1986

A Microsequenced Prolog Inference Engine

Jeffrey J. Ferguson
University of Central Florida

A MICROSEQUENCED PROLOG INFERENCE ENGINE

BY

JEFFREY JAMES FERGUSON
B.S.E.E., Ohio Northern University, 1982

RESEARCH REPORT

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Graduate Studies Program of the College of Engineering
University of Central Florida
Orlando, Florida

Summer Term
1986

ABSTRACT

Prolog is a symbolic logic language presently emerging among numerous expert system designs. The architecture for a microsequenced Prolog machine (UPM) capable of providing the basic language features to a host computer is proposed. The Prolog machine functions are partitioned into three processor components -- Input/Output, Memory, and Central (CPU), where the design of the Central Processor is emphasized. Detailed discussion outlines the CPU facilities used to implement the forward-chaining and backtracking functions for the UPM. The UPM features are compared to the PLM-1, a microsequenced Prolog inference engine under development at University of California, Berkeley. An emulation of the entire algorithm is provided, as well as a proposed microengine and associated microstore.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

To provide for the information demands of the 1990s, fifth-generation computer systems are now evolving - design emphasis for this generation of machines considers not only the ongoing efforts to increase speed and density, but to include utilization of more varied media, higher software productivity, and application of information technology to those areas in which existing information technology has not yet been applied.

Conventional (von Neumann) computers, structured primarily to perform numeric-intensive, sequential programs are being replaced by architectures which rely primarily on parallel processing, due to the fact that device speed has approached a limit for sequential processing. A second reason for the anticipated replacement of the traditional von Neumann design is the difficulty in realizing basic functions for non-numeric processing of speech, text, graphics, and patterns, and for artificial intelligence fields such as inference, association, and learning. For this reason, reference to "fifth-generation" machines generally implies reference to machines which provide knowledge information processing systems.

The Japanese are a major force in spearheading the efforts to a achieve new architectures for a fifth-generation knowledge information processor. Their function goals, as outlined in Table 1, are indicative of many expert systems presently under development{1}.

TABLE 1. FIFTH GENERATION KNOWLEDGE PROCESSOR GOALS

FUNCTIONS	DESCRIPTION
(1) Problem Solving and Inference function	Carry on logical reasoning using data and knowledge (facts and rules) stored in the system as well as information given to it from outside (user interaction/real time acquisition). Includes inference, inductive inference (including guessing) based on incomplete knowledge.
(2) Knowledge Base Function	Provide storage and retrieval of not only hard data, but also reasonable judgements and test results organized into a knowledge. Incorporates simultaneous utilization of distributed knowledge sources.
(3) Intelligent Interface Function	Increase flexibility in interaction with humans, including handling of speech, graphics, and images.
(4) Intelligent Programming Function	Enhance the intelligence of computers so that they can take over the burden of programming from humans. Ultimate goal is to achieve an ability to automatically convert problems into efficient computer programs.

The UPM of this report will focus on functions (1) and (4) of Table 1, excepting the "guessing" and "real time acquistion" aspects of Table 1.

II. DEFINING AN EXPERT SYSTEM

A key aspect of expert systems technology is that at least three kinds of knowledge have been generally identified as useful in symbolic provlem solving. These are facts, relations between these facts (also referred to as rules), and methods for using these relations in problem solving. Knowledge engineering is the subfield of artificial intelligence concerned with applying knowledge to solve problems that ordinarily require human intelligence. Solving problems in areas of human expertise such as engineering, medicine, and financial advising requires specialized know-how comparable to what a human expert possesses, hence the term "expert system." The method used for problem solving is the facet most heavily dependent on the application environment. For example, by searching for confirming evidence, a diagnostic medical expert system might reason backwards from all potential diseases it knows. Only when it encountered sufficient disconfirming data on the patient's condition would it proceed from one disease to the next candidate{2}. This "backward chaining" methodology offers contrast to a more traditional approach, that of the forward chaining, or data-driven engine{3}.

The expert system employing a forward chainings type of inference mechanism is best illustrated by an example. A user-interactive botany expert system would contain a collection of rules adhering perhaps to the classical "if-then" structure, as in the following:{4}

TABLE 2. PARTIAL RULE BASE FOR A BOTANY SYSTEM

RULE NUMBER	CONTENTS
(1)	Family is Cypress if: Leaf Shape is Scalelike and Class is Gymnosperm
(2)	Family is Pine if: Leaf Shape is Needlelike and Pattern is Random and Class is Gymnosperm
(3)	Family is Pine if: Class is Gymnosperm and Leaf Shape is Needlelike and Pattern is two even lines and Has Silvery Band
(4)	Type is Vine if: Stem is Woody and Position is Creeping
(5)	...

The botanist would then enter facts based on observations of the specimen to be classified (stem is woody, leafshape is needlelike, etc.), and then classify the specimen by issuing a query to the system, such as: To what Family does the specimen belong? The methodology of solving a query is language/architecture dependent and will be approached in later sections, however, the scenario is illustrative of major considerations in designing an expert system, namely:

A) The rule base may be augmented by the experience of many experts, and conclusions may therefore be reached via different paths depending on the knowledge or simply the preference of the contributing experts (note rules (2) and (3), Table 2).

B) As a consequence of A), facilities to accomodate a failure in investigating a possible solution path must exist. This mechanism is referred to as backtracking.

C) The ability of the system to ask the user questions in the event that no rule can be found that unequivocably leads to a family classification.

D) In the event of insufficient facts and/or rules to obtain a classification, the advanced expert system will attempt to yield a "best guess" of the family using a deductive reasoning scheme. This may incorporate certainty factors which designate the level of confidence or validity the data possesses{2}. This is similar to a doctor diagnosing pneumonia to be the ailment of a patient with a severe cough, high temperature, and shortness of breath, even though fluid build-up in the lungs is not yet evident. Certainty factors will not be included in the proposed expert system.

E) Expert systems are distinguished from other artificial intelligence programs in that their power is derived from the knowledge contained in the database, rather than from pre-designed heuristics and search methods. For this reason, explanation facilities to aid the user and justify conclusions are an important aspect of a well-rounded expert system.

Regardless of their differences in technique, expert systems consist of a database to hold rules, facts, and relationships, an inference engine to arrive at conclusions, and an input/output controller to facilitate communication with the programmer/user.

III. PROLOG AS THE LANGUAGE

For the reasons enumerated in section I, numeric-intensive languages are inappropriate for the symbolic, image, and list processing inherent in the application environment of most expert systems. Although early attempts at artificial intelligence have employed top down (von Neumann) formats consisting of more than 1000 "if-then" checks to arrive at conclusions, essentially two languages which contrast this solution methodology have risen to the forefront of artificial intelligence - Prolog and Lisp. Prolog has many parallels with Lisp, specifically: Both are interactive languages designed for symbolic data processing, and neither explicitly incorporates the machine-oriented concepts of assignment and references. Prolog, however, offers further benefits in many aspects, when compared with Lisp{5}:

A) General record structures take the place of Lisp's s-expressions. An unlimited number of different record types may be used. Records with any number of fields are possible, and there are no type restrictions on the fields of a record.

B) Pattern matching replaces the use of selector and constructor functions for operating on structured data.

C) Procedures may have multiple outputs as well as multiple inputs.

D) Input and output arguments of a procedure do not have to be distinguished in advance, but may vary from one call to another. Procedures may thus be multi-purpose.

E) Procedures may generate, via backtracking, a sequence of alternative results.

F) An "incomplete" data structure (containing free variables) may be returned as a procedure's output. The free variables can later be filled in by other procedures. The programmer need not be concerned with the status of a variable (assigned or unassigned) since this status is handled invisibly by the inference engine. This results in the impossibility of encountering an error condition due to an "undefined" operation - at worst Prolog would be unable to generate a solution with 100 percent surety due to insufficient relations in its database.

G) Program and data are identical in form, thus significantly reducing the front end burden of programmer orientation.

The resulting overall simplicity of Prolog (in adherence with fifth-generation design criteria), coupled with its relative youth in the potpourri of programming languages, make it an ideal candidate for a prototype expert system. The UPM described will implement the features of C), D), E), F), and G) enumerated above.

IV. PROLOG ORIENTATION

Since Prolog is not yet widely known (but nonetheless already suffering from the ever-present problem of being syntactically system dependent), a brief presentation of features germane to understanding the inference engine design and emulator routine for the UPM is now undertaken.

Syntax{6}

The primitive Prolog expression is called a clause. An example is:

```
father of(adam,john).                    (1)
```

"Father of" is considered the head of the clause and the arguments are "adam" and "john." A rule exists when the head of the clause is followed by a body consisting of a number (possibly zero) of goals (alternatively referred to as subgoals or procedure calls).

The clause in (1) is termed a fact and might be spoken in English as, "Adam is the father of John," although the order of interpretation is entirely programmer/user dependent as long as consistency is maintained. Other facts might be:

```
valuable(gold).                          (2)
pretty(sally,marie,amy).                 (3)
```

The three clauses above would be considered to have an arity of 2, 1, and 3 respectively, where arity refers to the number of arguments.

Facts obey the following syntax rules:

1) A fact is a clause with zero procedure calls.

2) Facts are finalized with a period.

3) Arguments are literals as indicated by the first letter being a small letter of the alphabet.

An example of a rule would be:

```
grandfather of(X,Z):=father of(X,Y),father of(Y,Z).        (4)
```

The rule of (4) might be interpreted as "The grandfather of any X will be Z if the father of X is Y and the father of Y is Z." Rules obey the following syntax and inference guidelines:

1) ":=" seperates the head from the body.

2) Arguments with capital letters at the beginning designate variables.

3) The ordering of the goals in the clause indicates control information to the inference engine (subgoal satisfaction is attempted in a left-to-right order).

4) Rules are finalized with a period.

Queries are issued to the Prolog system following the insertion of available rules and facts into the database. They are of the following form:

```
grandfather of(V,george)?                  (5)
```

Queries only have a head and must terminate with a question mark. The effect of the query in equation (5) is to ask, "Find some V which has George as a grandfather." The user may elicit a "true/false" response by entering a query already containing literals, as in:

grandfather of(albert,george)? (6)

Effectively asking "Is George the grandfather of Albert?."

Finally, the entry of ";" after an already successful unification indicates the desire for forced backtracking, or to say "go back and find additional solutions, if possible."

Facilities found in many Prolog implementations not covered in the UPM design are the "cut" and mathematical operations.

Prolog Execution Methodology

To execute a goal (initially entered as a query), the system searches for the first clause in the rule and fact base whose head matches or unifies with the goal. If a match is found, the matching clause is then activated and execution (from left to right) of each of the goals of its body (unless a fact, whereby unification of literals to variables is performed) follows in turn. If at any time, the system fails to find a match for a goal, it backtracks by rejecting the most recently activated clause (undoing any substitutions made on the match with the head of the clause). It proceeds by reconsidering the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.{5} This search for alternate rule clauses provides an "or" feature.

Sample Interactions with a Prolog Machine

To ensure a level of familiarity of Prolog sufficient to appreciate the task of the inference engine(s) described, the following programs run on the Prolog emulator of Appendix A are provided. Due to the requirement by the supporting machine to view the comma as a delimiter between input variables, it was necessary to depart from the standard Prolog syntax to the extent that the comma (,) is replaced by the slash (/) throughout the session.

```
? happy(jack):=dating(sally)/pretty(sally).
? happy(jack):=received(jack/raise).
THIS FUNCTOR ALREADY EXISTS...
REDUNDANT ENTRY(R),WRITEOVER(W),OR ABORT(touch enter) DESIRED  ?R
? dating(sally).
? received(jack/raise).
? happy(jack)?
TRUE
? grandfather of(X/Y):=father of(X/A)/father of(A/Y).
? father of(X/Y):=mother of(X/A)/wife of(Y/A).
? father of(X/Y):=sister of(X/A)/mother of(A/B)/wife of(Y/B).
THIS FUNCTOR ALREADY EXISTS...
REDUNDANT ENTRY(R),WRITEOVER(W),OR ABORT(touch enter) DESIRED  ?R
? mother of(betty/evelyn).
? mother of(clara/evelyn).
THIS FUNCTOR ALREADY EXISTS...
REDUNDANT ENTRY(R),WRITEOVER(W),OR ABORT(touch enter) DESIRED  ?R
? sister of(albert/betty).
? mother of(don/grace).
THIS FUNCTOR ALREADY EXISTS...
REDUNDANT ENTRY(R),WRITEOVER(W),OR ABORT(touch enter) DESIRED  ?R
? wife of(fred/grace).
? wife of(don/evelyn).
THIS FUNCTOR ALREADY EXISTS...
REDUNDANT ENTRY(R),WRITEOVER(W),OR ABORT(touch enter) DESIRED  ?R
? grandfather of(albert/ANYBODY)?
ANYBODY=fred
?
```

Figure 1. UPM Emulator Interactions

V. INFERENCE ENGINE CONSIDERATIONS

Overview

As previously discussed, the methodology of solution generation is heavily dependent on the environment. Some Prolog applications currently in use include MYCIN (diagnoses infections), PROSPECTOR (aids geologists in evaluating mineral sites), PUFF (analyzes pulmonary function tests), SACON (provides engineers with advice on structural analysis) and DRILLING ADVISOR (troubleshoots problems encountered when drilling an oil well), to name only a few.

In setting design priorities for such expert systems, attention must be given to ensuring that the strengths of the design are in alignment with the heaviest demands placed on it by the application environment. For example an expert system operating within the real-time constraints afforded by a cruise missile guidance system must emphasize speed of decision making and interaction with information ports. Most often it is the inference engine which represents the critical component in achieving desired performance goals.

What follows is a look at the PLM-1 (Aquarius) project being undertaken at Berkeley{7}. It is provided as a point of comparison for the significant goals to be achieved in the UPM design.

PLM-1

The PLM-1 is intended to handle concurrently both logic and numeric applications as an attached processor. The execution environment for

for PLM-1, as stated by Patt and Despain{7} is "to determine how a very large improvement in performance can be achieved in a machine specialized to solve some very difficult problems which are characterized by intensive numerical calculations tightly coupled to substantial symbolic manipulations." As such, it is designed to operate over an expansive database, a feature which will contrast sharply with the UPM.

The PLM-1 consists of three major modules: the Microengine, the Prolog Engine, and the Prolog Machine Interface; the Microengine is responsible for the control of its own state as well as the two other modules{8}.

The memory space (resident in an NCR/32 system acting as a host) is divided into two areas: the Code Space and the Data Space. The Code Space contains PLM-1 instructions which oversee the microsequencer actions needed to service the current Prolog query. The instructions are divided into five classes:

A) Gets - used to unify with the head of an invoked subgoal.

B) Puts - used to set up the argument registers prior to invoking a subgoal.

C) Unifies - construct and unify structured data.

D) Control - guide sequencing between subgoals, invoke built-in functions.

E) Indexing - select clauses, manage the choice point, and implement cuts.

The Data Space contains 32 bit tagged words representing all data items and state information for a running Prolog program. It is divided into three areas:

A) Trail - keeps track of variable bindings which must be unbound upon backtracking.

B) Stack - LIFO format containing processor state information to be restored on backtracking.

C) Heap - used for storage of lists and structures.

A fourth region, the Push Down List (a scratchpad area used during unifications) is maintained within the Prolog engine.

Processor registers controlling data flow are summarized below:

TABLE 3. PLM-1 WORKING REGISTERS

Register Name	Function
Program Pointer	Contains Code Space Address (CSA) of the next PLM-1 instruction to be executed
Continuation Pointer	Contains CSA of the next instruction to be executed upon successful completion of the current clause
Environment Register	Contains a Data Space Address (DSA) pointing to the current environment frame on the stack
N (Environment size)	Contains the size of the last allocated environment frame on the stack
Backtrack Register	Contains DSA pointing to the active choice point frame on the stack
Heap Pointer	Contains DSA pointing to the current top of the heap
Heap Backtrack Pointer	Contains DSA pointing to the top of the heap at the last backtrack point. Used to reclaim heap space on backtracking
Structure Pointer	Contains DSA pointing into the heap. Shows the location of the next item of a structure currently being processed

The reader is encouraged to consult Despain and Patt{8} at this point for additional insight into the microarchitecture and microengine of PLM-1, as an appreciation of its major design features is an asset in understanding the inference mechanisms of UPM.

UPM Design Criteria

The UPM is also intended to be an attached processor that will augment the facilities of a microcomputer host machine. Consisting of an I/O Sub Processor (handles communication with the host, and interprets Prolog strings - analogous to the PMI of PLM-1), a Memory Processor (provides interface with the main memory of the host, and handles alignment of local and global variables during the unification process), and a Central Processor Unit (CPU - microsequenced inference engine which maintains stacks, pointers, and counters needed for program execution), the UPM offers significant variations from the PLM-1 design in the following aspects:

A) It is intended to work directly with Prolog strings as a source code, via interpretation by the I/O Processor (as opposed to compiled Prolog).

B) The target database size is smaller (typically 64-128k), and is not divided into "Code Space" and "Data Space."

C) Built-in functions are not supported directly by the inference engine, but are interpreted in input (and carried out) by the host when necessary.

D) Due to the separation of the I/O, Memory, and Central Processors, a high degree of parallelism may be achieved. For example, the database may be expanded during execution of a query via a direct memory access path from the I/O processor to the Memory processor. Other facilities for parallelism are expounded upon in the system description.

E) Forced microbranch operations (interrupts in a real-time scenario) are not supported in the UPM, hence, next microaddress selection logic is simplified. This design feature arises from the assumption of a target system consisting of a stand-alone microcomputer.

F) Numeric processing is not provided. Numbers may be handled "brute force" by interpreting as a string, though this method would be inefficient.

G) Perhaps most significantly, all stacks which are maintained in the host memory by PLM-1 (accesses are "traditional" in that only pointers are maintained in the microengine, and read requests must be issued to, and serviced by, the host), are actually hardware resident in the UPM. Numerous stacks and pointers needed in the PLM-1 are eliminated or combined in the UPM. Table 4 relates the processor registers of PLM-1 and their associated UPM equivalent, emphasizing the overall reduction of maintenance pointers required. This scheme reduces and in some cases eliminates the problems discussed by Patt and Despain{8} reagarding a memory bottleneck when referencing the Code Space of the host. By maintaining the Choice Point Stack, Environment File, and the Goal Stack "in house" in the UPM, there is no requirement to shadow the registers or to buffer memory accesses, since each region is independently accessible. Wait states only occur when accessing the host memory for a new clause. There are additional consequences arising from this arrangement, to be addressed in the conclusions.

TABLE 4. CORRELATION BETWEEN PLM-1 AND UPM REGISTERS

PLM-1 Register Name	UPM Register Name
Structure Pointer	none-all Goals of a clause placed in Goal Stack
Continuation Pointer	Goal Stack Pointer (GSP)
Environment	none } inherent in Goal Stack
Program Pointer	none } inherent in Goal Stack
Choice Point Frame	Choice Point Stack (located in Memory Processor)
Push Down List	Argument Translation Table
Environment Register	Environment File and Pointer
Backtrack Register	Choice Point Stack (in CPU) consists of GSP, Environment File and Arguments of all previously unified clauses
Heap Pointer	
Heap Backtrack Pointer	
Trail	
Trail Pointer	

VI. UPM REALIZATION

The focus of the facilities realization portion of this writing will be on the CPU, however, its role in conjunction with the entire module will initially be addressed.

Description of Facilities

Figure 2 illustrates the major facilities of the UPM along with interconnecting buswork and communication protocol (single bit) lines.

Memory Processor

The Memory Processor is presented via the I/O Buffer the goal at the top of the goal stack and argument information consisting of either:

A) Bound variables, or

B) Argument file displacements (for unbound variables).

The rule and fact database is then searched in a top-to-bottom manner for a rule or fact which will unify (i.e., has a matching head and does not have conflicts for bound variables in the same argument position) with the goal.

Three conditions may result, and the Unify Process Logic and File will load the I/O Buffer accordingly:

A) Neither a rule nor fact is found. This causes the fail condition to be transmitted to the CPU.

B) A fact will be found. The arity field is set to 0 and transmitted arguments are all bound literals.

C) A rule will be found. New rules will be returned to the CPU one at a time. The goal field will contain the symbol for the new rule and the argument fields abide by the criteria of Table 6, page 30.

Finally, the Memory Processor has a resident Choice Point Stack (LIFO) which holds addresses of previously successful searches. This facilitates continuation of the top-down search strategy should backtracking be necessary.

Input/Output Processor

The Input/Output Processor provides an interface between the inference engine of the CPU and the host - it performs writing and reading of data to and from a predetermined control word in the address space of the the host.

It must perform bidirectional conversion between text strings of arbitrary length and eight bit (binary) symbols used in the inference process by the CPU - this association is achieved via a symbol table whose address represents the symbol and whose contents are the text string.

The I/O Processor also maintains a Query Status Table (QST) which is always clear between queries. It holds the EF position (within the CPU) of the arguments contained in the initial query along with a tag bit indicating which arguments were input as literals, and which were variables. In the event of successful goal satisfaction, the QST allows the I/O Processor to perform a DMA to the EF and retrieve new bindings for output to the user. Had the query been a True/False question (see the first emulation result, page 11) the I/O Processor will realize this by consulting (anding together) the tag bits of the QST.

Central Processing Unit

The CPU provides the capability for the inference and backtracking functions of the UPM including temporary storage (Environment File, Choice Point Stack, Goal Stack, and Local Variable Translation Table) of all parameters necessary for the resolution of a query. It does not provide direct sequencing control to either of the other two processors.

Table 5 describes the role of each register, file, and stack housed in the UPM and gives its location within the system. Figures 3 and 4 flowchart the actions of the Central and Memory Processors (respectively) encountered in executing a query. Stages shown in dotted lines are parallel processes, where similar dotting schemes indicate simultaneous events. Abbreviations used in all four figures are keyed below:

CP	= Choice Point Stack
CPP	= Choice Point Pointer
EM	= Environment Memory
EMP	= Environment Memory Pointer
GS	= Goal Stack
GSP	= Goal Stack Pointer
<I/O>da=	data available signal from I/O processor to CPU
<I/O>do=	data ready for output from CPU to I/O processor
dr<I/O>=	data received acknowledge to I/O processor from CPU
dl<I/O>=	data latched acknowledge from I/O processor to CPU
---previous four similar for Memory Processor to CPU channels---	
LVP	= Local Variable Translation Table Pointer
LVTT	= Local Variable Translation Table

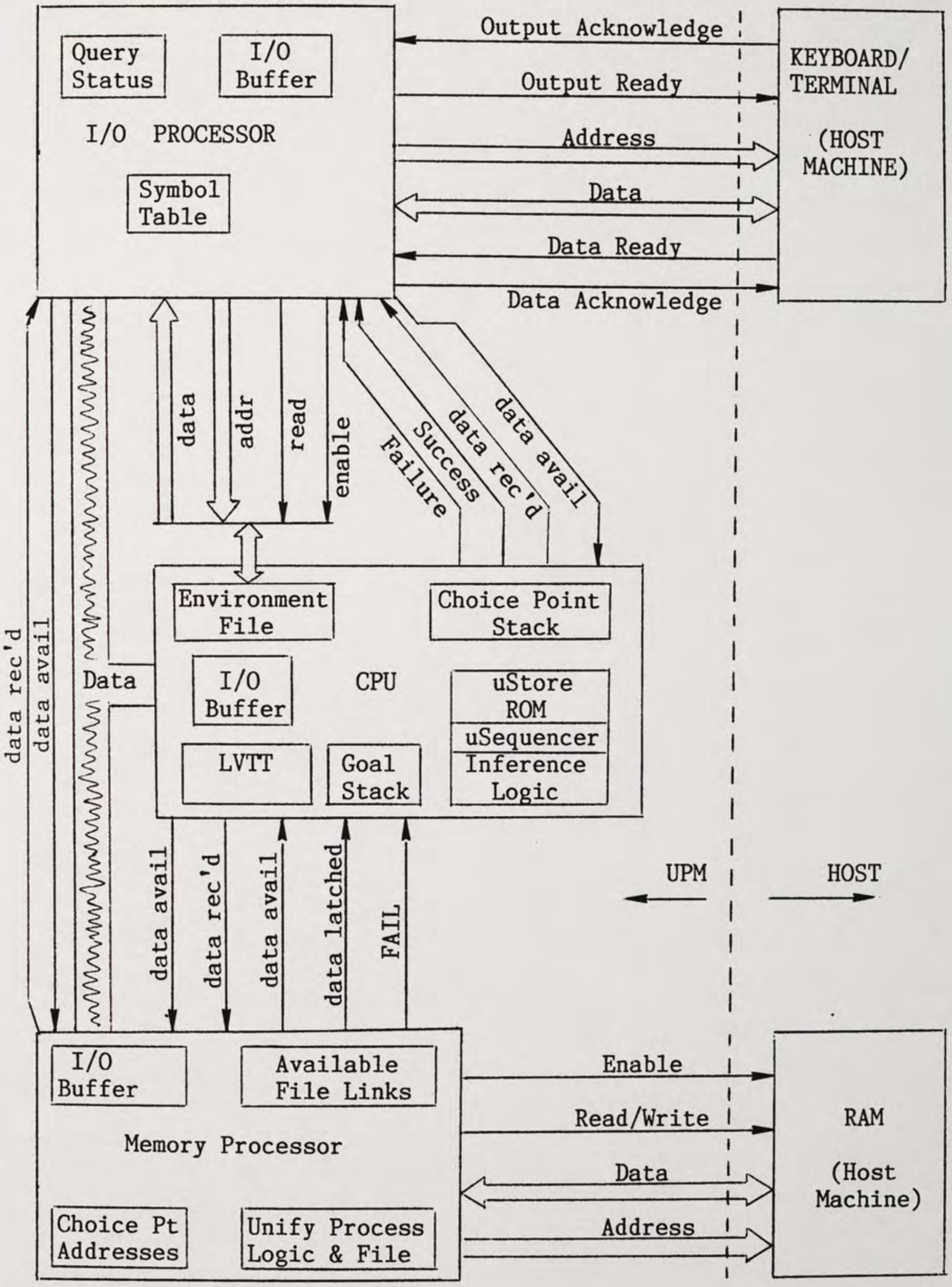


Figure 2. Block Diagram of Facilities

TABLE 5. REGISTER/FILE/STACK ROLES IN THE UPM

Name/Location/Size	Role
Query Status/ I/O Processor/ Unspecified	Retains Symbol Table address of Head and Arguments of Query currently being processed. Also holds the Environment File address of Arguments so that literals may be recovered upon completion of query.
Symbol Table/ I/O Processor/ Unspecified	Holds results of string-to-symbol translations performed by the I/O Processor on initial input of Rules, Facts, or Queries. Strings of arbitrary length are converted to 8 bit binary codes for use in the UPM.
Environment File/ CPU/ 9 Bits * 64 Records	Addresses of Environment File are global variables. Contents of EF are the literals (symbols) of bound variables. A single bit tag field is used to indicate if a binding has occurred.
Choice Point Stack/ CPU/ 33 Bits * 32 Records	Maintains information needed for backtracking, including: GSP of last successful unification, Tag fields to show unbindings that must occur, and addresses in EF of Arguments in last unified goal. (LIFO)
Goal Stack/ CPU/ 46 Bits * 32 Records	Holds all goals to be executed in a LIFO fashion. Includes symbol for goal, arity, and EFP's of arguments in head clause. A maximum of 4 Arguments per head clause is the prototype UPM design limit.
LVTT/ CPU/ 6 Bits * 8 Records	Memory Processor tags any new variables introduced by subgoals of a clause as "local". LVTT holds EF address(EFP) where new variable is placed in EF.
Choice Point Address Stack/ Memory processor/ 32 Records	Upon finding a successful unification for the target clause, Memory Processor writes the address of match in this LIFO stack. Pushed on unification, Popped on backtrack.
Unify Process Logic & File/ Memory Processor/ Unspecified	Performs alignment of subgoal arguments those of the head. Where local variables occur unify file must shadow to check for multiple occurances.

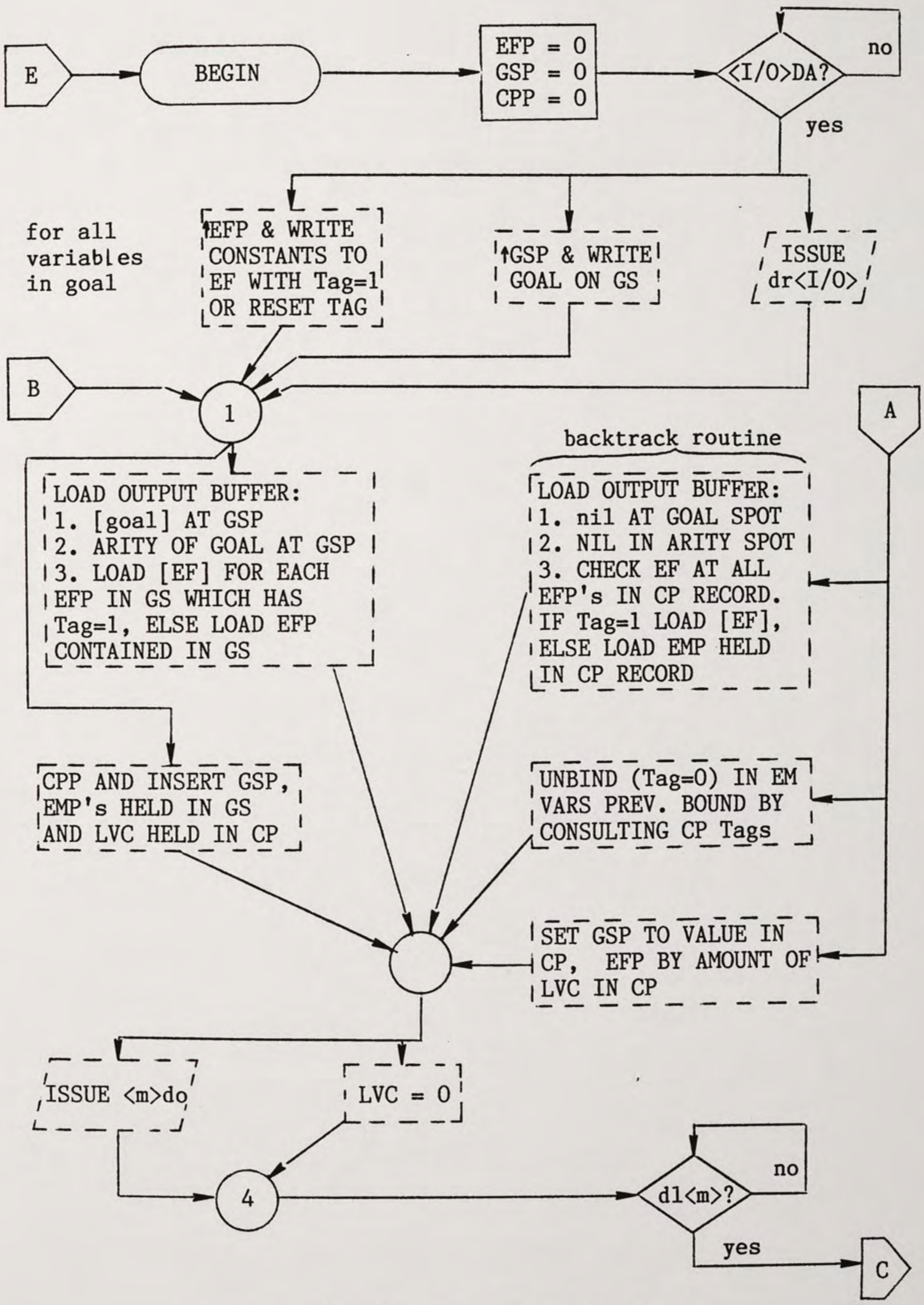
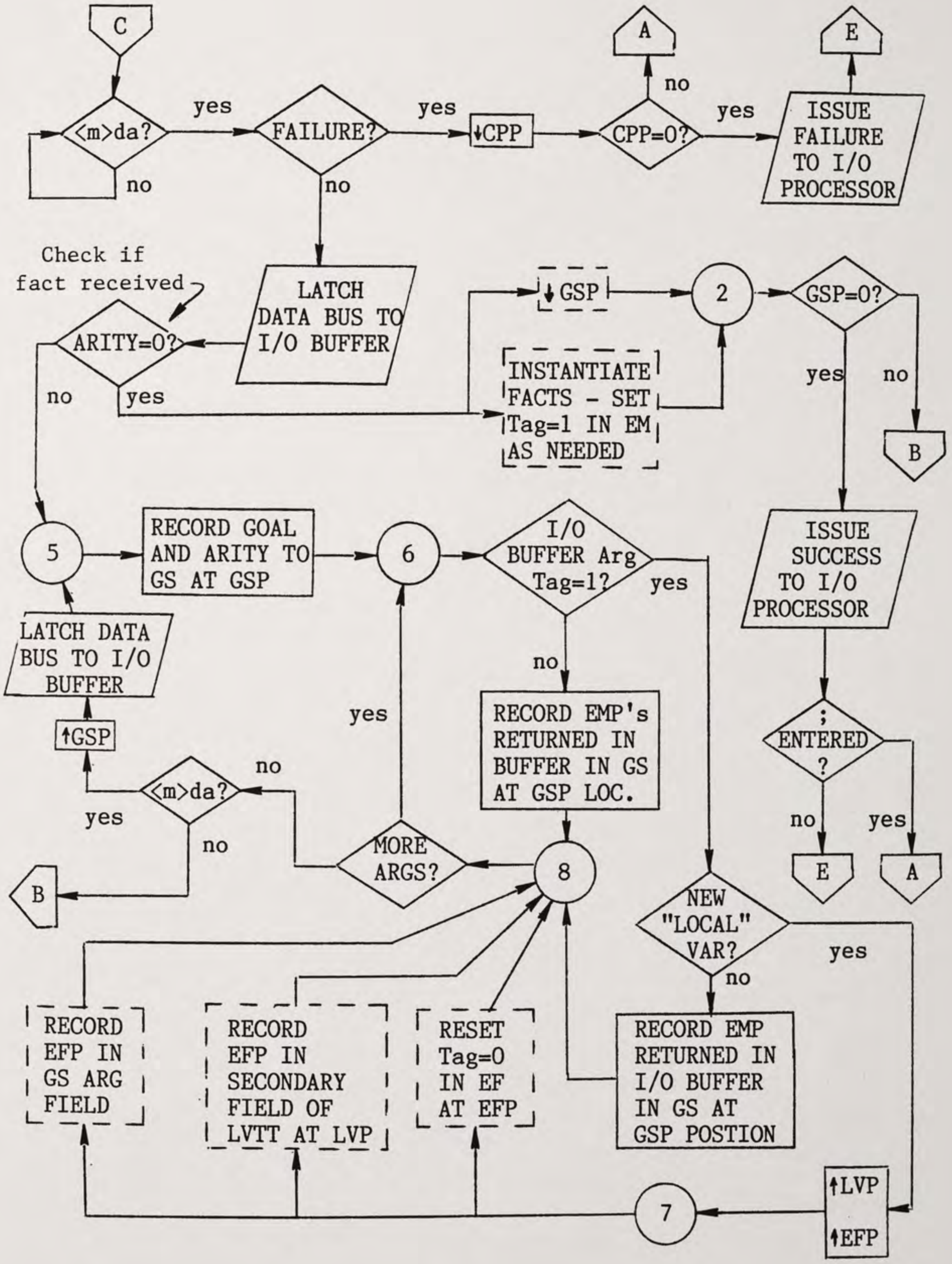
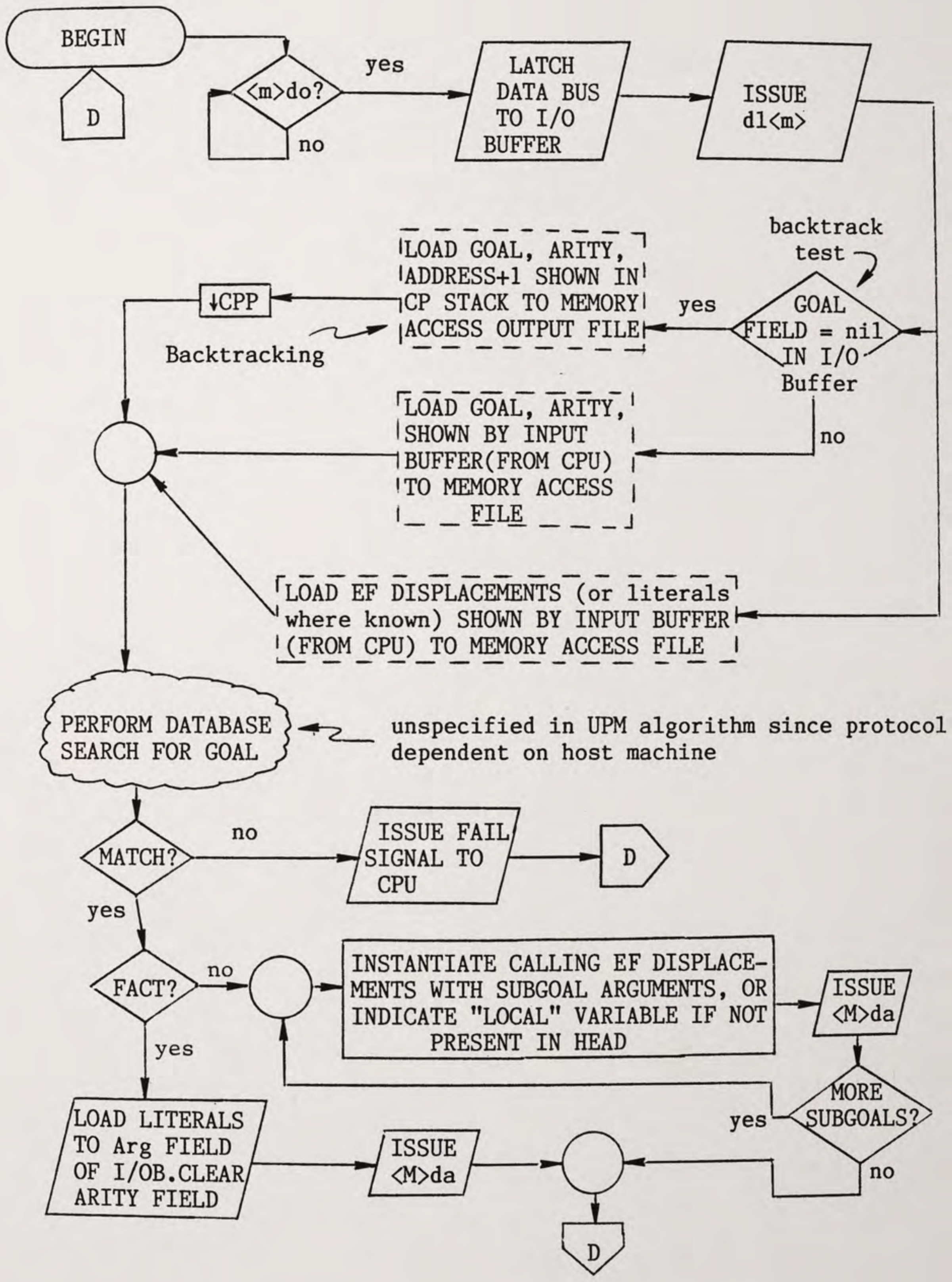

E
BEGIN
EFP = 0
GSP = 0
CPP = 0
<I/O>DA?
no
yes
for all variables in goal
↑EFP & WRITE CONSTANTS TO EF WITH Tag=1 OR RESET TAG
↑GSP & WRITE GOAL ON GS
ISSUE dr<I/O>
B
1
A
backtrack routine
LOAD OUTPUT BUFFER: 1. [goal] AT GSP 2. ARITY OF GOAL AT GSP 3. LOAD [EF] FOR EACH EFP IN GS WHICH HAS Tag=1, ELSE LOAD EFP CONTAINED IN GS
LOAD OUTPUT BUFFER: 1. nil AT GOAL SPOT 2. NIL IN ARITY SPOT 3. CHECK EF AT ALL EFP's IN CP RECORD. IF Tag=1 LOAD [EF], ELSE LOAD EMP HELD IN CP RECORD
CPP AND INSERT GSP, EMP's HELD IN GS AND LVC HELD IN CP
UNBIND (Tag=0) IN EM VARS PREV. BOUND BY CONSULTING CP Tags
SET GSP TO VALUE IN CP, EFP BY AMOUNT OF LVC IN CP
ISSUE <m>do
LVC = 0
4
dl<m>?
no
yes
C

Figure 3. Flowchart of CPU Actions



C
<m>da?
yes
no
FAILURE?
yes
no
↓CPP
CPP=0?
no
yes
A
ISSUE FAILURE TO I/O PROCESSOR
E
Check if fact received
LATCH DATA BUS TO I/O BUFFER
ARITY=0?
no
yes
↓GSP
2
GSP=0?
yes
no
B
INSTANTIATE FACTS - SET Tag=1 IN EM AS NEEDED
5
RECORD GOAL AND ARITY TO GS AT GSP
6
I/O BUFFER Arg Tag=1?
yes
no
ISSUE SUCCESS TO I/O PROCESSOR
LATCH DATA BUS TO I/O BUFFER
↑GSP
yes
RECORD EMP's RETURNED IN BUFFER IN GS AT GSP LOC.
; ENTERED ?
no
yes
E
A
<m>da?
no
yes
no
B
MORE ARGS?
8
NEW "LOCAL" VAR?
yes
no
RECORD EFP IN GS ARG FIELD
RECORD EFP IN SECONDARY FIELD OF LVTT AT LVP
RESET Tag=0 IN EF AT EFP
RECORD EMP RETURNED IN I/O BUFFER IN GS AT GSP POSTION
↑LVP
↑EFP
7

Figure 3. Flowchart of CPU Actions (continued)


BEGIN
D
<m>do?
yes
no
LATCH DATA BUS TO I/O BUFFER
ISSUE dl<m>
backtrack test
GOAL FIELD = nil IN I/O Buffer
yes
no
LOAD GOAL, ARITY, ADDRESS+1 SHOWN IN CP STACK TO MEMORY ACCESS OUTPUT FILE
↓CPP
Backtracking
LOAD GOAL, ARITY, SHOWN BY INPUT BUFFER(FROM CPU) TO MEMORY ACCESS FILE
LOAD EF DISPLACEMENTS (or literals where known) SHOWN BY INPUT BUFFER (FROM CPU) TO MEMORY ACCESS FILE
PERFORM DATABASE SEARCH FOR GOAL
unspecified in UPM algorithm since protocol dependent on host machine
MATCH?
no
ISSUE FAIL SIGNAL TO CPU
D
yes
FACT?
no
INSTANTIATE CALLING EF DISPLACEMENTS WITH SUBGOAL ARGUMENTS, OR INDICATE "LOCAL" VARIABLE IF NOT PRESENT IN HEAD
ISSUE <M>da
MORE SUBGOALS?
yes
no
yes
LOAD LITERALS TO Arg FIELD OF I/OB.CLEAR ARITY FIELD
ISSUE <M>da
D

no

Figure 4. Flowchart of Memory Processor Actions

Register Behavior During a Query

As an illustration of the unification and backtracking scheme employed, the behavior of the major registers, stacks, and files of the UPM will be sequentially displayed throughout the steps needed to process the "grandfather of(albert,ANYBODY)?" query. The rules and facts are assumed to have been entered into the database (note emulation results of this query, page 11).

EVENT #	GS CONTENTS				EF CONTENTS	Tag	CP STACK CONTENTS						
	contents are symbol for goal	Arity	1st arg EF loc	2nd arg EF loc			1st Arg EF loc	2nd Arg EF loc	GSP where began	1st Arg CP tag	2nd Arg CP tag	Amt of EFP↑	address where rule found
1 FLOWCHART POINT 1	GSP→[go]	2	1	2	1)[albert] EFP→2) nil contents are symbol for Albert	1 0	CPP=0 indicates bound literal EF(2) is placeholder for ANYBODY variable						
2 1►3►4► 5►6►8► 6►7►8► 5	GSP→[fo] [go] goal discarded to CP upon unification. Subgoals being written to GS in rev. order.	2	3	2	1)[albert] 2) nil EFP→3) nil EF expanded to allocate position for local variable introduced by [fo] subgoal.	1 0 0	CPP→1	2	1	0	0	1	<go>
							no bindings had occurred in this search						

Figure 4. Register/File/Stack (RFS) Behavior During a Query

3 5►6►8► 6►8►1	▼ [fo] 2 3 2 GSP [fo] 2 1 3 All subgoals of [go] loaded to GS.	1)[albert] 1 2) nil 0 EFP→3) nil 0 See note 3 explanation of Memory Processor action in aligning args for GS loading.	CPP→1 2 1 0 0 1 <go>
4 1►3►4	[fo] 2 3 2 GSP→[fo] 2 1 3	1)[albert] 1 2) nil 0 EFP→3) nil 0	▼ 1 2 1 0 0 1 <go> CPP→1 3 2 Arg locations in EF and GSP of goal are in CP, anticipate successful search.
5 4►5►6► 8►6►7► 8►5►6► 8►6►7► 8►1	[fo] 2 3 2 [wo] 2 3 4 GSP→[mo] 2 1 4 Subgoals [mo], [wo] written in rev. order. 2nd. [fo] rule overwritten since discard to CP	1)[albert] 1 2) nil 0 ▼ 3) nil 0 EFP→4) nil 0 EF expanded to provide place for local var. "A".	1 2 1 0 0 1 <go> CPP→1 3 2 0 0 1 <fo> No args were bound by this rule
6 1►3►4► A►3	[fo] 2 3 2 GSP→[wo] 2 3 4 ▲	1)[albert] 1 2) nil 0 EFP→3) nil 0 ▲	CPP→1 2 1 0 0 1 <go> ▲
	Failure to match [wo] goal caused CP "pop" (backtrack) -- all appropriate pointers decremented.		
7 3►4►5►6 8►6►7►8 5►6►7►8 6►8►5►6 8►6►8►1	[fo] 2 3 2 ▼ [wo] 2 3 4 ▼ [mo] 2 5 4 GSP→[so] 2 1 5 New subgoals in GS, 2nd [fo] overwritten	1)[albert] 1 2) nil 0 ▼ 3) nil 0 ▼ 4) nil 0 EFP→5) nil 0 EFP increments twice, since 2 local variables	▼ 1 2 1 0 0 1 <go> CPP→1 3 2 0 0 2 <fo>

Figure 4. RFS Behavior During a Query (continued)

8 1➤3➤4➤ 2➤1	[fo] [wo] GSP→[mo] ▲ Decrement to access next subgoal parameters	2 2 2	3 3 5	2 4 4	1)[albert] 2) nil 3) nil 4) nil EFP→5)[betty] Literal from fact written	1 0 0 0 1	1 1 CPP→1 2nd arg bound by this fact No new EF locations	2 3 5	1 2 4	0 0 0	0 0 1	1 2 0	<go> <fo> <so>
9 1➤3➤4➤ 2➤1	[fo] GSP→[wo] ▲	2 2	3 3	2 4	1)[albert] 2) nil 3) nil 4)[evelyn] EFP→5)[betty]	1 0 0 1 1	1 1 ▼ 1 CPP→5	2 3 5 4	1 2 4 3	0 0 0 0	0 0 1 1	1 2 0 0	<go> <fo> <so> <mo>
10 1➤3➤4➤ 2➤1	GSP→[fo] ▲	2	3	2	1)[albert] 2) nil 3)[don] ▼ 4)[evelyn] EFP→5)[betty]	1 0 1 1 1	1 1 1 ▼ 5 CPP→3	2 3 5 4 4	1 2 4 3 2	0 0 0 0 1	0 0 1 1 0	1 2 0 0 0	<go> <fo> <so> <mo> <wo>
11 1➤3➤4➤5➤ 6➤6➤8➤6	▼ [wo] GSP→[mo]	2 2	2 3	6 6	1)[albert] 2) nil 3)[don] 4)[evelyn] ▼ 5)[betty] EFP→6) nil	1 0 1 1 1 0	1 1 1 5 ▼ 3 CPP→3	2 3 5 4 4 2	1 2 4 3 2 1	0 0 0 0 1 0	0 0 1 1 0 0	1 2 0 0 0 1	<go> <fo> <so> <mo> <wo> <fo>
12 1➤3➤4➤ 2➤1	GSP→[wo] ▲	2	2	6	1)[albert] 2) nil 3)[don] 4)[evelyn] 5)[betty] EFP→6)[grace]	1 0 1 1 1 1	SAME ▼ 3 CPP→3	2 6	1 2	0 0	0 1	1 0	<fo> <mo>
13 1➤3➤4➤ 2➤E	GSP = 0				1)[albert] 2)[fred] 3)[don] 4)[evelyn] 5)[betty] EFP→6)[grace]	1 1 1 1 1 1	SAME 3 ▼ 3 CPP→2	2 6 6	1 2 1	0 0 1	0 1 0	1 0 0	<fo> <mo> <wo>

Figure 4. RFS Behavior During a Query (continued)

Implementation Notes

1. [...] denotes "symbol for."

2. <...> denotes "address of."

3. Memory processor (not specified in this writing) must perform argument alignment prior to returning subgoals of a clause to CPU. This task merits additional explanation as it is the crux of the unification concept. For the unification of a rule, in event # 3, Memory Processor receives via the data bus (DB):

GOAL	Arg(1)	Arg(2)	Arg(3)	Arg(4)	Tags	Arity
DB(0:7)	DB(8:15)	DB(16:23)	DB(24:31)	DB(32:39)	DB(40:43,44:45)	
[father of]	[albert]	3	nil	nil	1 0 0 0	2

Memory Processor, upon finding <fo> will return goals in reverse order of appearance in rule, i.e.:

[mother of]	1	1	nil	nil	0 1 0 0	2
	1st Arg of calling clause	1st local variable	0 indicates that Arg(1) is not a local variable		1 indicates Arg(2) is a local var.	

Followed by:

[wife of]	2	1	nil	nil	0 1 0 0	2

In the case of a fact, in event # 8, Memory Processor receives:

[sister of]	[albert]	5	nil	nil	1 0 0 0	2

and returns:

[x]	[albert]	[betty]	nil	nil	x x x x	0

The arity field value of zero is the means by which the CPU detects that a fact is being returned - it will recognize all argument values as literals. Thus unifications are actually performed by the Memory Processor. It should be noted that the meaning of the contents of the Argument and Tag fields of the Data Bus have different interpretations depending in direction of transmission. The bus protocol is summarized in Table 6.

TABLE 6. DATA BUS PROTOCOL

Direction of Transmission	If Tag(n) Holds:	It Means:	Argument(n) Holds:
CPU ➤ Memory	0	EF positon is unbound	Displacement in EF of variable
	1	EF literal is available	Symbol for literal of bound variable
Memory ➤ CPU	0	Variable (or literal) returned was in calling clause	Position in calling clause of variable (or literal)
	1	Local Variable being returned	Number of local variable (may be many)

4. The final status of the registers (see event # 13) shows the capability for forced backtracking, should the user desire. Entry of ; at this point would "pop" CP record # 6 to restore GS record # 1; database search would begin with previous [wo] match. Also, EF record # 2 ([fred]) would be unbound in an attempt to see if [evelyn] were the wife of anyone else (Note this Prolog implementation does not prohibit polygamy!).

5. When GSP = 0, the "success" line to the I/O Controller is activated. The I/O Controller then executes a DMA to EF to access new bindings. By consulting the Query Status file, a determination can be made regarding which EF postions need be accessed for output to the user. Failure, had it occurred, would have been indicated by an empty CP stack.

Proposed Microstore and Facilities

The microstore for the UPM is shown in pseudocode format in Figure 5, while a sketch of facilities is contained in Appendix B. The AMD 2910 is chosen as a target controller since its addressing capability is within that required be the microstore of Figure 5. The alternative of cascading AMD 2903s is also available, but needlelessly more cumbersome.

As a result of the de-emphasis of mathematical operations in the UPM, the need for an ALU is nearly obviated - the two uses of the AMD 2901 microprocessor slice is to compare the LVC to the Local Variable Number returned by the memory processor to determine is a LVTT expansion is in order, and to do the EMP decrement of step 35. A savings in microstore width was achieved by installing a look-up ROM to supply the limited (less than eight variations) number of ALU control bits to the nine bit instruction field.

The needed control word width is seventy-nine bits, of which seven are provided for direct input of non-incremental branch addresses. next addresses. Multi-clock cycle subroutines are needed in the steps annotated with an asterisk (those either implementing ALU functions, or performing group transfers) - the subroutines are not specified in the pseudocode.

Finally, some notes regarding the microstore content format:

1. "I/OB" throughout stands for Input/Output Buffer connected to the external Data Bus.

2. The code conforms to AHPL conventions. For example, at address 3, the verbal interpretation would be "Perform a synchronous transfer of the data contained in the [goal] field of the Input/Output Buffer to the [goal] field of the goal stack pointed to by the goal stack pointer.

uStore Address	Commands	Conditional Branch?	Next Address
0	Clear GSP/EFP/CPP	no	inc
1		<I/O>da?	yes 2 no 1
2	↑GSP;output dr<I/O>	no	inc
3	GS[goal(GSP)]←I/OB[goal]	no	inc
4	↑EFP	I/OB [arg]=0?	yes 6 no 4
*5	EF[tag(n)]←I/OB[tag(n)] EF[arg(n)]←I/OB[arg(n)]	no	inc
6	↑CPP;I/OB[goal]←GS[goal(GSP)] ;I/OB[arity]←GS[arity(gsp)]	no	inc
7	I/OB[arg(1)]←(GS[arg(1)]! EF[arg(GS[arg(1)])])*(GS[tag(1)],GS[tag(1)]);CP[arg(1)]← EF(GS[arg(1)]);I/OB[tag(1)]← (1!0)*(GS[tag(1)],GS[tag(1)])	no	inc
8	---same as 7, except all subscripts (2)---	no	inc
9	---same as 7, except all subscripts (3)---	no	inc
10	---same as 7, except all subscripts (4)---	no	inc
11	output <M>do;LVC←0	dl<M>?	yes 12 no 11
12		<M>da?	yes 13 no 12
13		Failure?	yes 33 no 14
14	I/OB←Data Bus	no	inc

Figure 6. Microstore Contents

15		Arity=0?	yes 41 no 16
16	GS[goal]←I/OB[goal];GS[arity]←I/OB[arity]	I/OB[tag(1)]=1?	yes 21 no 17
17	GS[arg(1)]←I/OB[arg(1)]	I/OB[tag(2)]=1?	yes 22 no 18
18	GS[arg(2)]←I/OB[arg(2)]	I/OB[tag(3)]=1?	yes 23 no 19
19	GS[arg(3)]←I/OB[arg(3)]	I/OB[tag(4)]=1?	yes 24 no 20
20	GS[arg(4)]←I/OB[arg(4)]	<M>da?	yes 14 no 21
*21		LV#>LVC?	yes 25 no 17
*22		LV#>LVC?	yes 27 no 18
*23		LV#>LVC?	yes 29 no 19
*24		LV#>LVC?	yes 31 no 20
25	↑LVP;↑EFP	no	inc
26	EF[tag(EMP)]←0;GS[arg(1)]←EFP;LVTT(LVP)←EFP	I/OB[tag(2)]=1?	yes 22 no 18
27	↑LVP;↑EFP	no	inc
28	EF[tag(EMP)]←0;GS[arg(2)]←EFP;LVTT(LVP)←EFP	I/OB[tag(3)]=1?	yes 23 no 19
29	↑LVP;↑EFP	no	inc
30	EF[tag(EFP)]←0;GS[arg(3)]←EFP;LVTT(LVP)←EFP	I/OB[tag(4)]=1?	yes 24 no 20
31	↑LVP;↑EFP	no	inc

Figure 6. Microstore Contents (continued)

32	EF[Tag(EFP)]←0;GS[arg(4)] EFP;LVTT(LVP)←EFP	<M>da?	yes 14 no 6
33	↑CPP	no	inc
34		CPP=0?	yes 40 no 35
*35	I/OB[goal]←0;I/OB[arity]←0; GSP←CP[gsp(CPP)];EFP←EFP-CP[lvc(CPP)]	no	inc
36	I/OB[arg(1)]*CP[tag1(CPP)]← EF(CP[arg(1)]);EF(CP[arg(1)]) *CP[tag1(CPP)]←0	no	inc
37	---same as 36, except all 1's become 2's---	no	inc
38	---same as 36, except all 1's become 3's---	no	inc
39	---same as 36, except all 1's become 4's---	no	11
40	output FAILURE to I/O Proc.	no	0
41	↑GSP;EF(GS[arg(1)])*EF(GS[tag (1)]←I/OB[arg(1)]	no	42
42	EF(GS[arg(2)])*EF(GS[tag(2)]← I/OB[arg(2)]	no	43
43	EF(GS[arg(3)])*EF(GS[tag(3)]← I/OB[arg(3)]	no	44
44	EF(GS[arg(4)])*EF(GS[tag(4)]← I/OB[arg(4)]	GSP=0?	yes 45 no 6

Figure 6. Microstore Contents (continued)

VII. CONCLUSIONS

Prolog has been determined by numerous artificial intelligence research communities to be a language worthy of investigation, and this report has touched on many of the points of discussion carried on by these efforts. An attempts has been made to illustrate the relative ease with which Prolog may be both implemented and exercised by an individual intent on expeditiously interrogating the knowledge of a small scale database.

Major differences between UPM and PLM-1 include:

1. Instruction and Data Types:

A. PLM-1: Queries are compiled from C-Prolog into an abstract Prolog instruction set which includes I/O, memory reference, and control instructions. The traditional top-to-bottom, left-to-right execution/ search strategy associated with Prolog inference engines is controlled by the compiled code.

B. UPM: Queries and program strings have the same form: encoded Prolog strings. The data types used are similar to PLM-1, with the exception of local variables. In the UPM, all variables are allotted a position in the EF, thus "temporary" variables are never truly destroyed between clauses, as in the PLM-1.

A trade-off is apparent in handling unbound arguments - less maintenance is required (thus increased speed) by the UPM, however, the EF capability may be exceeded in a program involving numerous local variables.

2. Architecture:

A. PLM-1: A single microengine controls I/O, memory, and Prolog inference tasks. With exception of the Push-Down List, the Data Space internal to the PLM-1 consists only of pointers and indexes used in addressing and tracking host memory space.

B. UPM: Three processors are specified, and major operations are parceled out between the three. Especially significant is that the unification function is combined with memory processing. The UPM contains all inference information resident in dedicated RAM.

A speed increase for manipulation of overhead parameters (Choice Points, Environment, Trail) can be expected in the UPM, due to reduction of memory access traffic at the host interface bottleneck. The consequence of having an invarient (hardware limited) ceiling on the number of records available for tracking bindings, arguments, and choice points might be intolerable on large scale systems anticipating a deep level of backtracking, or numerous variables.

For the prototype UPM, the limits are as follows: (see Table 5)

256	symbols (8 bit argument fields)
32	levels of goal nesting per query
32	nodes retraceable on backtracking
64	total arguments per query
8	local variables per clause

These choices were arbitrary and made at an early stage of the design process. Examination of the target operational environment might reveal the need to vary these values. The advantage of a microsequenced system is apparent should this action be required, since expansion is achieved simply by adding memory and using additional microstore fields to carry the extra addressing bits. Widening of the data bus argument fields is necessary if more than 256 arguments are needed.

The basic emulation program of Appendix A serves to validate the design, as well as provide a test base for further capability examination, however, it falls short of yielding the necessary time consumption parameters needed to document the speed improvement claimed as a byproduct of the design. Further efforts should thus be centered on building the hardware portion of the UPM specification, and performing benchmark comparisons to verify its strengths and weaknesses.

APPENDIX A

BASIC EMULATION OF UPM

```
1 ' ***********************************************************************
2 ' *                                                                     *
3 ' *  "PROLOG EMULATOR"                                     MAY 30, 1986 *
4 ' *                                                 JEFFREY J. FERGUSON *
5 ' *                                                                     *
6 ' *  THIS PROGRAM USES MICROSOFT GW-BASIC (VER 2) TO EMULATE THE        *
7 ' *  ACTIONS OF THE CENTRAL, I/O, AND MEMORY PROCESSORS REQUIRED TO     *
8 ' *  SUPPORT THE PROLOG DIALECT DESCRIBED BY CLOCKSIN AND MELLISH       *
9 ' *  IN "PROGRAMMING IN PROLOG" {6}.  THE ACCOMPANYING REPORT           *
10 ' *  OUTLINES THE SYNTAX FEATURES.                                      *
11 ' *                                                                    *
12 ' *  VARIABLE USAGE:                                                   *
13 ' *                                                                    *
14 ' *  GS,GS$ = GOAL STACK                                               *
15 ' *       GS$(X) = [goal](X)                                           *
16 ' *       GS1(X,1) = EF POSITION OF 1st VARIABLE OF [goal](X)          *
17 ' *                                                                    *
18 ' *  CP = CHOICE POINT PARAMETERS                                      *
19 ' *       CP(N) = MEMORY ADDRESS OF PREVIOUS MATCH (RULE OR FACT)      *
20 ' *       CP1(N,M) = TAG FIELDS OF CHOICE POINT(N) INDICATING IF       *
21 ' *       VARIABLE(M) WAS BOUND BY THIS STEP. 1 IF BOUND, 0 IF NOT.    *
22 ' *       CP2(N,M):                                                    *
23 ' *            FOR M=1TO4, CP2 HOLDS EF LOCATION OF VARIABLE(M) FOR    *
24 ' *            THE 4 VARIABLES USED IN CHOICE POINT(N). HOLDS 0 IF     *
25 ' *            VARIABLE POSITION UNUSED.                               *
26 ' *            FOR M=5, CP2 HOLDS GS POSITION WHERE GOAL(N) WAS        *
27 ' *            OBTAINED FOR COMPARISON TO DATABASE.                    *
28 ' *       CP3(X) = RECORDS AMOUNT OF EF SIZE INCREASE CAUSED BY        *
29 ' *       CHOICE POINT(X), DUE TO NEW LOCAL VARIABLES                  *
30 ' *                                                                    *
31 ' *  EF$,EF = ENVIRONMENT FILE                                         *
32 ' *       EF$(X) = POSITION X IS VARIABLE. IF BOUND, CONTENTS ARE      *
33 ' *       SYMBOL FOR THE VARIABLE(LITERAL).                            *
34 ' *       EF1(X) = TAG FIELD TO INDICATE IF CONTENTS OF EF$(X) ARE     *
35 ' *       A LITERAL. 1 = LITERAL, 0 = VARIABLE(UNBOUND).               *
36 ' *                                                                    *
37 ' *  LV$,LV = LOCAL VARIABLE TRANSLATION TABLE                         *
38 ' *       LV$(N) HOLDS SYMBOL FOR LOCAL VARIABLE(N) INTRODUCED BY      *
39 ' *       CLAUSE(X).                                                   *
40 ' *       LV(N) HOLDS ENVIRONMENT FILE POSITION OF LOCAL VARIABLE      *
41 ' *       HELD IN LV$(N).                                              *
42 ' *                                                                    *
43 ' **********************************************************************
```

```
50 '
140 '
150 OPTION BASE 1
160 DIM MATRIX$(100,5,5)
170 DIM EF$(50), EF1(50), CP2(60,5), CP(60), GS$(20), GS1(20,4),CP1(60,4)
180 DIM KEEPER$(4), GB(4), LV$(20), LV1(20), CP3(60)
190 '
200 '----------ACCEPT KEYBOARD INPUT HERE------------------------------
210 '
220 TRACE = 0:GOOD = 0
230 RR$ = "J": K=0: INPUT KAY$
240 IF KAY$ = "t" THEN TRACE = 1:GOTO 230
250 IF KAY$ = "nt" THEN TRACE = 0:GOTO 230
260 IF KAY$=";" GOTO 1220                        '---FORCED BACKTRACKING
270 IF RIGHT$(KAY$,1) = "." GOTO 2740          '---MUST BE A RULE OR FACT
280 '
290 '----------THIS AREA DOES QUERIES------------------------------------
300 '
310 EFP=1: GSP=1: CPP=0
320 WAL1=1
330 GOSUB 2040                    '---MUST LOAD GOAL STACK WITH NEW QUERY
340 IF OOT1 = 0 GOTO 230          '---TOO MANY ARGUMENTS IN QUERY-INVALID
350 '
360 '----------EXECUTE TOP OF GOAL STACK--------------------------------
370 '
380 ARITY=1
390 WHILE GS1(GSP,ARITY) <> 0
400 ARITY = ARITY + 1                            '---FINDS ARITY OF GOAL
410 WEND
420 ARITY = ARITY - 1
430 RULE$ = GS$(GSP)
440 GOSUB 2340                 '---FIND APPROXIMATE FILE LOCATION OF GOAL
450 IF TRACE = 0 GOTO 680
460 LOCATE 1,2:WIDTH 40:CLS
470 PRINT "LOC# [goal] arg(1) arg(2) arg(3) arg(4)"
480 LOCATE 2,1,0:PRINT "----------------------------------------"
490 FOR A5 = 1 TO 10
500 LOCATE A5+2,1:PRINT USING" ##  \    \ ##    ##    ##    ##    ##
";A5,GS$(A5),GS1(A5,1),GS1(A5,2),GS1(A5,3),GS1(A5,4)
510 NEXT A5
520 LOCATE 14,17:PRINT "GSP = ";GSP
530 LOCATE 24,1,1:INPUT "",R$:CLS
540 LOCATE 1,2:WIDTH 40:PRINT "       EF LOC#   [arg(N)]   tag"
550 LOCATE 2,1,0:PRINT"       --------------------------"
560 FOR A5 = 1 TO 14
570 LOCATE A5+2,1:PRINT USING"       ##      \     \   ##";A5,EF$(A5
),EF1(A5)
580 NEXT A5
590 LOCATE 18,21:PRINT "EFP = ";EFP
600 LOCATE 24,1,1:INPUT "",R$:CLS
610 LOCATE 1,2:WIDTH 40:PRINT "LOC# EF(1) EF(2) GSP t(1) t(2) EFP ADDR"
```

```
620 LOCATE 2,1,0:PRINT"---------------------------------------"
630 FOR A5 = 1 TO 10
640 LOCATE A5+2,1:PRINT USING" ##   ##    ##   ##   ##   ##   ## ##  #
##";A5,CP2(A5,1),CP2(A5,2),CP1(A5,1),CP1(A5,2),CP3(A5),CP(A5)
650 NEXT A5
660 LOCATE 14,17:PRINT "CPP = ";CPP
670 LOCATE 24,1,1:INPUT "",R$:CLS:WIDTH 80:IF GOOD = 1 GOTO 1580
680 GOSUB 2390                         '---FIND EXACT FUNCTOR AND ARITY
690 IF FAIL = 1 THEN IF CPP = 0 GOTO 230 ELSE GOTO 1220      '---RESTART
700 '
710 '----------SUCCESSFUL SEARCH-MUST STORE OLD CALLING VARIABLES IN CP
720 '----------ASSOCIATE "LOCAL" VARS WITH CALLING VARS FROM GOAL STACK
730 '----------CREATE NEW EF POSITIONS WHERE NO ASSOCIATION EXISTS
740 '
750 GOSUB 2550
760 LVC=0
770 FOR TTT = 1 TO 20
780 LV$(TTT) = ""
790 NEXT TTT
800 IF MATRIX$(HASH,2,1) = "" GOTO 1350'---MUST BE A FACT IF NO SUBGOALS
810 GOSUB 2670                   '---SAVE ARGUMENT FILE DISPLACEMENTS SO
THAT PRESENT GOAL STACK POSITION CAN BE WRITTEN OVER WITH LAST SUBGOAL
820 '
830 '-----BEGIN LOADING SUBGOALS INTO GS BEGINNING WITH LAST SUBGOAL
840 '
850 FOR LL = 5 TO 2 STEP -1
860 IF MATRIX$(HASH,LL,1) = "" GOTO 1060       '---NO SUBGOAL HERE, EMPTY
870 GS$(GSP) = MATRIX$(HASH,LL,1)
880    FOR KK = 2 TO 5             '---NOW TRANSLATE ARGUMENTS TO GOAL STACK
890    IF MATRIX$(HASH,LL,KK) = "" THEN GS1(GSP,KK-1)=0:GOTO 1040
 '---NO ARGUMENT HERE
900        FOR JJ = 2 TO ARITY + 1
910        IF MATRIX$(HASH,LL,KK) = MATRIX$(HASH,1,JJ) THEN GS1(GSP,KK-1)
= GB(JJ-1): GOTO 1040'--SCAN TARGET HEAD CLAUSE TO FIND ASSOCIATION WITH
SUBGOAL VARIABLES
920        NEXT JJ
930    TTT=1
940    WHILE TTT <= LVC          '---CHECK IF LOCAL VARIABLE ALREADY EXISTS
950    IF LV$(TTT) = MATRIX$(HASH,LL,KK) THEN GS1(GSP,KK-1) = LV1(TTT):
GOTO 1040
960    TTT = TTT+1
970    WEND
980 '-----MUST HAVE NEW LOCAL VARIABLE,SO EXPAND LOCAL VARIABLE TRACKING
990 '-----ALSO MUST EXPAND ENVIRONMENT FILE
1000   EFP=EFP+1: HOLD = ASC(LEFT$(MATRIX$(HASH,LL,KK),1))
1010   IF HOLD >= 97 AND HOLD <= 122 THEN EF1(EFP) = 1:EF$(EFP) =
MATRIX$(HASH,LL,KK) ELSE  EF1(EFP) = 0
1020   LVC=LVC+1: LV$(LVC) = MATRIX$(HASH,LL,KK): LV1(LVC) = EFP
1030   GS1(GSP,KK-1) = EFP
1040   NEXT KK
1050 GSP=GSP+1
```

```
1060 NEXT LL
1070 GSP=GSP-1
1080 CP3(CPP) = LVC
1090 GOTO 380           '---THIS GOAL COMPLETE - CYCLE BACK FOR NEXT GOAL
1100 '
1110 '----------ARGUMENT FILE LOADING SUBROUTINE--------------------------
1120 '
1130 TEST$ = MID$(KAY$,VAL1+1,VAL2-VAL1-1)
1140 X$ = LEFT$(TEST$,1)
1150 IF ASC(X$) >= 97 AND ASC(X$) <= 122 THEN EF$(TT)=TEST$: EF1(TT)=1:
ELSE EF1(TT)=0
1160 KEEPER$(TT) = TEST$
1170 RETURN
1180 '
1190 '----------RESTART IS ACCESSED AFTER FAILED SEARCHES FOR A FUNCTOR
1200 '----------"POP" CHOICE POINT TO DETERMINE SEARCH POINT
1210 '
1220 GSP = CP2(CPP,5)            '---REWRITE GOAL STACK FROM LAST MATCH
1230 GS$(GSP) = MATRIX$(CP(CPP),1,1)      '---NOW HAVE PREVIOUS FUNCTOR
1240 FOR LL = 1 TO 4
1250 GS1(GSP,LL) = CP2(CPP,LL)     '---PLACE AF LOCATIONS IN GOAL STACK
1260 IF CP1(CPP,LL)=1 THEN EF1(CP2(CPP,LL))=0 '---UNBIND ARGS IF NEEDED
1270 NEXT LL
1280 HASH = CP(CPP) + 1
1290 EFP = EFP -CP3(CPP)
1300 CPP = CPP - 1
1310 GOTO 450                                      '---BACKTRACK WITH THIS
1320 '
1330 '----THIS PORTION INSTANTIATES FACTS WITH THEIR RESPECTIVE EF LOCS
1340 '
1350 FOR LL = 1 TO 4
1360 HOLD(LL) = 0
1370 NEXT LL
1380 FOR LL = 1 TO 4
1390 TT = GS1(GSP,LL): IF TT = 0 THEN CP1(CPP,LL)=0: GOTO 1510 '---DONE
1400 TEST$ = MATRIX$(HASH,1,LL+1)
1410 X$ = LEFT$(TEST$,1)
1420 IF ASC(X$) <= 96 OR ASC(X$) >= 123 THEN CP1(CPP,LL)=0: GOTO 1510
1430 IF EF1(TT) = 0 THEN EF$(TT) = TEST$: EF1(TT)=1: CP1(CPP,LL)=1:HOLD
(LL) = TT:GOTO 1510
1440 IF TEST$ = EF$(TT) THEN CP1(CPP,LL)=0: GOTO 1510
1450 FAIL = 0: GOSUB 2420: CPP=CPP-1
1460 FOR LL = 1 TO 4
1470 IF HOLD(LL) <> 0 THEN EF1(HOLD(LL)) = 0
1480 NEXT LL
1490 GOTO 690
1500 '-----CHECKED FOR SMALL CHARACTER - BIND IF IT IS
1510 NEXT LL
1520 GSP = GSP - 1
1530 IF GSP <> 0 GOTO 380                      '---DO NEXT GOAL ON GOAL STACK
1540 '
```

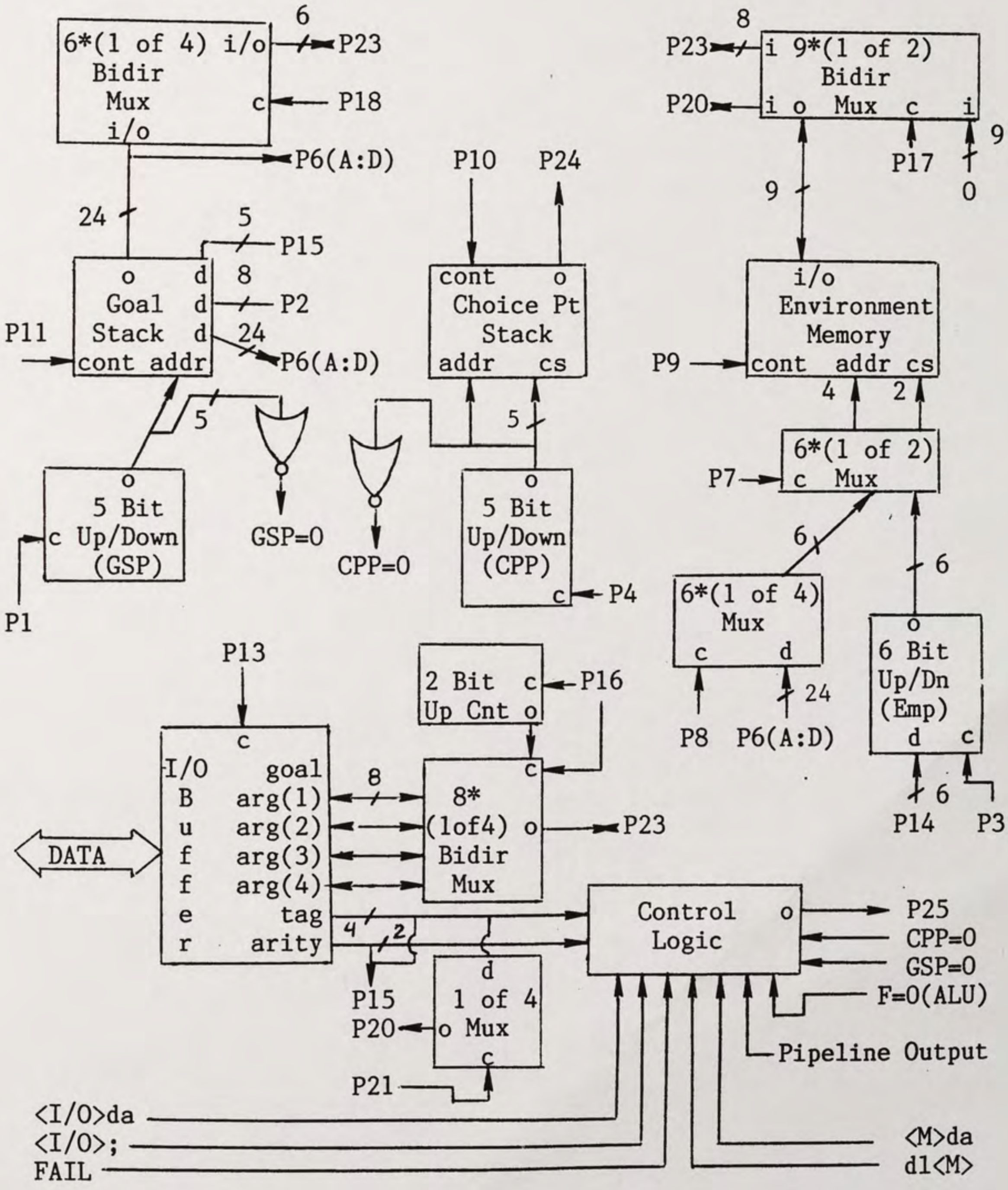
```
1550 '----------SUCCESS PORTION OUTPUTS NEW BINDINGS TO OPERATOR
1560 '
1570 OOT = 0
1580 FOR KK = 1 TO 4
1590 IF KEEPER$(KK) = "" GOTO 1620
1600 IF ASC(LEFT$(KEEPER$(KK),1)) < 97 THEN PRINT KEEPER$(KK)"="EF$(KK):
OOT=1
1610 NEXT KK
1620 IF OOT = 0 THEN PRINT "TRUE"              '---NO BINDINGS, T/F QUERY
1630 GOOD=0:GOTO 230            '---GET A NEW QUERY OR FORCED BACKTRACKING
1640 '
1650 '----------THIS AREA FINDS THE CORRECT LOCATION IN FILE FOR A GOAL
1660 '
1670 PLUG = HASH+24                   '---SET LIMIT OF SEARCH WITHIN REASON
1680 IF PLUG > 100 THEN PLUG = PLUG - 100                  '---CIRCULAR FILE
1690 IF MATRIX$(HASH,1,1) = "" GOTO 1810               '---EMPTY SPOT FOUND
1700 IF MATRIX$(HASH,1,1) = RULE$ GOTO 1750'---COLLISION, CHECK VALIDITY
1710 HASH = HASH + 1: IF HASH = 101 THEN HASH = 0
1720 IF HASH = PLUG THEN PRINT "MEMORY FULL, ENTRY DISALLOWED": GOTO 230
1730 GOTO 1690
1740 '
1750 IF TIST = 1 GOTO 1710
1760 PRINT "THIS FUNCTOR ALREADY EXISTS..."
1770 INPUT "REDUNDANT ENTRY(R), WRITEOVER(W) OR ABORT(touch enter)
DESIRED?  "; RR$
1780 IF RR$ = "R" OR RR$ = "r" THEN TIST = 1:GOTO 1710 '---LOOK FOR SPOT
1790 IF RR$ = "" THEN RETURN
1800 '-----LOAD FUNCTOR HERE, THEN GOSUB TO LOAD ARGUMENTS
1810 MATRIX$(HASH,1,1) = RULE$
1820 SUB2=1: WAL1=INSTR(KAY$,":=")
1830 IF WAL1 = 0 THEN WAL1 = INSTR(WAL2,KAY$,")"): K=1  '---MUST BE FACT
1840 GOSUB 1890
1850 RETURN
1860 '
1870 '----------ARGUMENTS PORTION SCANS INPUT AND LOADS ARGS TO FILE----
1880 '
1890 OOT = 0
1900 FOR TT = 2 TO 5
1910 VAL2 = INSTR(WAL2+1,KAY$,"/")
1920 IF VAL2 >= WAL1 OR VAL2 = 0 THEN VAL2 = INSTR(WAL2,KAY$,")"): OOT=1
1930 MATRIX$(HASH,SUB2,TT) = MID$(KAY$,WAL2+1,VAL2-WAL2-1)
1940 WAL2 = VAL2
1950 IF OOT = 1 GOTO 1990
1960 NEXT TT
1970 PRINT "TOO MANY ARGUMENTS - ENTER AGAIN"
1980 MATRIX$(HASH,1,1) = "": RR$ = "": RETURN
1990 IF TT = 5 THEN A1 = 2: A2 = 1 ELSE A1 = 1: A2 = TT + 1
2000 RETURN
2010 '
2020 '----------THIS SUBROUTINE LOADS GOAL STACK WITH INITIAL QUERY
2030 '
```

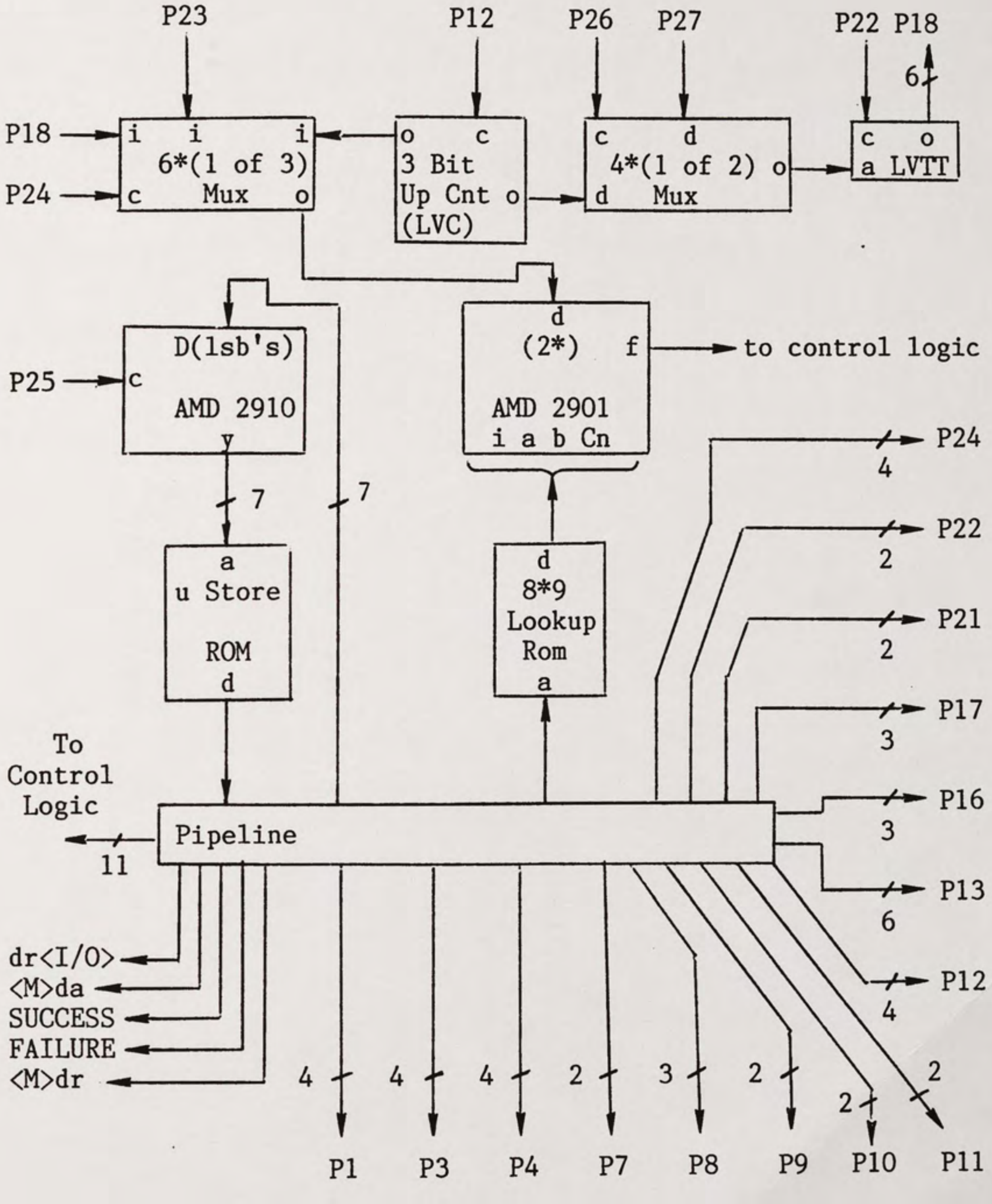
```
2040 OOT1 = 0
2050 VAL1 = INSTR(WAL1,KAY$,"(")
2060 GS$(GSP) = LEFT$(KAY$,VAL1-1)      '---LOAD FUNCTOR ONTO GOAL STACK
2070 FOR TT = 1 TO 4
2080 VAL2 = INSTR(VAL1+1,KAY$,"/")
2090 IF VAL2 = 0 THEN VAL2 = INSTR(VAL1,KAY$,")"):OOT1=1
2100 GS1(GSP,TT) = TT
2110 GOSUB 1130                                  '---NOW LOAD ARGUMENTS
2120 VAL1 = VAL2
2130 IF OOT1 = 1 GOTO 2170                        '---NO MORE ARGUMENTS
2140 EFP = EFP + 1
2150 NEXT TT                    '---CYCLE BACK TO SCAN FOR MORE ARGUMENTS
2160 OOT1 = 0
2170 FOR LL = TT+1 TO 4
2180 GS1(GSP,LL) = 0
2190 KEEPER$(LL) = ""
2200 NEXT LL                     '---CLEAR REMAINING GOAL STACK LOCATIONS
2210 RETURN
2220 '
2230 '----------THIS SUBROUTINE CLEARS UNUSED STORAGE LOCATIONS IN FILE
2240 '
2250 FOR LL A1 TO 5
2260    FOR KK = A2 TO 5
2270    MATRIX$(HASH,LL,KK) = ""
2280    NEXT KK
2290 NEXT LL
2300 RETURN
2310 '
2320 '----------HASHING SCHEME FOR APPROXIMATE LOCATION OF GOAL--------
2330 '
2340 HASH = INT(3.8*(ASC(LEFT$(RULE$,1))-96))
2350 RETURN
2360 '
2370 '----------THIS SUBROUTINE SEARCHES FOR EXACT FUNCTOR MATCH IN FILE
2380 '
2390 FAIL = 0
2400 PLUG = HASH+24: IF PLUG > 100 THEN PLUG = PLUG - 100
2410 IF MATRIX$(HASH,1,1) = GS$(GSP) GOTO 2460
2420 HASH = HASH+1: IF HASH > 100 THEN HASH = HASH - 100
2430 IF HASH = PLUG THEN FAIL = 1: RETURN
2440 GOTO 2410
2450'---CONFIRM ARITY MATCH HERE
2460 CNT = 1
2470 WHILE MATRIX$(HASH,1,CNT+1) <> ""
2480 CNT = CNT+1
2490 WEND
2500 CNT = CNT-1
2510 IF CNT <> ARITY GOTO 2420          '---FAILED ARITY TEST, TRY AGAIN
2520 '---IF FLOW REACHES HERE, DEFINITELY HAVE A MATCH - PROCESS IT
2530 RETURN
2540 '
```

```
2550 '-----------THIS SUB LOADS EF AFTER SUCCESSFUL GOAL SEARCH---------
2560 '
2570 CPP = CPP+1
2580 FOR TT = 1 TO 4
2590 CP2(CPP,TT) = GS1(GSP,TT)
2600 NEXT TT
2610 CP2(CPP,5) = GSP
2620 CP(CPP) = HASH          '---ALSO DON'T FORGET FILE LOCATION OF MATCH
2630 RETURN
2640 '
2650 '-----------THIS SUBROUTINE LOADS TEMPORARY BUFFER FOR CURRENT GOAL
2660 '
2670 FOR AAA = 1 TO 4
2680 GB(AAA) = GS1(GSP,AAA):CP1(CPP,AAA) = 0
2690 NEXT AAA
2700 RETURN
2710 '
2720 '----------THIS PORTION OVERSEES LOADING OF RULES & FACTS INPUTTED
2730 '
2740 WAL2 = INSTR(KAY$,"(")
2750 RULE$ = LEFT$(KAY$,WAL2-1)
2760 GOSUB 2340                  '---HASHING FOR APPROX FILE LOCATION
2770 GOSUB 1670                  '---NOW GET A CLEAR POSITION IN FILE
2780 IF RR$ = "" GOTO 230          '---NO LOCATIONS AVAILABLE - ABORT
2790 GOSUB 2250                   '---CLEAR OUT REMAINING FILE PLACES
2800 IF K = 1 GOTO 230                             '---AWAIT NEW INPUT
2810 '
2820 '----------THIS PORTION OVERSEES LOADING OF RULES-----------------
2830 '
2840 FOR SUB2 = 2 TO 5
2850 WAL1 = WAL1+2
2860 WAL2 = INSTR(WAL1,KAY$,"(")
2870 MATRIX$(HASH,SUB2,1) = MID$(KAY$,WAL1,WAL2-WAL1)'---SUBHEAD LOADED
2880 WAL1 = INSTR(WAL2,KAY$,")")
2890 GOSUB 1890                  '---NOW LOAD ARGUMENTS OF SUBHEADING
2900 IF WAL1 = LEN(KAY$)-1 GOTO 230                '---END OF INPUT STRING
2910 NEXT SUB2
2920 A1=1: A2=1: GOSUB 2250      '---IF FALL THROUGH, TOO MANY SUBGOALS
2930 PRINT ":TOO MANY SUBGOALS - ENTER AGAIN"
2940 GOTO 230
```

APPENDIX B

SKETCH OF FACILITIES (CPU)




P23
P12
P26
P27
P22
P18
6
P18
P24
i i i
6*(1 of 3)
c Mux o
o c
3 Bit
Up Cnt o
(LVC)
c d
4*(1 of 2) o
d Mux
c o
a LVTT
d
(2*) f
AMD 2901
i a b Cn
to control logic
D(lsb's)
P25
c
AMD 2910
y
7
7
a
u Store
ROM
d
d
8*9
Lookup
Rom
a
To
Control
Logic
Pipeline
11
dr<I/O>
<M>da
SUCCESS
FAILURE
<M>dr
P24
4
P22
2
P21
2
P17
3
P16
3
P13
6
P12
4
4
4
4
2
3
2
2
2
P1
P3
P4
P7
P8
P9
P10
P11

LIST OF REFERENCES

{1} Lemmons, Phil."Japan and the Fifth Generation." Byte, November 1983, pp. 394-401.

{2} Hayes-Roth, Fredrick "The Knowledge-Based Expert System: A Tutorial," IEEE Computer Society (September 1984):11-28.

{3} D'Ambrosio, Bruce "Expert Systems-Myth or Reality?" Byte, January 1985, pp. 275-282.

{4} Thompson, Beverly A. and Thompson, William A. "Inside an Expert System." Byte, April 1985, pp. 315-330.

{5} Warren, David H. Prolog-The Language and its Implementation Compared with Lisp. Edinburgh, Scotland: University of Edinburgh, 1977.

{6} Clocksin, W.F. and Mellish, C.S. Programming in Prolog. New York: Springer-Verlag, 1981.

{7} Despain, Alvin M., and Patt, Yale N. "Aquarius--A High Performance Computing System for Symbolic/Numeric Applications" IEEE Proceedings. October 1984, p. 376.

{8} Dobry, T.P., Patt, Y.N., and Despain, A.M. "Design Decisions Influencing the Microarchitecture for a Prolog Machine," Micro 17 Proceedings. October 1984, p. 87.

{9} Degroot, Doug. Prolog and Knowledge Information Processing: A Tutorial. New York: IBM Research, T.J. Watson Research Center, 1984.

{10} Rigas, H., Booth, T. "Artificial Intelligence Research in Japan," IEEE Computer Society. Sept. 1985.

{11} Barr, A., and Feigenbaum, E.A. The Handbook of Artificial Intelligence, Vol. II. Stanford, Cal.: Stanford University, 1982.

{12} Siewiorek, D.P., Bell, G.C., and Newell, A. Computer Structures: Principles and Examples. New York: McGraw-Hill, Inc., 1982.