,
    A[2] =: ( A[0]  - A[1]  + E[2]  + E[3])  * T,
    A[3] =: (-A[0]  - A[1]  + A[2]  + E[3])  * T,
    M is N - 1,
    wh2(M,A,T,Y).
```

Table 3 shows the measurements obtained from simulation of the ANP architecture. The second column shows the number of floating point operations (flop) in one iteration of the corresponding benchmark. Columns three to six show the variation in mega-flops (MFLOPS) when each benchmark is run for one hundred, one thousand, ten thousand and one hundred thousand iterations.

Table 3: Simulated Benchmark Performance (units in MFLOPS)

| | | iterations (double precision calculations) | | | | |
|---|---|---|---|---|---|---|
| test | flop | 100 | 1K | 10K | 100K | comments |
| wh1 | 16 | 4.36 | 4.55 | 4.57 | 4.57 | simple identifier |
| wh2 | 16 | 4.42 | 4.56 | 4.57 | 4.57 | array element |
| wh3 | 102 | 3.42 | 3.43 | 3.43 | 3.43 | array as parameter |
| wh6 | 15 | 3.38 | 3.48 | 3.49 | 3.49 | integer arithmetic |
| wh8 | 7 | 2.32 | 2.40 | 2.41 | 2.41 | procedure call |
| wh10 | 5 | 3.64 | 3.83 | 3.84 | 3.85 | integer arithmetic |
| mac | 511 | 16.08 | 18.24 | 18.25 | 18.25 | inner product |

## 7. Conclusions

Our preliminary results are encouraging. The language constructs we developed allow clean, compact and easy to understand numeric programs. They also have semantics within kernel Prolog and efficient mappings into the specialized hardware of the ANP/PLM system. Simulation results indicate a performance of 4 MFLOPS (in double precision) on selected modules of the Whetstone benchmark written in Prolog. Thus numeric performance is reasonably well matched with the PLM symbolic performance.

**Acknowledgement**

The Aquarius Project, of which this study is a part, is a group effort. The authors wish to express their gratitude to the other members of the group. In particular Yale Patt, Tep Dobry, Vason Srini, Mike Carlton and Jerric Tam. We also acknowledge the stimulating interactions with William Kahan and Paul Hilfinger of UCB, Robert Owen of BIT Inc., Lynn Ackler, Chris Brooks and Barton Sano of Xenologic Inc. and John Yung of SUN Microsystems for his help in simulating the microarchitecture. We also thank Valid Logic Corporation and MMI for their generous contributions of equipment and parts and Doug McCafferty for his help in obtaining the CAD software for schematic capture and circuit simulation. This work is partially sponsored by the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871 and monitored by Space and Naval Warfare System Command under contract No. N00039-84-C-0089, and by the California MICRO Program.

## References

1. *A Proposed Standard for Binary Floating-Point Arithmetic - Task P754*, Microprocessor Standards Committee, IEEE Computer Society (1981).
2. *The SB-Prolog System, Version 2.2, A User Manual*, Dept. of CS, University of Arizona (Mar 1987).
3. Bush, W. R. et al., "An Advanced Silicon Compiler In Prolog," *Conf. Proc. of the ICCD - 1987*, (1987).
4. Chikayama, T., *Unique Features of ESP*, ICOT Technical Report TM-0055 (Apr 1984).
5. Curnow, H. J. and Wichman, B. A., "A Synthetic Benchmark," *Computer Journal* 19(1)(Feb 1976).
6. Despain, A. M. et al., "Aquarius," *Computer Architecture News*, (Mar 1987).
7. Despain, A. M., "A High Performance Prolog Co-Processor," *Digest of Papers, WESCON 1985*, IEEE Press, (1985).
8. Dobry, T. P., Despain, A. M., and Patt, Y. N., "Performance Studies of a Prolog Machine Architecture," *Conf. Proc. of the 12th Annual ISCA*, (Jun 1985).
9. Dobry, T. P., "A High Performance Architecture For Prolog," PhD Dissertation, Computer Science Division, University of California, Berkeley CA (May 1987).
10. Kahan, W. M., "Handling Arithmetic Exceptions," unpublished notes, Computer Science Division, University of California, Berkeley CA (1987).
11. McGeer, R. et al., "Prolog for VLSI Layout: Experiences in the Design and Implementation of Topolog, A Prolog Based Module Generation and Layout System," *Report No. UCB/CSD 87/363*, Computer Science Division, University of California, (July 1987).
12. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes - The Art of Scientific Computing*. 1986.
13. Warren, D. H. D. and , F. Pereira, "Extendible arrays with logarithmic access time," *Quintus Prolog Library*, (1985).
14. Yung, Robert and Despain, Alvin M., "Aquarius Numeric Processor," *Submitted to the 13th annual ISCA*, (May 1988).
15. Yung, Robert, "Aquarius Numeric Processor," *Masters Thesis*, Computer Science Division, University of California, (In preparation).