An Extended Prolog Architecture for Integrated Symbolic and Numeric Executions

Robert Yung
Alvin M. Despain
Yale N. Patt

Dept. of Electrical Engineering & Computer Science
University of California, Berkeley
Berkeley, CA 94720
(415) 642-5615
yung@ji.Berkeley.EDU
despain@ji.Berkeley.EDU
patt@ji.Berkeley.EDU

14 June 88

ABSTRACT

Numerically intensive calculations are not well supported by Prolog, yet there are important applications that require tightly coupled symbolic and numeric calculations. The Aquarius Numeric Processor (ANP) is an extended numeric Instruction Set Architecture (ISA) based on the Berkeley Programmed Logic Machine (PLM) to support integrated symbolic and numeric calculations. This extension expands the existing numeric data type to include 32- and 64-bit integers, single and double precision floating-point numbers conforming to the IEEE Standard P754. A new class of data structure, numeric arrays, is added to represent matrices and arrays found in most scientific programming languages. Powerful numeric instructions are included to manipulate the new data types. Dynamic Operand Coercion (DOC) provides dynamic type checking and coercing of operands. The ANP and PLM together provide for the efficient execution of symbolic and numeric operations written in AI languages such as Prolog and Lisp. Simulated performance results indicate the system will achieve about 10 MFLOPs on the Prolog version of some Whetstone and Linpack benchmarks and close to 20 MFLOPS on some matrix operations (all in double precision).

Keywords and phrases:
Dynamic Operand Coercion, Prolog, horizontal writable control store, integrated symbolic and numeric executions, multi-stage dynamic pipeline, scalar and vector processing.

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 linear algebra, digital signal processing, computer-aided design, engineering and manufacturing, design automation [3], robotics, Constraint Logic Programming [8, 9, 11], geometric modeling and reasoning with probabilistic evidence.

In our Aquarius project [4], 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 [7]. 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) [18].

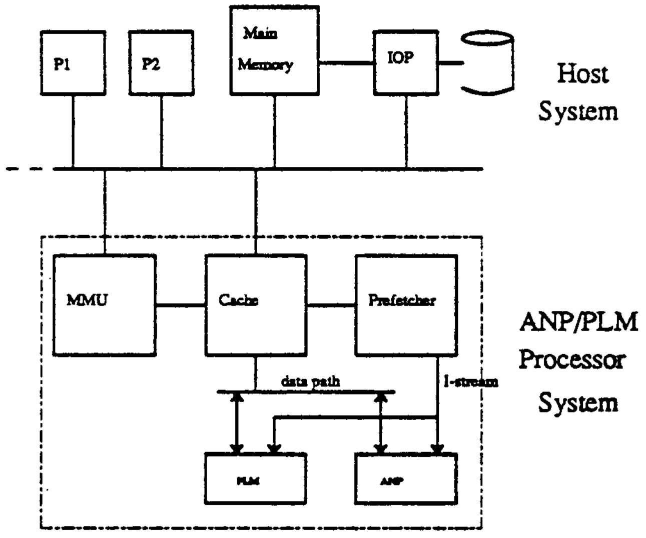


Figure 1: ANP/PLM System Block Diagram

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 co-processor 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.

2. Programming Model

Because the Aquarius Numeric Processor (ANP) is a co-processor to the Programmed Logic Machine (PLM), it inherits the data types and programming model from the PLM [7]. 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 [2]. An extended numeric register set and a large repertoire of integer and floating point operations are provided for these new data types.

2.1. The PLM programming model

The programming model of the PLM includes an execution model based on a modified and extended Warren Abstract Machine [6, 16]. The execution model of the PLM consists of four distinct memory areas and a number of registers associated with them. The **Code Space** contains compiled Prolog procedures and clauses. The **Data**

[1]Our current experimental system is a Xenologic model X-1 [5] co-processor with a Sun 3/160 host. The X-1 is an improved, commercial version of the PLM.

Space defines the dynamic state of the PLM and is divided into three areas each organized as a LIFO[2] memory. The **Stack** contains control informations defining the environments and choice points created during the execution of PLM instructions. The **Heap** is a general area for the storage of data. Finally, the **Trail** keeps track of variable bindings which must be unbound during backtracking. In addition to the state registers described above, there are eight data registers, A_0 - A_7, in the PLM for parameter passing and for the storage of temporary and frequently used data items. Following is a summary of registers that make up the machine state of the PLM.

P	- Program counter	**HB**	- Heap Backtrack pointer
CP	- Continuation Pointer	**S**	- Structure pointer
E	- Environment pointer	**TR**	- Trail pointer
B	- Backtrack pointer	**Status**	- mode, conditional code, ...
H	- Heap pointer	**Ax(0) - Ax(7)**	- Argument registers

2.2. 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 integers, 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 [2].

2.3. 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 (tagging) information. The Structure Numeric Representation utilizes a place holder as an indirect pointer to access the numeric operands. The place holder is modified with a structure primary tag, garbage collection and cdr bits, and a 28-bit address pointer pointing to the numeric data structure. Figure 3 shows this numeric data structure which consists of a place holder, a numeric header and the numeric operand it is representing. The first entry of the numeric operand is a numeric header which has the same top 6-bit tags of the place holder, a 4-bit numeric tag and a 16-bit vector

[2]LIFO is 'Last In, First Out' or stack managed memory.

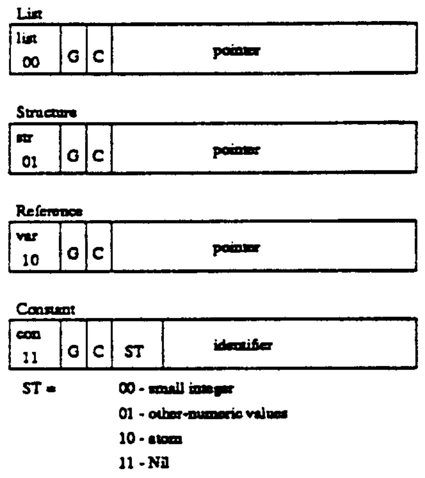


Figure 2: PLM Data Tagging Representation

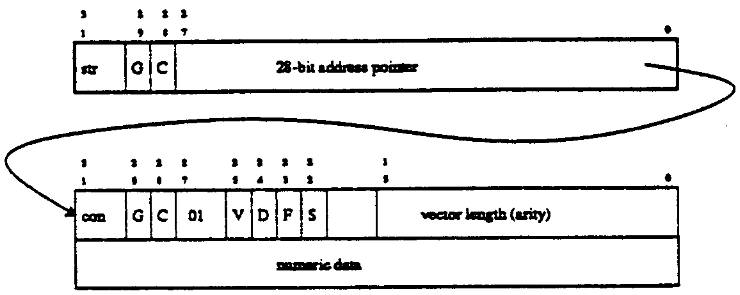


Figure 3: Structure Numeric Representation

length[3]. Bit<27:26 = 01> identify this header to be a numeric functor and the numeric tags specify the extended data types including 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 'Bignum', decimal, and complex numbers. Since the PLM data path cannot directly operate on 32-bit numeric operands, entire numeric structure addressed by an indirect pointer will not be transferred into the PLM register set, but into the ANP instead.

2.4. Dynamic Operand Coercion

Many numeric operations generally appear in the instruction set 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. This is fine for some applications that have static data types. For others, any change in the input data type often requires a recompilation effort. For general programs, code for testing the input data types used 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 increases the demand on a generally critical system resource, input/output to memory storage. 2) the added test and branch opcodes decrease the efficiency in the processors' (pre-)fetching mechanism.

The second side-effect is greatly magnified in a vector processing system in which the functional units are pipelined. Thus, the Dynamic Operand Coercion scheme (DOC) is supported in the ANP.

Dynamic Operand Coercion (DOC) is a mechanism built into the ANP architecture to support dynamic data type checking and operand coercion. The programmer can describe the numeric operations that are required to

[3] In effect, the vector length is just the arity of the structure.

accomplish a goal without any consideration of the input data types involved. The ANP will do dynamic type checking and coerce the operands if necessary. The implementation is such that there is no overhead when no coercion is done (i.e. if the types are identical).

2.5. Extended Numeric Register Set

The architecture of the ANP extends the PLM programming model with a number of data and state registers. There are eight general purpose data registers, F_0 - F_7, that can be configured for scalar (Fx) or (256-element Bx) vector storage. Vector instructions can directly access corresponding elements across vector registers during computation. In additional to the eight general purpose data registers, there are 56 scalar registers,[4] 128 predefined or user-defined constants, and two control and status registers. Each (element of a) data registers can store a 32-bit integer, a single or a double precision floating point number. The data type and vector length of each data register are stored in the corresponding 32-bit header register (Hx). There are two registers in the ANP for status and control communication between the ANP, the PLM and the memory unit.

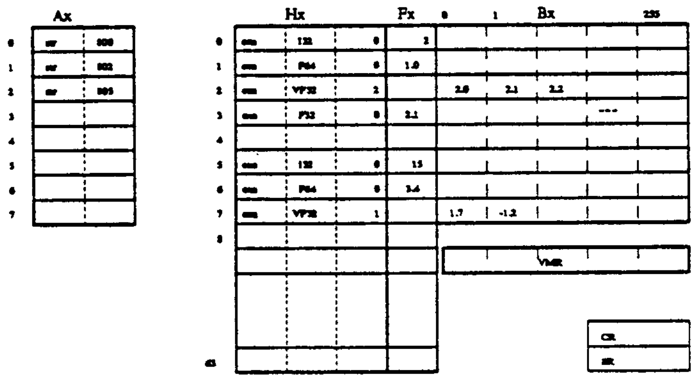


Figure 4: Extended PLM/ANP register set

2.6. The ANP Instruction Set

Data movement instructions provide a mean to load and store programmer visible registers in the ANP. These instructions set up operands for data manipulations and return results to memory. Address calculation for these instructions is done in the PLM. **Floadi** transfers an element from a vector register into a scalar register. **Fstorei** stores a scalar in into an element of the vector register.

ABSolute and NEGate instructions can be applied to all numeric data types in scalar and vector forms. Dyadic instructions perform additions, subtractions, multiplications and divisions. Mismatched data types are handled by the Dynamic Operand Coercion mechanism in the underlying architecture. Instructions with a prefix 'a'

[4]Top 16 scalar registers are reserved as scratch-pads, leaving only 48 programmer usable registers.

(i.e. **suba**) carry out an additional operation, the absolute values of the result are returned instead. 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 from a scalar register.

Comparison and test instructions are used to test and set condition code for branching. Conversion instructions provide a means to change between data formats. Compound instruction is a collection of monadic and dyadic operations in a single instruction. **Mac** and **Smac** are sum-of-product and difference-of-product operations for vector operands. **Incgt** and **Decle** are loop control instructions. **Set_v_countX** sets up the index of the operand for vector operations.

Data Movement	Arithmetic	Logical		Conversion	
Fload	**Add**	**Nand**	**Xnor**	**Spu**	**Usp**
Fstore	**Adda**	**Andnx**	**Xor**	**Spi**	**Udp**
Fload_CR	**Sub**	**Andny**	**Notx**	**Spdp**	**Isp**
Fstore_SR	**Suba**	**And**	**Noty**	**Dpu**	**Idp**
Floadi	**Subx**	**Ornx**	**Passx**	**Dpi**	
Fstorei	**Subxa**	**Orny**	**Passy**	**Dpsp**	
Fmove	**Mult**	**Or**	**Set**		
	Multa	**Nor**	**Clr**		
	Div				
Monadic	Bit	Compare	Test	Compound	Misc.
Abs	**Lshift**	**Cmp_XX**	**Tests**	**Sqrt**	**Clf**
Neg	**Ashift**	**Max**	**Testr**	**Mac**	**Nop**
	Rotate	**Min**		**Smac**	**Incgt**
					Decle
					Set_v_count1
					Set_v_count2
					Set_v_countR

Table 1: Summary of the ANP Instruction Set

3. The ANP Architecture and an Implementation

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 co-processor to the PLM. The programmer perceives the ANP/PLM 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.

3.1. The Execution Pipeline

One of the bottom-neck in the ANP/PLM system lies in the memory bandwidth of the PMB. Although the PMB has separate Opcode and MAR address busses, the Memdat bus is multiplexed between code and data. Furthermore, the memory access cycle time on the PMB is one Pclk cycle which is equal to two Fclk cycles (50ns each).

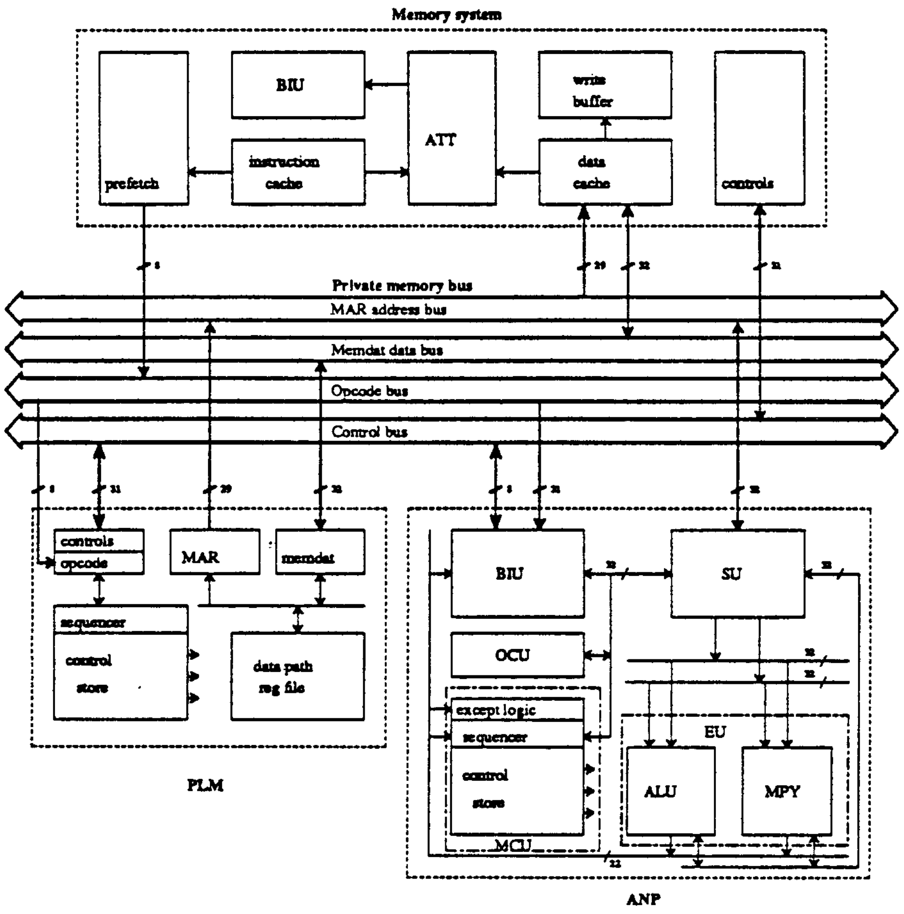


Figure 4: The ANP System Block Diagram

Like in most high performance vector computer design [13], pipelining is employed to increase the throughput of the system by overlapping execution of ANP instructions. This is accomplished with a two-word deep instruction buffer and a dynamic execution pipeline. The ANP execution pipeline can issue a register-to-register instruction every two fclk cycles for most scalar operations. For vector and other scalar operations the execution pipeline is dynamically *stretched* to accommodate multi-cycles instructions. Maximum performance is obtained for scalar operations while no penalty is incurred on others.

3.1.1. Opcode Formats

All ANP instructions consist of an 8-bit opcode and a 32-bit argument (operand)[5]. The opcode differentiates one of three ANP instruction classes: **Load**, **Execute** or **Store**. The first and third instruction classes include all the

[5]Similar to a size four PLM instruction.

transfer operations between the ANP, PLM and the memory system. The Execute class consists of all register-to-register operations in the ANP.

For Load and Store instructions, bytes zero and two of the argument contains the Ax register where the operand is transferred from or into the ANP Fx register respectively. Byte one is an index for element extraction from or insertion into a vector register. Most Execute instructions utilize all four bytes of the argument. Byte three contains the **Fopcode** which specifies the execution opcode for the ANP calculation. Bytes zero and one specify the two source operands of a dyadic operation. Byte two is the destination operand where a calculation is stored.

3.2. The Instruction Buffer - Fetch/Decode Stages

The purpose of the instruction buffer (**IBuffer**) is to overlap the execution of the current instruction with pre-fetching and decoding of the next one. The IBuffer consists of two latches, **Darg** and **Earg**, which together make up the first two stages (**Fetch** and **Decode**) of the system pipeline. The pipeline flow diagram is shown in figure 5.

When an ANP opcode is pre-fetched by the PLM, the eight-bit opcode as well as the 32-bit argument (operand) are transferred into the **Darg** latch. The 8-bit Fopcode is decoded by the OCU. If no coercion (internal exception) is needed, the instruction is transferred into the **Earg** latch for execution[6]. Register reads, execution, and storage of results take place after the ANP instruction is transferred into Earg latch.

3.2.1. Read/Execute/Write/Store Stages

After decoding in the **Decode** stage of the pipeline, an ANP Execute instruction is transferred into the Earg latch for execution. The third **Read** stage of the system pipeline transfers the operand(s) from the ANP register file (SU) into the EU. Two 64-bit reads are accomplished in a Fclk cycle. The register file addresses are derived from the lower three bytes of the Earg latch.

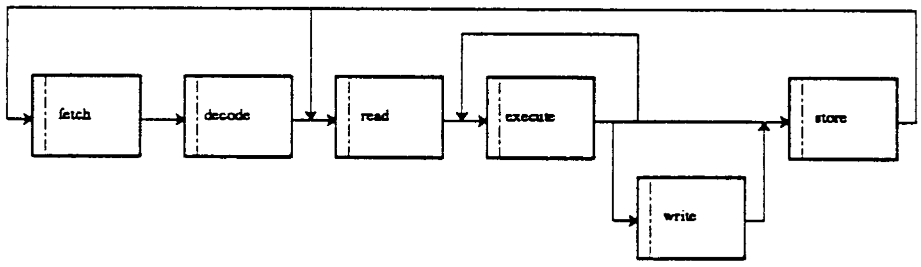


Figure 5: Pipeline Data Flow Diagram

[6]Transfer into the Earg latch may be delayed if the current instruction is a multi-cycle operation.

Following the operand access is the fourth **Execute** stage of the pipeline. Operand(s) are operated upon in the execution data path (EU) of the ANP. Most operations take one Fclk cycle in this stage. For others, like **div** and **sqrt** instructions, the Execute stage is *stretched* for multi-cycle execution.

Finally, the result is written back into the SU during the **Store** stage. For a vector operation, an additional **Write** stage is added before the Store stage to relax the timing constraint on a slower and denser vector register file. The pipeline stages after Decode are repeated for every element of a vector operation.

While most scalar register files are latches, vector register files are often implemented by Random Access Memory (RAM) with additional logic to control multi-port access, arbitration and timing. Adding the Write stage in the vector pipeline allows less complicated timing circuitry for access control and a slower RAM may be utilized in an implementation.

3.2.2. Pipeline Interlock

Two common problems in a pipeline design is *structural* and *data dependent hazards* [10]. Structural hazards occur when two pieces of data attempt to use the same stage of a pipeline simultaneously. This hazard may occur if cares are not taken during the design and scheduling of the pipeline. An example in the ANP design is the

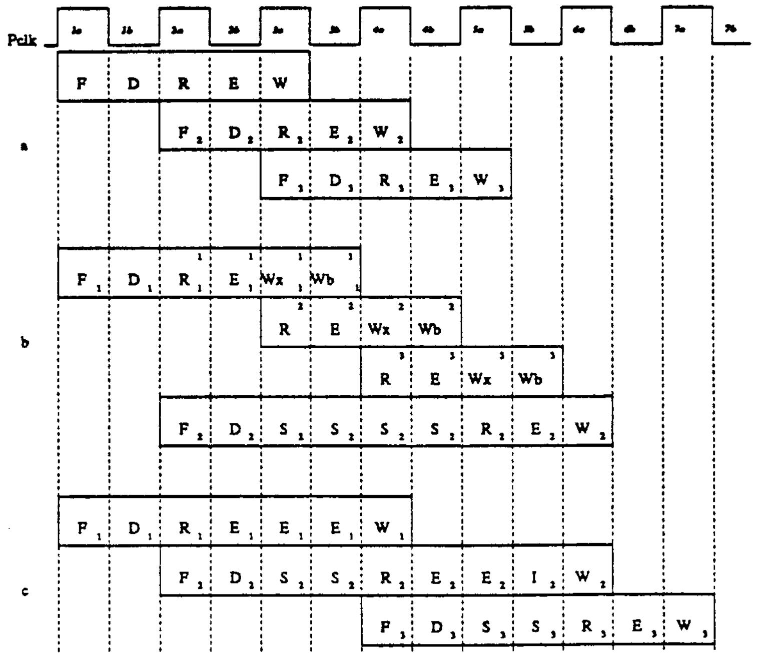


Figure 6: Pipeline Timing Diagram

advancement to the Read stage of the pipeline for the next instruction while a multi-cycle execution is currently taking place. The hazard lies in the SU access and the usage of the EU. This hazard was avoided by: 1) careful design of the pipeline. 2) detail analysis of the inter-dependency between each stages and instructions. 3) use of the reservation table technique described in [10] and the micro-code flow-chart method described in [15].

Data dependent hazards occur when the state of pipeline depends on other stages. These happen when two separate instructions attempt to access the same data (i.e. register, memory location) when their executions are overlapped in the pipeline. There are three such hazards:

Read-After-Write (RAW) - The data read by a (logically) preceding instruction is modified by a following instruction before the access is completed. The data read by the earlier instruction is no longer the initial content but the new value written by the second instruction.

Write-After-Read (WAR) - The data read by a following instruction occurred before it is modified by a (logically) preceding instruction. The data read is the stalled data.

Write-After-Write (WAW) - The data written by a following instruction occurred before it is written by a (logically) preceding instruction. The data stored is the stalled data from the earlier instruction.

The ANP pipeline design is free from data dependent hazards by 1) maintaining the sequentiality of instruction flow. 2) proper scheduling of instructions through the pipeline [10, 14, 17]. 3) employing a pipeline interlocking technique [12]. The RAW and WAW hazard cannot occur in the ANP execution pipeline because of serial execution of instructions. Read and Write (Wx for vector instruction) accesses are scheduled so that they are aligned during the first half of a Fclk cycle (as shown in figure 6). A **Pipeline manager** within the OCU detects and resolves any potential WAR hazards by forwarding the new data from the current Store stage directly to the input of the EU via a feedback path.

3.3. Bus Interface Unit

The Bus Interface Unit (BIU) is responsible for all communications between ANP, PLM and the memory system. The primary function of the BIU is to enforce the **Co-processor Interface Protocol** (CIP) for orderly transfer of instructions and data between ANP, PLM and the memory system. The co-processor interface consists of physical connections that enable the transfers of information between units attached to it, and a controlling mechanism that implements the CIP.

System parameters, such as interrupt enables and mode controls, can be written to a **Control Register** (CR) for initialization and debugging of the ANP hardware. Condition codes and flags from execution of ANP instructions are written into a **Status Register** and can be accessed by the PLM or the memory system from the BIU.

The PLM was designed as a single processor system. The interface between the PLM and the memory system has no support for additional (co-)processors. Thus, an interface protocol is needed for the addition of the ANP. The Co-processor Interface Protocol (CIP) is a bus protocol for initialization, usage of the PMB (for transfer of instruction and data), and exception handling between the ANP, PLM and the memory system.

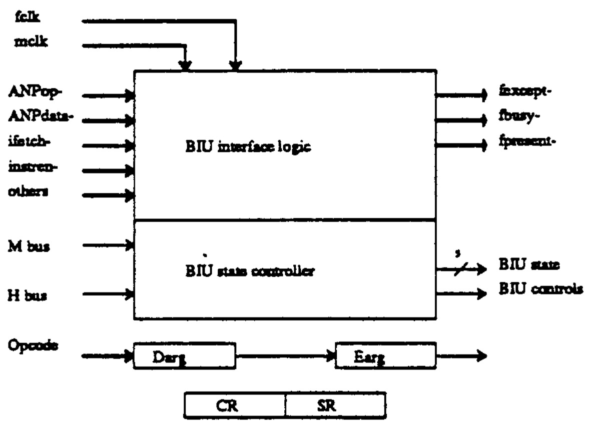


Figure 7: Bus Interface Unit Block Diagram

The initialization protocol is kept close to that of the PLM for two reasons. First, enhancements based on the existing hardware and micro-code require minimal changes in the system. As a result, no change is needed in the existing hardware of the PLM. This allows fast prototyping of the new hardware and software by utilizing the existing PLM debugging environment. Second, keeping the ANP's micro-code in the PLM's code area allows easy modification and enhancement of the micro-routines. Debugging of a complex system like the ANP/PLM is easier. Specialized micro-routines can be loaded for the processing of specific applications (i.e. DSP, CAD).

3.3.1. Transfers of Instruction and Data

The execution model of the ANP primarily consists of a sequence of load, execute and store operations. When an ANP opcode is detected on the Opcode Bus, the BIU enters one of three modes:

Load - The ANP waits for the transfer of operand(s) from memory into its internal Fx/Bx registers via the PMB data Bus. Address calculation and initiation of the memory transfer are done in the PLM.

Execute - is a register to register operation in the ANP. Data coercion may be done to complete the operation. Fbusy* is asserted during the execute cycle when IBuffer is full. Fexcept* may be asserted on the PMB when an external exception is detected.

Store - The ANP transfers operands from its internal Fx/Bx registers to memory along with data type (numeric header) information. The data structure is built on the top of the Heap. Address calculation and initiation of memory transfer are done in the PLM along with the updating of state registers (i.e. H, TR).

When a numeric opcode appears on the Opcode Bus (on the PMB), the ANP as well as PLM receive the opcode. The ANP transfers the opcode into the instruction buffer (IBuffer) during the Fetch stage of the pipeline and decodes it during the Decode stage. The PLM calculates the address and fetches or stores operands needed to complete the operation. The ANP is synchronized to the PLM to receive/supply the operands from/to the memory system respectively. The sharing of the PLM address, data and control busses simplifies the protocol to incorporate the ANP in a PLM system. The co-processor architecture allows the ANP to access all of the resources available to the PLM. This includes instruction (pre-)fetching, (cache) memory access, memory management and interface to the host system. Furthermore, the PLM's debugging environment can be easily enhanced to accommodate the ANP. This allows fast prototyping and verification of the design on both the hardware and software.

Address generations for the ANP are done in the PLM for two reasons.

1) A shadow register set in the ANP is not needed.

2) The overheads of maintaining and synchronizing two separate processor states are eliminated.

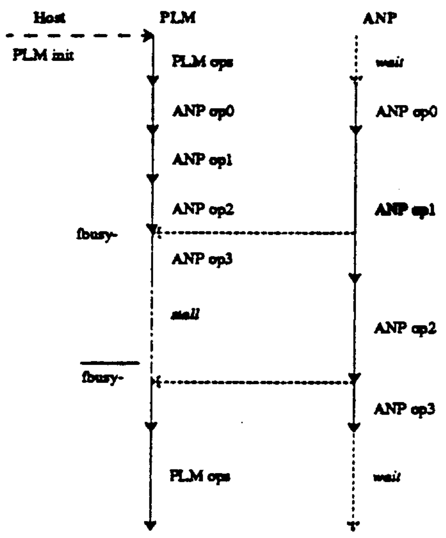
A shadow register set will increase the complexity and cost of the hardware. The overheads for the synchronization and maintenance of two separate sets of data and state registers are large. Any change in one of the register set necessitates a broadcast to the other. For examples, when a PLM instruction modifies an AX register or updates the heap or trail pointer (i.e. laying a choice point or instantiating an unbound variable), the ANP needs to be notified and updated. When a numeric data structure is built or modified on the heap, the PLM needs to be updated. All of these extra transfers will further burden a critical system resource, the PMB. Moreover, additional hand-shaking protocols and signals for synchronization are eliminated. All of these factors will translate into a slower and more expensive system.

Most calculations in the ANP take two Fclk cycles except for multi-cycles and vector operations, and in the event of exceptions. The PLM pre-fetches during execution of an ANP instruction (similar to the execution of a nop) and continues to the next one. During an ANP calculation, the Fbusy* signal may be asserted on the PMB to notify that it is occupied by an ANP instruction execution and the two words IBuffer is full. If the ANP cannot complete the instruction before the next ANP instruction is fetched by the PLM, the PLM will enter a busy wait state until the ANP release the Fbusy* signal. A simplified flow diagram is shown in figure 8.

The ANP normally receives its instructions from the Opcode Bus. When the PLM encounters numeric data types during the execution of an instruction (i.e. floating point unify), it may request service from the ANP via the PMB data bus by asserting ANPop*. Then the PLM enters the busy wait loop until the Fbusy* signal is de-asserted by the ANP. When the ANP completes the current operations (i.e. instructions in the IBuffer), the request from the PLM is executed. The PLM may continue to the next instruction as if the operation is finished within its data path or fail if the numeric operation is not successful[7] (i.e. unification of two unequal floating point numbers).

The dual execution modes of the ANP, both as a parallel processor that shares the same Opcode Bus with the PLM and a slave co-processor to the PLM, allow high performance execution of integrated symbolic and numeric programs while maintaining a tightly-coupled execution model in the system.

[7]Numeric exceptions will not fail the operation immediately but forward the exception to the exception handler in the host.



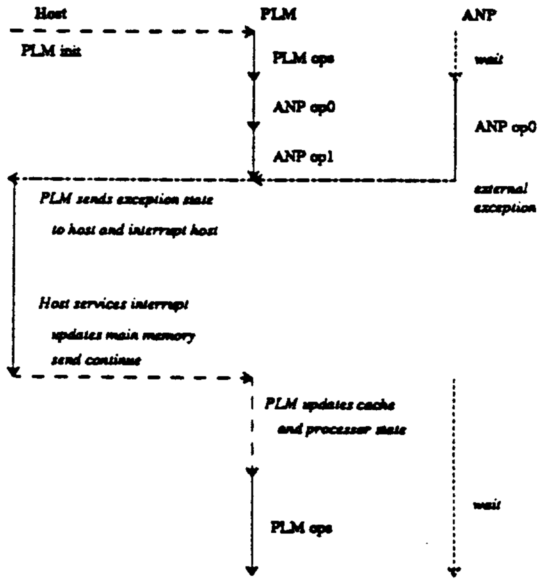

Host
PLM
ANP
PLM init
PLM ops
wait
ANP op0
ANP op0
ANP op1
external exception
PLM sends exception state to host and interrupt host
Host services interrupt updates main memory send continue
PLM updates cache and processor state
wait
PLM ops

Figure 8: Co-processor Interface Protocol **Figure 9: Exception Processing**

3.3.2. Exception Handling

The ANP is viewed as an extension of the PLM data path and register set. The fact that the ANP is physically separate from the PLM is transparent to the programmer. Thus, the exception processing is coordinated by the PLM in a manner that is consistent across all exception types, whether detected during the execution of an instruction native to the PLM or by the ANP instruction.

The processing of an exception detected by the execution of an ANP instruction involves the two basic steps:
Internal exception - The ANP detects an exception that can be resolved within its data path (i.e. operand coercion) and continues on with the execution. Examples are invalid operands and invalid operations.
External exception - The ANP detects an exception and internal exception processing cannot resolve the exception. An external exception is sent to the PLM which in turn forwards it to the host for further exception processing.

The ANP internal exceptions are caused by invalid operands or invalid operations. These are handled by the Dynamic Operand Coercion mechanism in the Operand Coercion Unit. The ANP external exceptions are any error other than internal exceptions. These types of exceptions, which include those described in the IEEE Standard 754, cannot be resolved within the ANP and an exception interrupt is generated to the host processor for further handling.

3.4. Operand Coercion Unit

The Operand Coercion Unit (OCU) implements the Dynamic Operand Coercion (DOC) mechanism. which provides partial instruction decoding, operand type checking and coercion, and vector operation management for the

Execution Unit (EU). Since data typing information is not specified in Prolog code, an ANP instruction with mismatched operand type/size must be detected during run-time by means of an internal exception to the MCU, or by sending a failure or (external) exception signal to the BIU.

The OCU takes as inputs an 8-bit ANP Fopcode (most significant byte of the Darg latch from the BIU) and the Hx registers from the two input operands[8]. It then provides the MCU with the entry point of the micro-routine to perform the instruction or call a data coercion subroutine to process an internal exception. When no (internal exception) coercion is needed, the OCU generates the required opcode, **feu_op**, to the EU. Otherwise, an 8-bit conversion opcode, **conv_op**, is sent to the EU for operand coercion. Upon completion of coercion, the **feu_op** is sent to the EU for the execution of the instruction. Both Feu_op and conv_op are opcodes of the BIT chip set [1].

The second function of the OCU is to maintain vector count for the instruction currently executing in the EU. If there is a mismatched between scalar/vector and instruction/data, a failure signal is sent to the BIU. There are two counters each tracking the elements in the Load and Store stages of the vector pipeline.

3.5. Storage Unit

The Storage Unit (SU) consists of 64 Hx header registers, 64 Fx scalar registers, eight 256-elements Bx vector registers, 128 predefined and user-defined constants in this implementation. Programmer visible floating point registers Fx/Bx, can be configured as 32/64-bit scalar or vector register according to the numeric header stored in the corresponding Hx register. Integers, single precision numbers and the least significant word of double precision numbers are stored in the lower half of a Fx register, the most significant word is stored in the upper half. Vector

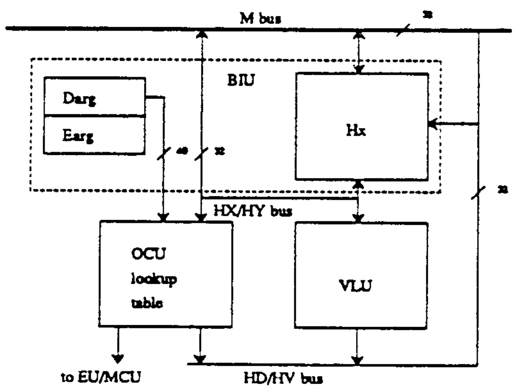


Figure 10: Operand Coercion Unit Block Diagram

[8] Hx register for second source is ignored for monadic instructions.

operands with length up to 65536 are supported in the ANP[9]. For operations that involve two vector operands, same calculation is applied to each corresponding element of the two vectors starting from element zero.

As high precision numeric calculations are needed for scientific data processing, a set of constants are frequently used by these applications. This set of high precision numeric constants, often used in trigonometric, statistical and other numeric calculations, are difficult to derive or to reload each time they are used. In the SU, a set of 128 64-bit pre-defined (or user-defined) constants can be accessed by most numeric operations. All constants are identified with the most significant bit set (equal to one) in the operand byte of an opcode (i.e. register 128 to 255). The ANP register set is shown in figure 4.

The Storage Unit is made up of a set of multi-port register files and several 32/64-bit busses connecting them and to other functional units. Maximum throughput is maintained in the Execution Unit by combining the SU and EU to form a 3-stage execution pipeline for scalar and 4-stage pipeline for vector operations. Two 64-bit reads and a 64-bit write to the scalar Fx register file can be completed in a Fclk cycle. An additional pipeline stage is added before the input of the Vector Bx register to accommodate the slower but denser Bx register file. Two 64-bit reads and a 64-bit write to the Bx register file can be completed in two Fclk cycles. During a vector operation, 64-bit operands are transferred from Bx to the Execution Unit. Results are first transferred into the Wx register before they are stored in the Bx register. Hx register file contains data type information for the corresponding Fx/Bx registers that is used by the BIU and OCU.

Two separate register files are utilized to implement the SU. The vector Bx register file is denser and slower while the scalar register is less dense and faster. Along with the pipelining techniques described above, maximum throughput can be maintained in the Execution Unit for all ANP instructions without paying the high cost of an integrated dense and fast multi-port register file like the Fx. Another advantage of separating the scalar from the vector storage is to provide special hardware support only for specific area in the design without applying it for the entire implementation.

3.6. Execution Unit

The Execution Unit (EU) contains the data path for integer and floating point operations. Its design is based on the Bipolar Integrated Technology (BIT) numeric chip set which consists of an ALU and a Multiplier There are six high speed internal busses connecting the SU, MCU and the EU. Two 32-bit X and Y **Busses** are shared by each of two input ports of the ALU and Multipier chip. The 32-bit **FA** and **FB Bus** connect the output ports of ALU and Multipier to the scalar Fx and Wx vector input registers. Condition code and status from an operation are transferred to the MCU over a 22-bit **Status Bus**. Finally, an 8-bit **I Bus** supplies opcode to the chip set. [1].

[9]Current SU hardware can store up to 256 elements of a vector in a Bx register at a time.

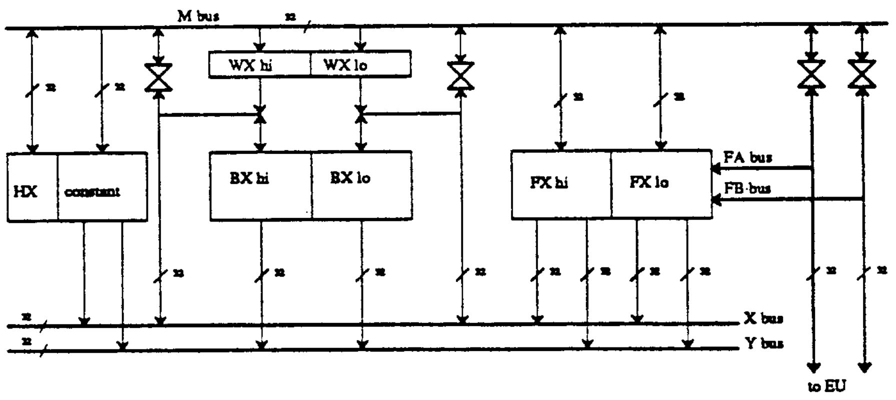


Figure 11: Storage Unit Block Diagram

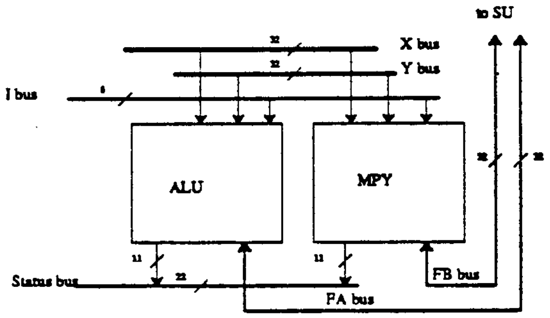


Figure 12: Execution Unit Block Diagram

3.6.1. Scalar Operations

Execution of a scalar instruction (without internal exception) encompasses four stages of the system pipeline each taking a Fclk cycle. Opcode decoding, OCU access and register address generation for the next instruction are overlapped in the third and fourth pipeline stages of a scalar instruction.

Read - Up to two 64-bit operands are transferred from Fx register file to the input ports of ALU or Multiplier over the X and Y Busses in a Fclk cycle. A common 8-bit Feu_op or conv_op is loaded into the I-ports of the chip set.

Execute - Execution in the ALU data path takes a Fclk cycle while Multiplier data path may takes one or more Fclk cycles. This stage may be extended to a multiple of Fclk cycles for multi-cycle execution in the Multiplier. An Idle stage may be added at the end to synchronize with the rising of the PLM (Pclk) cycle.

Store - Result and condition codes are stored away in the Fx register in the SU and Status register in the BIU.

Idle - An Idle stage may be added to synchronize with the rising of the PLM (Cclk) cycle. This stage may be eliminated if the cycle time is same between the ANP and the PLM.

3.6.2. Vector Operations

The execution of a vector instruction is pipelined to maximize the throughput for long vectors. At the same time, short vectors are not penalized with a long flow-through time. A four stage pipeline is chosen to implement such a pipeline. In additional to the first three stages described above, a write stage is added in between Execute and Store stage. This stage is added to accommodate the slower and denser vector register file. A maximum of two elements of a vector can be simultaneously operated in the pipeline.

Write - Result and condition codes are written in the intermediate Wx register and Status register in the BIU respectively.

Store - The content of the Wx register is written in the Bx register file.

3.7. Micro Control Unit

The heart of the ANP is a Micro Control Unit (MCU) which consists of a micro-program sequencer, a 96-bit horizontal writable control store, and other circuitry to handle exception processing and initialization of micro-code. Fast random logic are chosen for the implementation of the sequencer in order to meet the (50ns) Fclk cycle. A 11-bit next micro-program address allows for direct control of up to 2K-words of micro-programs in the writable control store. A block diagram of the MCU is shown in figure 13.

Ability to branch in micro-instruction sequencing directly affects the efficiency of a micro-engine, which in turn affects the performance of an architecture. Delayed branch logic is employed in the MCU to lessen the effect

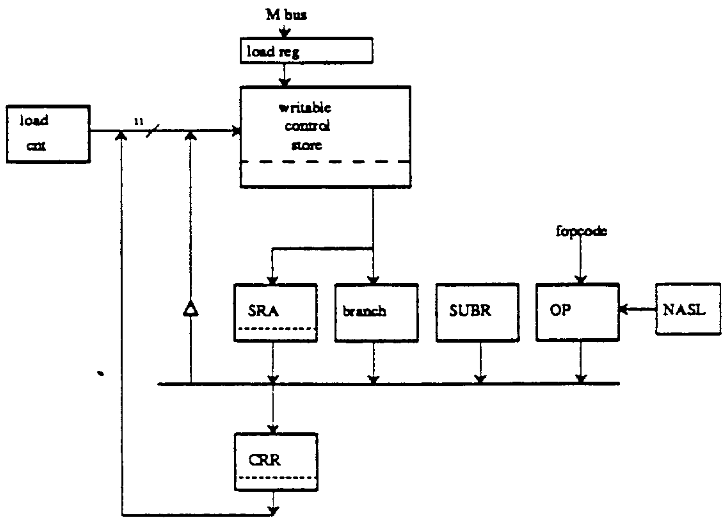


Figure 13: Micro Control Unit Block Diagram

of conditional branching. The MCU micro-sequencer is divided into two control levels. First, the **BRANCH** logic provides a four way branch by modifying the lower two bits of the control store next address seed under the control of a 5-bit field in the micro-word. The 5-bit control field selects up to 32 events from the condition code of previous EU execution or data type of the input operands.

In normal instruction sequencing, one of four address sources are selected to generate the next micro-program address. These address sources include the modified next address seed (**BRANCH**) as described above, a subroutine entry address (**SUBR**) and subroutine return address (**SRA**) and the decoded opcode (**OP**) from the OCU. Selection of these four sources are made in the next address selection logic (NASL) under the control of a 5-bit field in the micro-word.

The concept of subroutine call in high level programming languages allows programmer to write compact and modular code. The SUBR/SRA pair implements a micro-subroutine call by pushing the return address (modified next address seed) in the SRA, a one-word subroutine stack, and making a jump to the subroutine entry address provided by the SUBR. Besides entry points for all the micro-subroutines, the SUBR also contains entry points of common routines such as initialization and exception processing.

Decoded opcodes are provided by the OCU. Normally, the OCU passes an 8-bit feu_op unmodified to OP (with the most significant two bit zeroed out). When an exception occurs in the OCU, a modified 11-bit opcode from the OCU replaces the instruction opcode. This 11-bit conv_op is the entry point of the (internal) exception processing micro-program to process the exception condition. If internal exception processing fails, the PLM is notified by the ANP via assertion of the Fexcept* signal on the PMB.

A mechanism, under the control of the BIU, is provided in the micro-sequencer to support system related functions such as transferring the processor states to the host for debugging. A control return register (**CRR**) is placed at the output of the NASL branching logic to support system level subroutine calls similar to the SRA. A 11-bit boot counter (**BC**) is selected as next micro-program address during system initialization for transferring micro-codes from the PMB into the writable control store.

4. 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.

4.1. Measurement Results

A set of Prolog programs, including modules translated from (double precision) Whetstone benchmark modules, is used to verify the correctness and measure the performance of the ANP/PLM system. Table 2 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. All benchmark programs except mac are scalar operations.

Simulated Benchmark Performance (units in MFLOPS)

		Iterations (double precision calculations)				
test	flop	100	1K	10K	100K	comments
wh1	16	9.67	9.97	10.00	10.00	simple identifier
wh2	16	7.85	7.99	8.00	8.00	array element
wh3	96	6.04	6.04	6.04	6.04	array as parameter
wh4	3	1.57	1.58	1.58	1.58	conditional jump
wh5						omitted in Whetstone
wh6	15	7.70	7.87	7.89	7.89	integer arithmetic
wh7	94	8.68	8.7	8.7	8.7	trig. function
wh8	9	3.10	3.10	3.10	3.10	procedure call
wh9	0					array ref. (no flop count)
wh10	5	9.52	9.95	10.00	10.00	integer arithmetic
wh11	24	9.14	9.22	9.23	9.23	standard function
mac	511	19.88	19.88	19.88	19.88	mult-n-accum (256 elem. vector)

5. Conclusion

Preliminary results obtained from simulation of the ANP/PLM system are encouraging. Simulation results indicate a performance of 10 MFLOPs (in double precision) on Prolog version of some Whetsstone and Linpack benchmarks and close to 20 MFLOPs on some matrix operations. This indicates that the co-processor architecture concept is feasible for a specialized architecture like the PLM. At present, the ANP is under construction and will be integrated into the PLM system for final testing. Additional research is needed in software developments and evaluation of the hardware implementation under real world constraints. The final system based on the ANP/PLM will be an excellent test bed for the studies of CAM/CAE, expert system and other applications that have high performance requirements for integrated symbolic/numeric calculations.

6. 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 Prof. Yale Patt, Dr. Tep Dobry, Dr. Vason Srini, Bruce Holmer, Peter Vanroy and Jerric Tam. We also benefited considerably from the stimulating interactions with Professor William Kahan, Robert Owen of Bipolar Integrated Technology incoporated, the staff at Xenologic incoporated, and John Yung at SUN Micro System.

We also thank Valid Logic Corporation for their generous contribution of equipments, Doug McCafferty and Mike Borsum for their help in obtaining the CAD softwares for schematic capture and circuit simulation. This work is partially sponsored by the Defense Advanced Research Projects Agency (DoD) and monitored by Naval Electronic System Command under contract No. N00039-84-C-0089.

References

1. , *BIT B2110/B2120 Floating Point Chip Set Data Sheet*, Bipolar Integrated Technology, Inc., Beaverton OR (Sep 1987).
2. , *A Proposed Standard for Binary Floating-Point Arithmetic - Task P754*, Microprocessor Standards Committee, IEEE Computer Society (1981).
3. Bush, W.R. et al., "An Advanced Silicon Compiler In Prolog," *Conf. Proc. of the International Conference on Computer Design*, (1987).
4. Despain, A.M. et al., "Aquarius," *Computer Architecture News*, (Mar 1987).
5. Despain, A.M., "A High Performance Prolog Co-Processor," *Proc. of the WESCON 1985*, IEEE Press, (Sep. 1985).
6. Dobry, T.P., "A High Performance Architecture For Prolog," PhD Dissertation, Computer Science Div., Univ. of California, Berkeley CA (May 1987).
7. Dobry, T.P., Despain, A.M., and Patt, Y.N., "Performance Studies of a Prolog Machine Architecture," *Conf. Proc. of the 12th Annual International Symposium of Computer Architecture*, (Jun 1985).
8. Heintze, N.C. et al., "CLP(*R*) and Some Electrical Engineering Problems," *Conf. Proc. of the 4th International Conference of Logic Programming*, (1987).
9. Jaffar, J. and Lassez, J.-L., "Constraint Logic Programming," *Conf. Proc. on Principle of Programming Lanuguage*, (1987).
10. Kogge, P.M., *The Architecture of Pipelined Computers*, Hemisphere Publ. Corp. (1981).
11. Lassez, C., McAloon, K., and Yap, R., "Constraint Logic Programming and Option Trading," IBM Technical Report RC 12649 ().
12. Patterson, D.A., "Reduced Instruction Set Computer," *Communication of the ACM* 28(1)(Jan 1985).
13. Russell, R.M., "Cray-1 Computer System," *Communication of ACM* 21 pp. 63-72 (Jan. 1978).
14. Tomasulo, R.M., "An Efficient Algorithm for Exploiting Mulitple Arithmetic Units," *IBM Journal* 11(Jan 1967).
15. Tredennick, N., *Microprocessor Logic Design: the Flowchart Method*, Digital Press, Beford, MA (1987).
16. Warren, D.H.D., *An Abstract PROLOG Instruction Set*, Menlo Park, CA (June 1985).
17. Weiss, S. and Smith, J.E., "Instruction Issue Logic for Pipelined Supercomputer," *Conf. Proc. of the 11th Annual International Symposium on Computer Architecture*, pp. 110-118 (1984).
18. Yung, R., "The Aquarius Numeric Processor," *Masters Thesis*, Computer Science Div., Univ. of California, (May 1988).