SRI International

AN OVERLAPPED PROLOG PROCESSOR

Technical Note 308

October 1983

By: Evan Tick, Summer Student

Artificial Intelligence Center
Computer Science and Technology Division

SRI Project 4776

Client: Digital Equipment Corporation

Open Publication. Release of Information.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

An Overlapped Prolog Processor

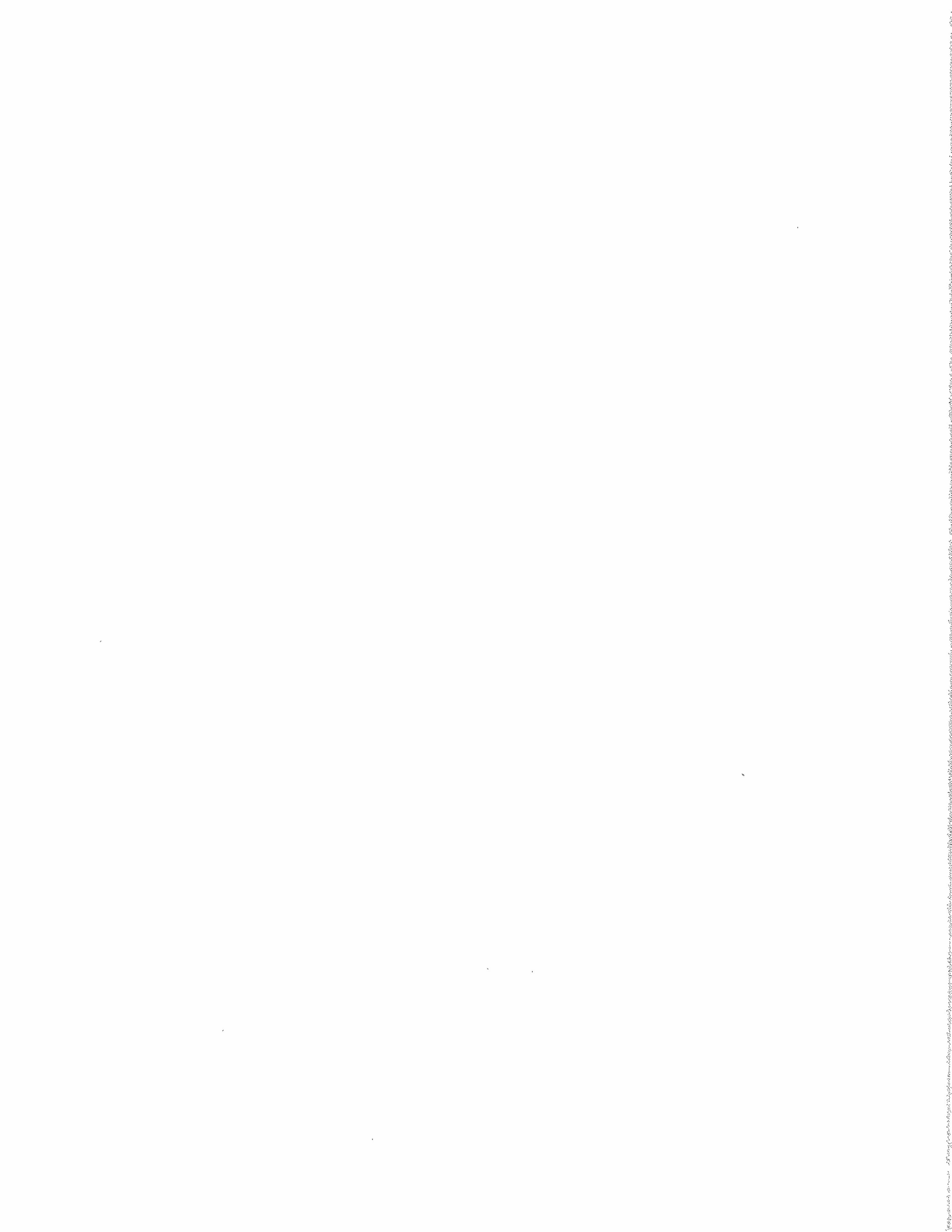
Evan Tick
Artificial Intelligence Center
SRI International

Abstract

This report describes the design of a Prolog machine organization implementing D. Warren's architecture [7]. The objective was to determine the maximum performance attainable by a sequential Prolog machine for "reasonable" cost. The report compares the organization to both general-purpose, microcoded machines and reduced-instruction-set machines. Hand timings indicate that a peak performance rate of 450 K LIPS (logical inferences per second) is well within current technology limitations and 1 M LIPS is potentially feasible.

Table of Contents

1. Introduction 1
2. Organization 3
2.1 Memory 3
2.2 E-Unit 3
2.2.1 Micromachine 4
2.2.2 Tag Definition 5
2.2.3 Datapaths 8
2.3 I-Unit 10
2.4 Summary 12
3. Timing Results 14
3.1 Determinate Concatenate 14
3.2 Nondeterminate Concatenate 16
3.3 Analysis 21
4. Conclusions and Future Work 23
I. Appendix: Instruction Set Microcode 25
II. Appendix: Determinate Concatenate Trace 44
III. Appendix: Nondeterminate Concatenate Trace 46



1. Introduction

Japan's Fifth Generation Computer Systems project [2] aims to build highly parallel logical inference machines with prodigious performance, by exploiting advanced circuit technology, and by pursuing research into non-von Neumann architectures. The target is a performance of 100-1000 M LIPS (logical inferences per second).

To attain such a performance, it will be necessary to exploit large-scale parallelism in logic programs, of which the main kinds are AND parallelism (where several goals in a clause are executed concurrently) and OR parallelism (where several clauses matching a goal are processed concurrently). However it remains to be seen whether practical logic programs have enough large-scale parallelism to enable such ambitious performance targets to be achieved. Certainly, there are important examples of logic programs that do not have any inherent large-scale parallelism, e.g. simple list concatenation.

From a machine design standpoint, the problem is analogous to the classical argument between advocates of vector machines and advocates of fast scalar machines in the numerical computation environment. To attain very high performance, a vector capability is necessary; however, performance is bottlenecked by scalar performance. Similarly, *to attain very high performance in logical inference machines, inherent parallelism must be exploited; however, performance will be bottlenecked by the speed of sequential inference.*

In view of these concerns it is important to investigate the maximum performance that can be achieved by a sequential Prolog machine, where only small-scale parallelism (invisible to the programmer) is exploited. It is also generally agreed that systems relying on radical departures in both hardware and software technology usually achieve less than what is expected. For this reason, conventional pipelining methods are used to achieve high performance.

Although the processor model discussed here is sequential, the architecture is structured to permit exploitation of *unification* parallelism, by allowing implementations with multiple execution units. AND and OR parallelism can also be successfully implemented around this machine model in a tightly coupled multiprocessor system, of say 8 to 16 processors, attaining very high performance.

This report describes the design of a Prolog machine organization implementing D. Warren's architecture [7]. The objective was to determine the maximum performance attainable by a sequential Prolog machine for "reasonable" cost. A compiler is used to produce object programs in a high-level, stack-oriented instruction set. As with most high-level language processors, e.g. ICOT's **PSI** [3] and the Symbolics **3600 Lisp Machine** [5], the organization is centered around a micro-controller because of the complex nature of the instruction set. In this design, the following criteria are stressed:

- *A lean cycle, i.e. the hardware is partitioned to minimize the number of logic levels between latches in the datapath.*

- *Issue one microinstruction per cycle if interlocks allow.*

The cost of expanding the high-level machine instructions into microsequences is offset by overlapping the microinstructions in a pipelined execution unit. Memory accesses are also overlapped by use of an interleaved memory. This allows the optimal use of slow memory, which is more cost-effective for the large physical memories required by symbolic processing applications. Such a consideration is especially important because memory references tend to be more random than in numerical processing, so that caching data is less effective.

The report falls broadly into three parts. The first part describes the machine organization. The second part presents preliminary results in the form of hand timings. In the last part conclusions are drawn and future work is summarized. Refer throughout this report to Warren [7] for details of the Prolog machine architecture.

2. Organization

The model described is a single-user, single-pipeline Prolog processor. The memory system, instruction and execution units (I-Unit and E-Unit(s)) and μcontroller are discussed in this report. Systems issues, e.g. interrupt handling, are not discussed. The model description will set the stage for an answer to the following question (see *Conclusions*):

Instead of designing a special-purpose processor, why not emulate the instruction set on a general-purpose μcoded machine, e.g. Symbolics 3600, or compile it onto a reduced instruction set machine, e.g. IBM 801 [4]?

A summary of the design evolution is given at the end of this chapter. It is helpful in keeping subtle and interrelated design decisions in perspective, but cannot be discussed until terms are defined.

2.1 Memory

The memory model is an interleaved memory with a four cycle access. Because memory accesses are overlapped, access time is not a critical parameter in the processor model. For a single E-Unit, first-come-first-served (FCFS) module queues prevent the possibility of read-write, write-read and write-write races; (extension to multiple E-Units will require a more complex solution).

The model can be extended to include a cache in front of memory or in the I-Unit only. If the locality of heap references is minimal, the cache is better used for instructions only, especially in a multiple E-Unit system.

2.2 E-Unit

The basic datapaths of the E-Unit (*figure 1*) form a three stage pipeline:

- C stage - Array access of the stack buffer, register file, trail buffer and control counters, latching results into the temporary registers (**T**,**T1**) and push-down list (PDL).

- E stage - arithmetic-logic unit (ALU) execution, latching results into the result register (**R**), memory address register (**MAR**) and memory data register (**MDR**).

- **P** stage - Put-away into **C** stage arrays.

2.2.1 Micromachine

Many of the high-level Prolog machine instructions make an arbitrary number of passes through the execution pipe. Controlling such complex sequences while minimizing pipeline breaks is well suited for *data-stationary* μcode [1]. The μcontroller (*figure 2*) function is to supply the execution pipe with μinstructions of the form:

```
| C,E,P control | locks to set | branch control | branch address |
```

Thus each μinstruction contains control information for a single pass through the pipe. A μinstruction is joined with the machine instruction operand to form a *control word*. Control words are latched in a series of *control registers*, one per stage (*figure 3*). At each stage, the control word is checked against *resource locks*. A control word can proceed to the next stage if no required resources are locked and *subsequent* control words can proceed. If the control word cannot proceed, constituting a pipeline break, the result of stage execution is not latched. Resource locks, as indicated in the μinstruction, are initially set when the control word first enters the pipe.

The I-Unit delivers the initial μinstruction address of the μsequence corresponding to each machine instruction. These are queued in the E-Unit. The model assumes distinct μsequences for instructions executed in read and write modes. Because the μinstructions are overlapped, the mode may be selected after subsequent μsequence addresses have been queued. Therefore either the I-Unit must deliver two alternative μaddresses (corresponding to the two modes) from which the E-Unit selects one, or else alternative μsequences are allocated on sufficient boundaries in the μstore to allow concatenating the mode to a single μaddress to form the correct μaddress.

The μstore is a two port read-only memory (ROM) permitting access to the next sequential μinstruction and the target μinstruction indicated by the branch address field of the current μinstruction. The type of μcontrol transfer is indicated by the branch control field. The controller supports μroutine call, return, unconditional branch, conditional branch, dispatch next machine instruction and n-way branch (via a μaddress ROM). The controller can dispatch a new machine instruction every cycle (if the I-Unit can supply them) by virtue of a bypass around the μaddress queue. A conditional

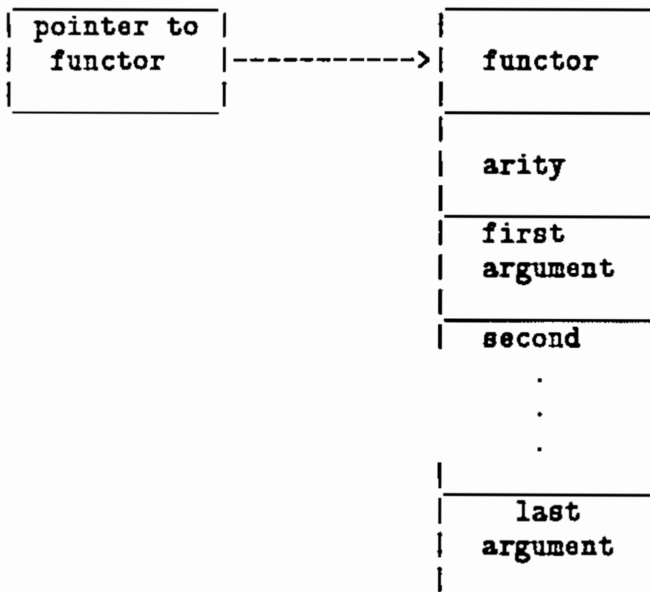
branch can be resolved by a logic signal produced early in the cycle, selecting the correct μinstruction late in the cycle. For branch conditions generated too late in the cycle, e.g., by arithmetic comparison, an extra cycle is taken, keeping the cycle lean.

2.2.2 Tag Definition

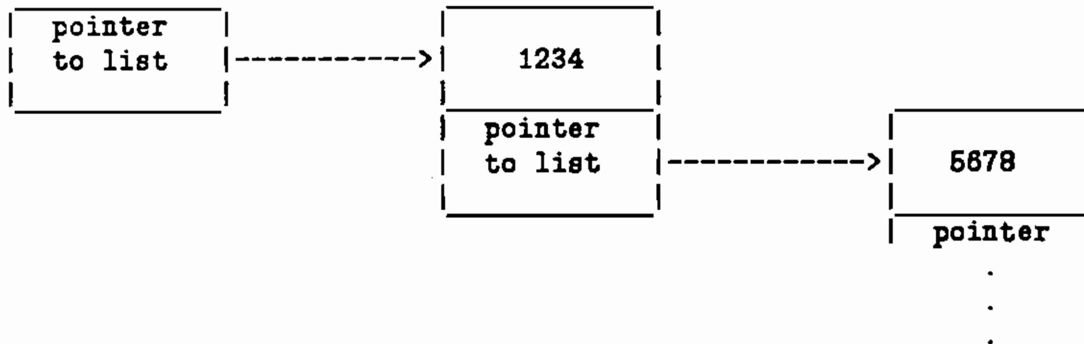
As with other tagged architecture machines, careful consideration must be given to defining an extensive, but not excessive, set of tags. The tag encoding must permit quick decoding for determining object type, a criterion directly related to the critical path of the μcontroller because conditional branches can be resolved by condition codes set by tag decoding. A benefit of the tagged architecture is the ability to introduce hardware type checking in the Prolog engine.

Associated with each object in the machine is a 5-8 bit tag. This simplifies testing data objects in unification and testing procedure objects in clause indexing. Most objects are one word in length (32 bits) not including the tag. Longer objects, e.g., lists, structures, and real numbers, are composed of more than one word. Lists are a special type of structure with an implied arity of two. Structures and lists look like:

STRUCTURE:



LIST:



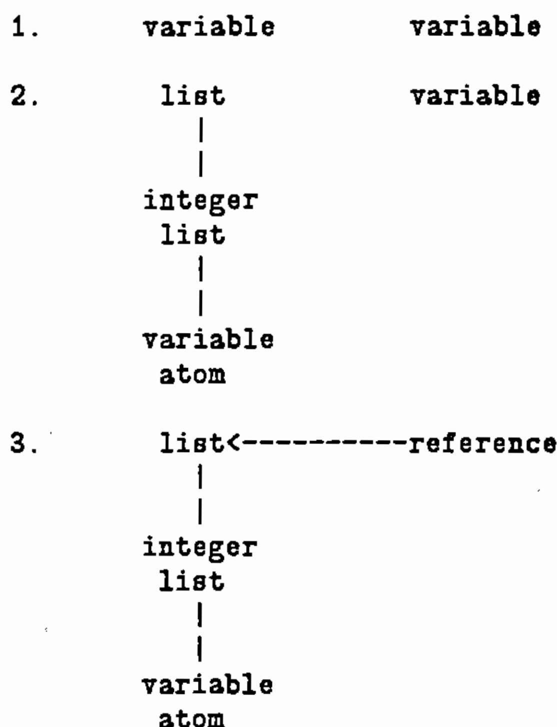
The tags are organized as follows:

tag	type	object contains:
000000	variable	its own address ("variable reference")
011000	reference	address of another object
010000	atom	identifier
010001	integer	integer value
010010	real	exponent value (immediately following is: manissa value)
010011	structure ptr	address of structure
010100	structure	functor identifier (immediately following are: arity (N) first argument second argument . . . Nth argument)
010101	list ptr	address of list
1XXXXX	instruction	code (remaining tag bits are opcode?)

The first tag bit represents data or program. The second tag bit indicates if the object is bound or unbound. The third bit indicates if the object is a reference to another object. The last bits distinguish between nonreference objects. This is confusing because a structure pointer is considered a "nonreference" object, even though it refers to a structure. Reference objects are the stuff with which the unification algorithm glues

together nonreference objects. Thus it makes no sense to have a list-reference object, but a list pointer object is needed.

Consider the following example. State 1 contains two unbound variables. State 2 depicts the binding of the first variable to a list. State 3 depicts the binding of the second variable to the first (bound) variable. The second variable becomes a reference, not a list pointer, otherwise it would represent the wrong structure.



The first three tag bits are kept in decoded form to permit fast testing in hardware. If there are less than 128 opcodes, then the remaining tag bits for instruction objects can be used for the opcodes. This organization also permits the extension of type checking of unbound variables and reference pointers. For instance, list-variables can be created by the compiler with tag=00101. During runtime, binding of such a list-variable to anything other than a list pointer causes an error interrupt. A list-reference with tag=01101 is then needed to bind a list-variable to a list pointer object (as in above example). Other unbound variables to be bound to the same reference chain compare their list-variable tags with the list-reference tag during runtime type checking.

In the dereferencing algorithm (see *Appendix I*), it is essential that the reference and

variable tags are mutually exclusive because only the reference tag is checked. If all variables were also considered references, then dereferencing would loop forever when it hit a leaf variable.

An object, once loaded into a temporary register, exists in two places: that register and the stack where it permanently resides. This means that a variable reference in a temporary register is like a nonvariable reference to a variable (reference). For this reason, whenever an object of tag "variable" is loaded into a temporary register, its tag is changed to "reference." The object in the temporary register will be referred to as "a reference to a variable."

```
    nonreference
    ____________
T: |    123     |                                    simple case
   |____________|                                    atom, integer, etc.


nonvariable reference
    ____________                 ____________
T: |     Y      |------------>  |    123     |       still simple
   |____________|               |____________|       Y is a bound variable

"nonvariable reference"      variable reference
    ____________                 ____________
T: |     X      |------------>  |     X      |--+    confusing
   |____________|               |____________|  |    X is an unbound variable
                                      ^         |    X's tag in T is changed from
                                      |         |    variable to reference.
                                      +---------+
```

2.2.3 Datapaths

The E-Unit datapath includes a stack buffer, general register file, trail buffer and PDL. The trail buffer is used to cache the trail stack segment, and is not strictly necessary. The PDL is used during unification. Both arrays are first-in-last-out stacks which are burst to memory when they fill up. A *multiple E-Unit* organization refers to multiple pipes, each with its own μcontroller and ALU, sharing a single I-Unit, stack buffer and register file.

The register file is modeled as a one input, one output array storing the temporary variables and procedure arguments. Control pointers are implemented in ad hoc

registers and counters. The **B**, **E** and **A** (top-of-stack) pointers are needed for managing the stack. The **S** and **H** pointers are kept in counters, reducing interlocks. The **P** and **CP** pointers require access from both the I-Unit and E-Unit(s).

The stack buffer caches the top of stack in a fast array. The stack holds two types of objects: environments and choice points. Each is arbitrary in length. An environment holds permanent variables, which are directly referenced. A choice point holds state pointers and goal arguments, which require nothing more than sequential referencing, but are accessed directly for design uniformity.

It is stipulated at present that, to avoid thrashing in the stack buffer, current environments not contained in the buffer must be copied onto the top of the stack from memory. This policy increases stack size in an effort to enforce locality of stack references. It also simplifies buffer management because all references are forced to map into the buffer.

Because the stack is a segment in the virtual address space, it is conceivable to reference the stack directly from memory. If a memory cache is needed anyway, e.g., for the heap and program, the stack reference penalty will be reduced. Such an organization does not differ greatly from a standard (scientific/numerical) processor [4]. However, in such a model, general-purpose register allocation puts a burden on the compiler whereas there is no register allocation, per se, for a specialized stack-buffer model. This problem worsens with multiple E-Units, which must lock portions of the stack. Setting and testing locks on word units is less expensive in a sequential stack buffer than in a set-associative cache.

Without a general-purpose cache, a specialized buffer is needed to decrease the stack reference penalty. The stack-buffer design we favor holds a sequential set of locations from the virtual stack segment. The buffer is managed explicitly by the μcontroller. A copy-back policy is instituted, i.e., updates are not immediately reflected in memory. All direct memory references interrogate the buffer and make updates if the virtual address falls between the bounds registers.

Stack references consist of an offset plus a base register. The offset is specified by machine instructions with a value. μinstructions can specify a value or hardware counter (for use when reading and writing choice points). The base register is either the

E or B register. Because of the time critical nature of stack address generation, the number of buffer entries must be kept low. The generated address is guaranteed to fall within the valid buffer range by virtue of the following policy. When the **allocate** and **try** instructions update a base register to become the new top of the stack and that point is within a certain number of buffer entries from the lowest page, a copy-back is initiated. In the current model, a copy-back cannot proceed in parallel with E-Unit operation, i.e., the pipe is broken.

2.3 I-Unit

The primary function of the instruction unit is to supply instructions to the E-Unit(s). The I-Unit also processes certain control instructions directly from the instruction buffers in an effort to reduce the procedure call penalty.

The Prolog machine instruction set has no conditional branches, only procedure calls, which can match one of possibly several clauses. The main design criterion of the I-Unit is to compute clause addresses quickly. To this end, only *indexing on tag* (**switch_on_term**) is optimized. Control instructions detected and executed in the I-Unit are **prefetch** and **prefetch_continuation**. These instructions attempt to prefetch the next clause to be executed into the I-Unit. If the indexing method is not by tag, however, no prefetching is done.

prefetch P is generated by the compiler anywhere before the corresponding **jump** or **invoke** instruction. P is the address of a **switch_on_term** instruction having four operands defining a dispatch table. Each operand, a clause address, corresponds to a different type tag. If the clause has alternatives, they are explicitly defined by **try**, **retry** and **trust** instructions linking the alternatives. The **jump** instruction is generated by the compiler at the clause end, indicating that the instruction streams should be switched, i.e. it is an unconditional branch to a clause determined in the preceding **prefetch**.

The I-Unit services **prefetch** P by using the low-order bits of P to address a small *map cache* (*figure 4*). This cache holds a set of previously seen **switch_on_term** instructions. If the entry key matches P, the tag of register **A1** (holding first argument of a procedure call) is used to select which clause is to be executed.

To allow synchronization between the I-Unit and E-Unit, there is a bit in the μinstruction associated with the **P** stage. The bit indicates that the put-away should also be directed to the I-Unit (this complicates the datapaths given in *figure 1*). The architecture specifies that only an A1 register (holding the functor of the first argument) put-away stage should be used in conjunction with this bit; however the mechanism described is more general and powerful, allowing any **A** register (any argument) to be used for hashing.

The functor is latched into a dedicated register in the I-Unit and the register then becomes valid. This allows the placement of the **prefetch** anywhere in the stream. The I-Unit simply retries the mapping each cycle if the register is not yet valid. Note that this mechanism allows the compiler to use the A1 register to accumulate partial results and then transmit the complete result for hashing.

This scheme complicates the machine architecture, however. Each of several operations used to modify an A register must have two opcodes corresponding to the special μinstruction bit. A compiler must recognize the final put-away of **A1** and use the alternate opcode.

It may be sufficient, in the cases where **A1** is used to accumulate partial results, for the compiler to simply "turn off" this optimization, i.e., not use prefetching. Thus an alternative scheme is to *assume* the compiler will never reload **A1**, and automatically direct the (single) **A1** put-away to the I-Unit. This method requires no extra opcodes.

When the mapping is finally completed and the map entry does not match, the instruction cache is accessed for P. A suitable map cache entry is chosen for replacement by the returning **switch_on_term** instruction. Note the map operates under the principle that clause addresses exhibit locality, i.e. computation in a given time window repeatedly executes the same set of clauses. This is reasonable because of recursive looping. The map buffer can be considered a dynamic collection of "road maps" for various procedure invocations.

There are two interchangeable instruction buffers (FCFS queues) in the I-Unit. With each is associated a program counter. At any given time, one is marked "current" and the other "future." The eventual clause address produced by the **prefetch** mapping is latched into the future program counter. This counter contends with the current

counter (and E-Unit(s)) for cache cycles, in an effort to fill up the future instruction buffer.

The `jump` instruction, also executed in the I-Unit, switches the buffers. In clauses requiring even moderate unification of procedure arguments, the I-Unit will be able to at least partially fill the future instruction buffer beneath normal E-Unit operation, if the cache holds the instructions. This gives a one cycle delay, needed to recognize the `jump` and switch I-buffers.

`prefetch_continuation` is similar to `prefetch`, but is used in unit clauses and has no operand. The `CP` register holds the address of a `switch_on_term` instruction. The `invoke` instruction is similar to `jump`, but stores the continuation address in the `CP` register. Both are executed in the I-Unit.

2.4 Summary

The machine design is the result of several iterations of both architecture and organization redefinition. The original architecture (goal-stacking) defined a "zero address" instruction set, i.e., an instruction got its operand (implicitly) from the stack. An organization model was developed, involving a stack buffer, and timings were done.

This architecture evolved into the environment stacking architecture, using a "single address" instruction set described in Warren [7]. The operand is specified explicitly in the instruction by a base register and offset, although address generation is more limited than in a conventional machine. The idea was to directly address the current environment on the stack. This permits multiple E-Units to operate concurrently with less interlocking than if all operands were implicit.

The organization model developed for the environment-stacking architecture gave timings similar to those of the goal-stacking architecture. This was due to model assumptions that the stack buffer could be accessed either explicitly or implicitly in one cycle.

The machine organization also evolved. Portions of the design are optional, such as the trail buffer and PDL. Removal of this hardware would involve relatively minor μcode modification. A major change involves increasing the number of temporary registers to three, by adding a `T0` register (placed between the {`E`, `B`, `HB`, `TR`} registers

and the ALU). This register speeds up certain instructions (e.g., **get_list**) at the expense of increased complexity and cost of the datapaths.

The next evolutionary step for the organization is to go to multiple E-Unit pipelines. Each such pipeline contains a private ALU,PDL and {**T**, **T1**, **MDR**, **MAR**, **R**, **S**} registers. All the pipelines share a common stack buffer, register file, and control registers (it may be necessary to increase the number of ports in each array). Each pipeline has it own μcontroller. The I-Unit issues *groups* of machine instructions to each pipe. A group contains the code unifying a single term or argument of the clause head or a goal in its body. Development of a multiple E-Unit model is a topic for future research (see *Conclusions*).

3. Timing Results

Using the hardware model described, *approximate* μcode translations were written for the machine instruction set. Two simple Prolog programs were expanded from machine instructions to μcode traces. This chapter presents hand timings of those traces.

3.1 Determinate Concatenate

The first program trace is the determinate execution of *concatenate*:

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

Recall from [7] that **concatenate** translates into the following machine instructions:

```
concatenate/3: switch_on_term(C1a,C1,C2,fail)

C1a:    try_me_else C2a
C1:     get_nil A1
        get_value A2,A3
        proceed

C2a:    trust_me_else fail
C2:     get_list A1
        unify_variable X4
        unify_variable A1
        get_list A3
        unify_value X4
        unify_variable A3
        execute concatenate/3
```

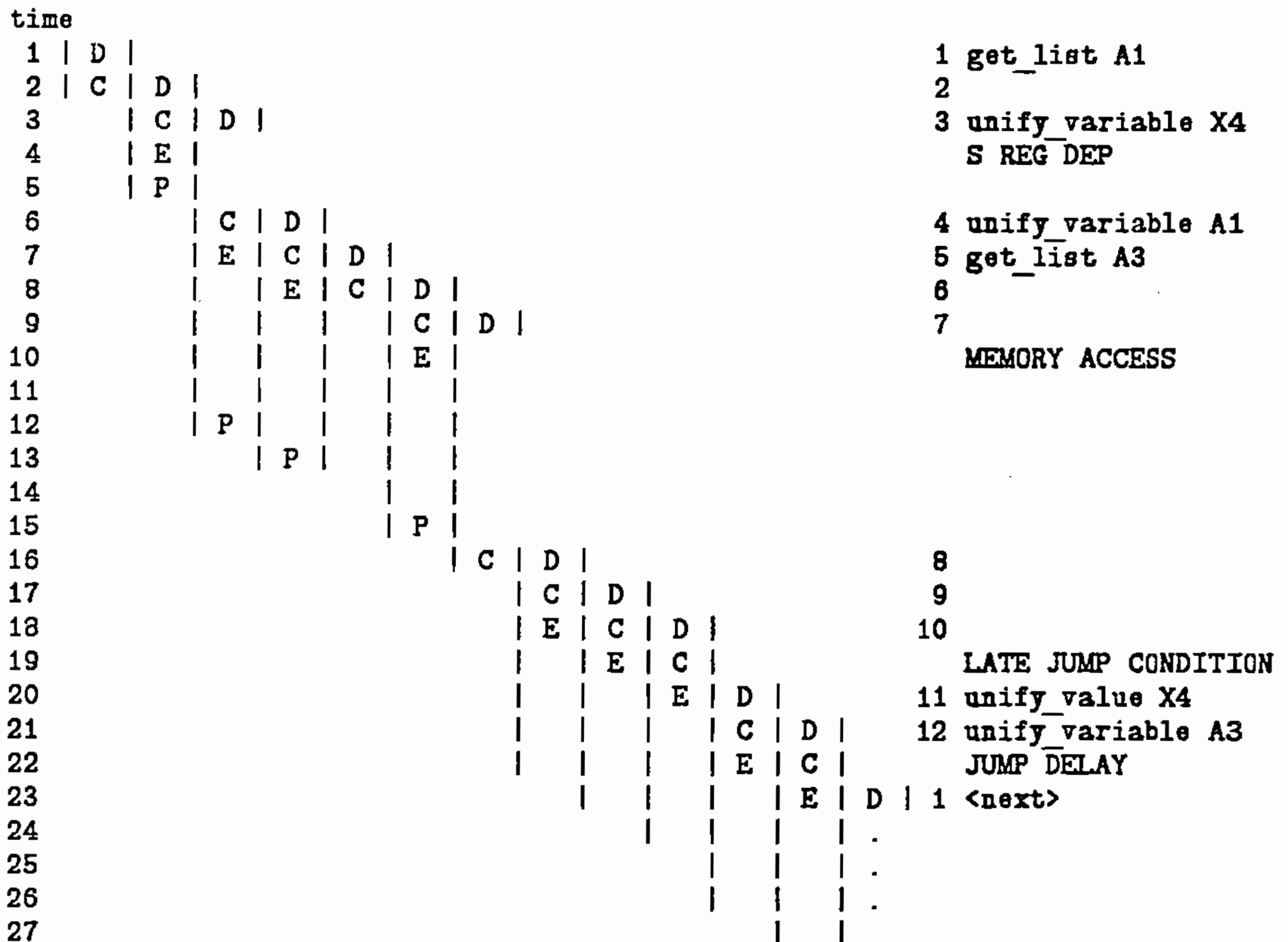
For instance, "`?- concatenate([a,b],[c,d,e],X).`" instantiates `X` to `[a,b,c,d,e]`. The timing considered here is based on the execution of "`?- concatenate([a,b,c,...],[z],X).`" To execute this goal, the following machine instructions are executed repeatedly.

```
prefetch         concatenate/3
get_list         A1
unify_variable   X4
unify_variable   A1
get_list         A3
unify_value      X4
unify_variable   A3
```

jump

The second clause is always immediately chosen, i.e., no unification is attempted with the first clause, by virtue of indexing. The 8 machine instructions are dynamically expanded into 12 μinstructions. The μcode trace is given in *Appendix II.*

The timing diagram below has time, in machine cycles, running vertically and pipe stages running horizontally. The decode (**D**) stages, representing the μstore access, are annotated with the μinstruction number. The start of a μsequence is labeled with the corresponding machine instruction. Memory references are denoted by blank stages extending beyond the execution stage. Memory reads are followed four cycles later by a put-away stage.



It is assumed that the I-Unit can supply an aligned instruction to the E-Unit each cycle within the context of a single clause, i.e., instruction boundary problems are ignored. It

is assumed that the heap is not cached. **jump** and **prefetch** are not shown because they are removed from the instruction stream by the I-Unit. The timing indicates 8 instructions execute in 22 cycles, giving 2.8 cycles per instruction. If one procedure call is considered a single logical inference, the performance is 22 *cycles per logical inference* (CPLI).

The following table summarizes the frequency and total penalty of each type of break in determinate concatenate:

	# occurrences	penalty (cycles)	% total
control reg. dependency	1 A	2	20
memory read	1 B	6	60
microcontrol	1 C	1	10
macrocontrol	1 D	1	10
		10	100

Microcontrol refers to delayed branch resolution due to arithmetic compares. Macrocontrol refers to jump delay. The control register dependency was due to an interlock on the S pointer.

3.2 Nondeterminate Concatenate

The second program trace is a list substring search requiring the nondeterminate execution of *concatenate*:

```
substring(Sub,Bef,Aft,Str) :- concatenate(Bef,Int,Str),
                              concatenate(Sub,Aft,Int).
```

For instance, "`?- substring([a,b],X,Y,[d,a,a,b,c]).`" instantiates X to [d,a] and Y to [c], indicating the strings delimiting the substring [a,b]. The timing considered here is based on the execution of "`?- substring([z],X,Y,[a,b,c,...]).`" which repeatedly fails. The machine instructions executed in each iteration are listed below:

```
<succeeding unit clause>
prefetch_continuation   prefetch next clause
try_me_else C1          create choice point
get_nil A1              1 level of indirection on dereferencing A1
                        bind A1 (variable) to nil
                        trail binding
```

```
get_value A2,A3          unify A3 (list pointer) to A2 (variable)
                         1 level of indirection on dereferencing A2
                         0 levels of indirection on dereferencing A3
                         trail binding
jump
```

```
<failing recursive clause>
prefetch concatenate/3
get_list A1
unify_variable X4
unify_variable A1
get_list A3
unify_value X4           unify bound to bound - fails, backtrack
                         detrail 2 bindings
```

```
<succeeding recursive clause>
prefetch  concatenate/3
get_list  A1             1 level of indirection on dereferencing A1
                         bind A1 (variable) to newly created list pointer
                         trail binding
unify_variable X4
unify_variable X1
get_list A3
unify_value X4           unify X4 (variable) to bound argument
                         1 level of indirection on dereferencing X4
                         no trailing
unify_variable A3
jump
```

The timing was done with assumptions similar to determinate *concatenate*. The failure of **unify_value X4** causes the top of stack to be reset to the last choice point. The 19 instructions expand into approximately 89 μinstructions. The μcode is given in *Appendix III*. The timing follows.

```
time
 1 | D |                                                    1 try_me_else
 2 | C | D |                                                2
 3 | E | .                                                    TEST FOR COPY-BACK
 4 | P | .
 5       .
 6         | D |                                              3
 7         | C | D |                                          4
 8         | E | C |                                            LATE JUMP CONDITION
 9             | E | D |                                      5
10             | P | C | D |                                  6
```

```
11                 | E | C |                           LATE JUMP CONDITION
12                     | E | D |                     7
13                     | P | C | D |                 8
14                         | E | C |                   LATE JUMP CONDITION
15                             | E | D |             9
16                             | P | C | D |         10
17                                 | E | C | D |     11
18                                 | P | E | C | D | 12
19 | D |                               | P | E | C | 13
20 | C | D |                               | P | E | 14
21 | E | C | D |                               | P | 15 get_nil A1
22 | P | E | C | D |                                 16
23     | P |   | C | D |                             17
24             | E |                                   MEMORY ACCESS
25             |   |
26             |   |
27             |   |
28             |   |
29             | P |
30                 | C | D |                         18
31                     | C | D |                     19
32                     | E | C | D |                 20
33                         | E | C |                   LATE JUMP CONDITION
34                             | E | D |             21
35                             |   | C | D |         22 get_val A2,A3
36                             |   | E | C | D |     23
37                             |   | P |   | C | D | 24
38                             |   |           | C |   DISPATCH VIA ROM
39 | D |                                             25
40 | C | D |                                         26
41 | E |                                               MEMORY ACCESS
42 |   |
43 |   |
44 |   |
45 |   |
46 | P |
47     | C | D |                                     27
48         | C |                                       DISPATCH ON ROM
49             | D |                                 28
50             | C | D |                             29
51             | E | C | D |                         30
52             |   | E | C |                           LATE JUMP CONDITION
53             |   |   | E | D |                     31
54             |   |       | C | D |                 32
55             |   |           | C | D |             33
56                             | E | C |               JUMP DELAY
----------------------------------------------------------------------
 1 | D |                       | P | E |               1 get_list A1
```

```
 2 | C | D |                       | P |                 2
 3     | C | D |                                         3 unify_var X4
 4     | E |                                               S REG DEP
 5     | P |
 6         | C | D |                                     4 unify_var A1
 7         | E | C | D |                                 5 get_list A3
 8         |   | E | C | D |                             6
 9         |   |   |   | C | D |                         7 unify_val X4
10         |   |   |   | E |                               S REG DEP
11         |   |   |   | P |
12         | P |   |       | C | D |                     8
13             | P |       | E |                           MEMORY ACCESS
14                         |   |
15                         |   |
16                         |   |
17                         |   |
18                         | P |
19                             | C |                       DISPATCH VIA ROM
20                                 | D |                 9
21                                 | C | D |            10
22                                       .                 FAILURE TESTS
23                                       .
```

```
 1 | D |                                                 1
 2 | C | D |                                             2
 3 | E | C | D |                                         3
 4 | P | E | C | D |                                     4
 5     | P | E | C | D |                                 5
 6         | P |   | C | D |                             6
 7                 | E | C |                               LATE JUMP CONDITION
 8                     | E | D |                         7
 9                         | C | D |                     8
10                         | E | C | D |                 9
11                         |   | E | C |                   LATE JUMP CONDITION
12                         |   |   | E | D |            10
13                         |   |       | C | D |        11
14                         |   |       | E | C | D |    12
15                                     |   | E | C |      LATE JUMP CONDITION
16 | D |                               |   |   | E |    13
17 | C | D |                           |   |            14
18 | E | C | D |                       |   |            15
19 | P | E | C | D |                                    16
20     | P | E | C | D |                                17
21         | P |   | C | D |                            18
22                 | E | C |                              LATE JUMP CONDITION
23                     | E | D |                        19
24                         | C | D |                    20
25                         | E | C |                      LATE JUMP CONDITION
```

```
26                             | E | D |              21
27                                 | C | D |          22
28                                 | E | C |             LATE JUMP CONDITION
29                                     | E | D |      23
```

```
 1 | D |                                   | C |       1 get_list A1
 2 | C | D |                                           2
 3     | C | D |                                       3
 4     | E |                                             MEMORY ACCESS
 5     |   |
 6     |   |
 7     |   |
 8     |   |
 9     | P |
10         | C | D |                                   4
11             | C | D |                               5
12             | E | C | D |                           6
13                 | E | C |                             LATE JUMP CONDITION
14                     | E | D |                       7
15                     |   | C | D |                   8 copy_var X4
16                     |   | E | C | D |               9 copy_var X1
17                     |   | P | E | C | D |          10 get_list A3
18                     |   |   | P | E | C | D |      11
19                             |   | P |   | C | D |  12 unify_val X4
20                             |   |   |   | E |         S REG DEP
21                             |   |   |   | P |
22 | D |                           |   |       | C |  13
23                                             | E |     MEMORY ACCESS
24                                             |   |
25                                             |   |
26                                             |   |
27                                             |   |
28                                             | P |
29 | C |                                                 DISPATCH VIA ROM
30     | D |                                          14
31     | C | D |                                      15
32     | E |                                             MEMORY ACCESS
33     |   |
34     |   |
35     |   |
36     |   |
37     | P |
38         | C | D |                                  16
39             | C |                                     DISPATCH VIA ROM
40                 | D |                              17
41                 | C | D |                          18
42                 | E | C | D |                      19
43                 |   | E | C |                         LATE JUMP CONDITION
```

```
44              |    |    | E | D |                        20
45              |    |        | C | D |                    21
46              |    |        | E | C |                       LATE JUMP CONDITION
47                                | E | D |                22
48                                    | C | D |            23 unify_var A3
49                                    | E | C |               JUMP DELAY
--------------------------------------------------------------------------------
50                                    | P | E | D |           <next>
51                                        |   | .
52                                        |   | .
53                                        |   | .
54                                        |   |
55                                        | P |
```

The trace executes in 165 cycles, giving 8.7 cycles per instruction and 55 CPLI (assuming each iteration corresponds to 3 logical inferences). The pipe breaks are summarized as follows.

	# occurrences	penalty (cycles)	% total
memory read	6	36	49
microcontrol			
late branch condition	14	14	19
dispatch via ROM	5	5	7
macrocontrol			
procedure call	2	2	3
stack tests		10	14
register dependency	3	6	8
		73	100

ROM dispatches are used by unification μcode for quick n-way branches dependent on the tags of the two terms unified. The *procedure call delay* assumes the prefetching mechanism hid most of the penalty. *Stack tests* refer to stack bounds checks when reading and writing a choice point. In this example, the choice point was always found in the stack buffer by virtue of tail recursion optimization.

3.3 Analysis

Assuming a 100ns cycle time for the model, which seems feasible using circuit technology equivalent to the Symbolics 3600, determinate concatenate runs at 450 K LIPS and nondeterminate concatenate runs at 180 K LIPS. To put these results in perspective,

- A firmware implementation of the Prolog instruction set on the Symbolics 3600 is estimated to run determinate concatenate at 110 K LIPS.

- Determinate concatenate compiled by DEC-10 Prolog compiler [6], runs on DEC-2060 at 40 K LIPS.

- PSI performance is predicted to be 30 K LIPS [3].

- On the basis of a prototype implementation it is estimated that a macrocode emulation of the Prolog instruction set on the VAX/780 would run determinate concatenate at 15 K LIPS.

For determinate concatenate with the heap referenced through a one cycle cache with bypass (and 100% hit ratio), 4 cycles are saved. Compiler optimization can prevent the interlock on S, saving 2 cycles. These modifications combined give the performance of 2.0 cycles/instruction and 16 CPLI, a 27% speed improvement.

For nondeterminate concatenate, a cached heap would, at best, save 24 cycles. Removing S interlocks saves 6 cycles. These modifications combined give the performance of 7.1 cycles/instruction and 45 CPLI, an 18% speed improvement.

4. Conclusions and Future Work

The work completed to date indicates that a sequential Prolog machine with significant performance can be built using conventional design principles for pipelined processors. Assuming reasonable technology, the timing results show the model runs significantly faster than all current or near-future implementations of Prolog. Far more importantly, it is felt the sequential pipelined machine will provide the best cost/performance ratio in the just now emerging high-end environment of logical inference processors and this cost/performance advantage will extend to implementations with multiple E-Units.

There appear to be several reasons why the pipelined Prolog processor can significantly out-perform a Symbolics 3600 μcode implementation. The lean cycle of the Prolog processor permits greater overlapping than the partially overlapped 3600 "fat" cycle. Given an equivalent technology, the model has a cycle time of less than half the 3600. Compared to the 3600, memory accesses are more highly overlapped, allowing a slow memory with less performance degradation. In addition, the specialized hardware support for procedure call, indexing and unification dispatching enhances Prolog performance. It should be borne in mind, however, that the present processor design is probably substantially more complex (and costly) than the 3600.

Justifying any advantage over reduced instruction set machines is more difficult. The microarchitecture of the Prolog processor is primitive and permits more parallelism, on the datapath level, than a conventional machine. The object code is more compact, making memory caching more effective. There are no conditional branches in the machine instruction set, only on the μinstruction level, permitting branch target prefetch and "late select." The Prolog macroinstruction prefetch unit is expected to be more efficient than a conventional prefetch unit, which must change context more frequently. Were the instruction set compiled into primitive instructions, *many* conditional branches and subroutine calls would be generated (if only to keep the object program to a reasonable length). Although the dynamic translation of machine instructions into μsequences has a large latency, it is usually hidden when calling procedures by executing certain procedural instructions in the I-Unit concurrently with E-Unit operation. Disadvantages of the Prolog machine include the effectiveness of a directly accessible stack buffer, which is unproved. In addition, the impact of a large μinstruction on hardware cost (and speed) has not been assessed.

Related design problems concern designing a better stack buffer and defining multiple E-Unit operation.

The fundamental problem with nondeterminate computation is the burden of saving the complete program history at each choice point. Currently, the stack holds both active environments and backtrack information, consisting of choice points and inactive environments. The creation of a choice point "freezes" all objects below it on the stack because resumption of that choice point must reinstate the machine exactly. This lessens the locality of the stack, i.e., current environments may lie deep within the stack, resulting in degraded stack performance. A primary concern is to increase the stack locality of the computation. This is currently achieved by copying the current environment to the top of stack. Two other ideas are being entertained:

- Split the stack into a choice-point stack and environment stack.
- Split the stack into two windows, holding the current choice point and the current environment. Note that one of these objects must be at the top of the stack.

Additional E-Unit pipes introduce new problems as well as aggravating old ones. Most critical is the instruction bandwidth produced by the I-Unit. This will limit the number of pipes, beyond which performance is no longer cost effective. In addition, memory requests from different pipes can cause races. These must be prevented either by careful compiler scheduling of the pipes or dynamic synchronization in hardware. An efficient set of interlocks and hardware locking mechanisms is needed. The combination of these considerations appear to limit the number of pipes from 2-4.

I. Appendix: Instruction Set Microcode

The following μcode *descriptions* are approximate in nature. Sequences for common instructions and abstract operations are given. Instructions not given can often be inferred from similar instructions. Control instructions are not given. Some complex portions of sequences are also missing, replaced with a "?", e.g., stack-buffer management operations.

The descriptions are given in an informal register transfer language, to indicate the pipe stages necessary for each. An example of a μinstruction description follows:

```
LOOP:   C       T = A3
                jump = islist(A3)
        E       R = T + T1
                MAR = T,X5
        P       Y4 = R                  :jump(LOOP)
```

This imaginary instruction loads register **A3** into the **T** register. A flag bit is set from **A3**'s tag (other tags are tested with **isref**, **istrct** and **isvar**). It assumes that a previous μinstruction has loaded the **T1** register. During the execution stage, the **T** and **T1** registers are added and the sum is latched into the **R** register. A memory read is initiated by loading the virtual address in **T** and the put-away target, register **X4**, into the **MAR** (a memory write is indicated by loading the **MDR**). During the put-away stage, the **R** register is stored into register **Y4**. When the memory read completes, another put-away is performed.

The label and jump indicate control flow. There are two pre-defined labels: DISPATCH, which indicates the first μinstruction of the next macroinstruction and RETURN, which indicates the μinstruction pointed to by the top of the micromachine push down list. There are several methods of transferring control within the μcode: unconditional branch, conditional branch, dispatch next macroinstruction, dispatch on unification table, call and return.

Certain stages can be "doing nothing." This is noted by a "nil" argument. All stages "doing nothing" after the last stage to "do something" are absent in the notation. For example, the following two μinstructions are equivalent:

```
C       nil
E       R = T + T1
P       nil                     : (DISPATCH)

C       nil
E       R = T + T1              : (DISPATCH)
```

There are often alternative choices of μcode for implementing the same operation. Time is traded for space, or time (for frequent operations) for time (for infrequent operations). Alternative datapaths introduce different μcode - here the trade-off is time for cost. Optimizing for speed is tricky because minimizing μcode paths (reducing number of cycles) often adversely effects cycle time by complicating the hardware (increasing the length of critical paths). Relevant alternatives are given so that final design decisions can be made for a given technology.

Basic Operations

Fail

This operation is performed when a failure occurs during unification. It causes backtracking to the most recent choice point. The pointers saved in the choice point are restored in following order:

```
B------>BP        current program pointer
        H         current heap pointer
        TR        current trail pointer
        B         current backtrack pointer (to last choice point)
        BCP       continuation pointer
        BCE       current environment pointer
        n         number of arguments
        {A1,A2,...,An} all of the valid A registers
```

The choice point is essentially discarded by restoring B to the previous value saved in the choice point. The trail is "unwound" as far as the choice point trail pointer, by popping references off the trail and resetting the variables they address to unbound.

```
FAIL:   C     ?              <check if choice point
                              is in buffer or not>

MISS:   C     ?              <purge buffer and reload>

HIT:    C     T = B          <pop stack to B>
        E     R = T
        P     A = R
```

```
        C    T1 = B0                  <get old program pointer>
        E    R = T1
        P    P = R

        C    T1 = B1                  <get old heap pointer>
        E    R = T1
        P    H = R

        C    T = B2                   <get old trail pointer>

LOOP:   C    T1 = TR                  <detrail bindings>
        E    done = T1<T

        C    nil
        E    jump = done              :jump(NEXT)

        C    T1 = top of trail
             TR = TR + 1
        E    MAR = T1
             MDR = T1||unbound tag    :(LOOP)

NEXT:   C    T1 = B3                  <get old backtrack pointer>
        E    R = T1
        P    B = R

        C    T1 = B4                  <get old continuation pointer>
        E    R = T1
        P    CP = R

        C    T1 = B5                  <get old environment pointer>
        E    R = T1
        P    E = R

        C    T = B6                   <get # of goal arguments>
             i = 7

LOOP:   C    T1 = Bi                  <restore A registers>
             i = i + 1
        E    done = T<(n+7)
             R = T1
        P    Ai = R

        C    nil
        E    jump = done              :jump(LOOP)

        C    nil                      :(DISPATCH)
```

Trailing

This operation is performed when a variable reference (in T) is bound. If the variable is in the heap and is before the heap backtrack point HB, or the variable is in the stack and is before the stack backtrack point B, the reference in T is pushed onto the trail. Otherwise, no action is taken.

The code, assuming a trail stack, follows:

```
1            C      T1 = H
             E      local = T > T1

2            C      T1 = HB
             E      trailHB = T > T1
                    jump = local              :jump(LOCAL)

3            C      nil
             E      jump = trailHB            :jump(NEXT)

4            C      nil
             E      R = T
             P      top of trail = R
                    TR = TR - 1               :(NEXT)

5    LOCAL:  C      T1 = B
             E      trailB = T > T1

6            C      nil
             E      jump = trailB             :jump(NEXT)

7            C      nil
             E      R = T
             P      top of trail = R
                    TR = TR - 1

     NEXT:
```

Without a trail stack, the trailing action is done as follows:

```
             C      T1 = TR
             E      MAR = T1
                    MDR = T
                    R = T1 - 1
             P      TR = R
```

Detrailing

This operation is performed during backtracking. The trail is "unwound" as far as the choice point trail pointer (in T), by popping references off the trail and resetting the variables they address to unbound. The following code assumes a trail stack.

```
1       LOOP:   C       T1 = TR
                E       done = T1 < T

2               C       nil
                E       jump = done             :jump(NEXT)

3               C       T1 = top of trail
                        TR = TR + 1
                E       MAR = T1
                        MDR = T1||unbound tag   :(LOOP)

        NEXT:
```

Unification

This operation is performed while popping the arguments of a procedure from the stack. The μinstructions for unify constitute a μroutine. The code assumes a unification dispatch table for calculating the unification case in one cycle. This is a 256 entry μaddress ROM, entered as follows:

microaddr	ROM address refT	refT1	listT	listT1	strctT	strctT1	varT	varT1
2REF	1	1	X	X	X	X	X	X
TREF	1	0	X	X	X	X	X	X
T1REF	0	1	X	X	X	X	X	X
TVAR	0	0	X	X	X	X	1	0
T1VAR	0	0	X	X	X	X	0	1
0BOUND	0	0	X	X	X	X	1	1
2LISTS	0	0	1	1	X	X	0	0
2STRUC	0	0	X	X	1	1	0	0
FAIL	0	0	1	0	X	X	0	0
FAIL	0	0	0	1	X	X	0	0
FAIL	0	0	X	X	1	0	0	0
FAIL	0	0	X	X	0	1	0	0
EQUAL	0	0	0	0	0	0	0	0

The following code assumes the two terms to be unified are in T and T1.

```
1       UNIFY:  C       nil                     :rom
```

```
2REF:    C      nil
         E      MAR = T,T

         C      jump = isref(T)          :jump(2REF)

T1REF:   C      nil
         E      MAR = T1,T1

         C      jump = isref(T1)         :jump(T1REF)

         C      nil                      :rom

TREF:    C      nil
         E      MAR = T,T

         C      jump = isref(T)          :jump(TREF)

         C      nil                      :rom
```

```
EQUAL:   C      nil
         E      jump = T=T1              :jump(RETURN)

         C      nil                      :(FAIL)
```

```
2STRUC:  C      nil
         E      jump = not(both structures of same functor)
                                         :jump(FAIL)

         C      top of PDL = M
                Q = Q + 1
         E      R = T - ?
         P      M = R

         C      top of PDL = S
                Q = Q + 1
         E      R = T - ?
         P      S = R

         C      top of PDL = S1
                Q = Q + 1
         E      R = T1 - ?
         P      S1 = R

LOOP1:   C      T = M
         E      R = T - 1
         P      M = R
```

```
          C       T = S
                  S = S + 1
          E       MAR = T,T

          C       T1 = S1
          E       MAR = T1,T1
                  R = T1 + 1
          P       S1 = R                  :call(UNIFY)

          C       T = M
          E       jump = T<>0             :(LOOP1)

          C       nil
          E       R = top of PDL
                  Q = Q - 1
          P       S1 = R

          C       nil
          E       R = top of PDL
                  Q = Q - 1
          P       S = R

          C       nil
          E       R = top of PDL
                  Q = Q - 1
          P       M = R                   :(RETURN)
```

```
2LISTS:   C       top of PDL = M
                  Q = Q + 1               <Q is top of PDL pointer>
          E       R = 2
          P       M = R

          C       top of PDL = S
                  Q = Q + 1
          E       R = T - 2
          P       S = R

          C       top of PDL = S1
                  Q = Q + 1
          E       R = T1 - 2
          P       S1 = R

LOOP2:    C       T = M
          E       R = T - 1
          P       M = R

          C       T = S
```

```
              S = S + 1
        E     MAR = T,T

        C     T1 = S1
        E     MAR = T1,T1
              R = T1 + 1
        P     S1 = R                     :call(UNIFY)

        C     T = M
        E     jump = T<>0                :jump(LOOP2)

        C     nil
        E     R = top of PDL
              Q = Q - 1
        P     S1 = R

        C     nil
        E     R = top of PDL
              Q = Q - 1
        P     S = R

        C     nil
        E     R = top of PDL
              Q = Q - 1
        P     M = R                      :(RETURN)
```

```
OBOUND: C     nil
        E     local = T1>T

        C     nil
        E     jump = local               :jump(TVAR)

T1VAR:  C     nil
        E     MAR = T1
              MDR = T

        C     T = H
        E     local = T1>T

        C     T = HB
        E     trailHB = T1>T
              jump = local               :jump(LOCAL2)

        C     nil
        E     jump = trailHB             :jump(RETURN)

        C     nil
        E     R = T1
```

```
         P        top of trail = R
                  TR = TR - 1              :(RETURN)

LOCAL2:  C        T = B
         E        trailB = T1>T

         C        nil
         E        jump = trailB            :jump(RETURN)

         C        nil
         E        R = T1
         P        top of trail = R
                  TR = TR - 1              :(RETURN)
```

```
TVAR:    C        nil
         E        MAR = T
                  MDR = T1

         C        T1 = H
         E        local = T>T1

         C        T1 = HB
         E        trailHB = T>T1
                  jump = local             :jump(LOCAL1)

         C        nil
         E        jump = trailHB           :jump(RETURN)

         C        nil
         E        R = T
         P        top of trail = R
                  TR = TR - 1              :(RETURN)

LOCAL1:  C        T1 = B
         E        trailB = T>T1

         C        nil
         E        jump = trailB            :jump(RETURN)

         C        nil
         E        R = T
         P        top of trail = R
                  TR = TR - 1              :(RETURN)
```

Indexing Instructions

`try_me_else n`

This operation is performed when entering a Prolog procedure for which there is more than one potentially matching clause. The following values are put into a choice point object:

{A1,A2,...,An}	all of the valid A registers
n	number of arguments
BCE	current environment pointer
BCP	continuation pointer
B	current backtrack pointer (to last choice point)
TR	current trail pointer
H	current heap pointer
BP	current program pointer

First, the top of stack register, A, is incremented by n+7 (computed by compiler). B is saved, and reset to the top of stack. The current program pointer is the last item to be pushed onto the choice point, because it must be popped as early as possible during the failure sequence to allow prefetching of the code. HB is set to the current heap pointer, and B is set to point to the current top of stack. The implementation of pushing a variable number of A registers onto the stack is rather tricky. A counter, initially specifying the number of valid A registers, is decremented each C cycle and used to access the register bank. The jump condition is set by the counter.

```
1               C       T = B                   <save old B>
                        T1 = A
                        i = 7
                E       R = T1 + (n+7)
                P       B = R
                        A = R

                C       ?                       compare A to Z in hardware
                                                if too close, copy-back
                                                buffer page.

        LOOP:   C       T1 = Xi
                E       R = T1
                P       Bi = R

                C       i = i + 1
                E       jump = i<(n+7)          :jump(LOOP)

                C       T1 = H
```

```
E       R = T1
P       B1 = R
        HB = R

C       nil                     <old B>
E       R = T
P       B2 = R

C       T1 = TR
E       R = T1
P       B3 = R

C       T = E
E       R = T
P       B4 = R

C       T = CP
E       R = T
P       B5 = R

C       T = P
E       R = T
P       B6 = R
```

Put Instructions

put_var_Y n,m

This instruction represents a goal argument that is an unbound variable. The instruction puts a reference to permanent variable **Yn** into register **Am** and initializes **Yn** with the same reference.

```
1               C       T = E||n||unbound tag
                E       R = T
                P       En = R
                        Am = R                  : (DISPATCH)
```

put_val_V n,m

This instruction represents a goal argument that is a bound variable. The instruction puts the value of register **Vn** into register **Am**.

```
1               C       T1 = Vn
                E       R = T1
                P       Am = R                  : (DISPATCH)
```

put_const C,m

This instruction represents a goal argument that is a constant. The instruction puts the constant C into register **Am**.

```
1          C      T = C
           E      R = T
           P      Am = R                    : (DISPATCH)
```

put_struct N,m

This instruction marks the beginning of a structure without substructures occurring as a goal argument. The instruction pushes the functor N for the structure onto the heap, and puts a corresponding structure pointer into register **Am**.

```
1          C      T = N
                  T1 = H
                  H = H + 1
           E      MAR = T1
                  MDR = T
                  R = T1
           P      Am = R                    : (DISPATCH)
```

Get Instructions

get_var_V n,m

This instruction represents a head argument that is an unbound variable. The instruction gets the value of register **Am** and stores it in register **Vn**.

```
1          C      T1 = Am
           E      R = T1
           P      Vn = R                    : (DISPATCH)
```

get_val_X n,m

This instruction represents a head argument that is a bound variable. The instruction gets the value in register **Am** and *unifies* it with the contents of register **Xn**. The final result is left in register **Xn**.

```
1          C      T = Am
```

```
2              C      T1 = Xn                  :call(UNIFY)

3              C      nil
               E      R = T1
               P      Xn = R                   :(DISPATCH)
```

get_val_Y n,m

This instruction represents a head argument that is a bound variable. The instruction gets the value in register **Am** and *unifies* it with the contents of register **Yn**.

```
1              C      T = Am
                      T1 = Yn                  :call(UNIFY)

2              C      nil                      :(DISPATCH)
```

get_const C,m

This instruction represents a head argument that is a constant. The instruction gets the value of register **Am** and dereferences it. If the result is a reference to a variable, that variable is bound to the constant C, and the binding is trailed if necessary. Otherwise, the result is compared with the constant C, and if the two values are not identical, backtracking occurs.

The following code assumes a **T0** register.

```
1              C      T = Am
                      listT = islist(Am)
                      jump = not isref(Am)     :jump(NOTREF)

2      LOOP:   C      T0 = B
               E      MAR = T,T1
                      trailB = T>T0

3              C      jump = isvar(T1)
                      T0 = HB
               E      trailHB = T>T0           :jump(VAR)

4              C      jump = isref(T1)
                      listT = islist(T1)
               E      R = T1
               P      T = R                    :jump(LOOP)
```

```
5       NOTREF: C      T1 = C
                E      jump = T=T1              :jump(DISPATCH)

6               C      nil                      :(FAIL)

7       VAR:    C      T = C
                E      jump = not trailHB and not trailB
                       MAR = T1
                       MDR = T                  :jump(DISPATCH)

8               C      nil                      <assume trail stack>
                E      R = T1
                P      top of trail = R
                       TR = TR - 1              :(DISPATCH)
```

The following code assumes no T0 register.

```
1               C      T = Am
                       listT = islist(Am)
                       jump = not isref(Am)     :jump(NOTREF)

2       LOOP:   C      nil
                E      MAR = T,T1

3               C      jump = isvar(T1)         :jump(VAR)

4               C      jump = isref(T1)
                       listT = islist(T1)
                E      R = T1
                P      T = R                    :jump(LOOP)

5       NOTREF: C      T1 = C
                E      jump = T=T1              :jump(DISPATCH)

6               C      nil                      :(FAIL)

7       VAR:    C      T = B
                E      trailB = T>T1

8               C      T = HB
                E      trailHB = T>T1

9               C      T = C
                E      jump = not trailHB and not trailB
                       MAR = T1
                       MDR = T                  :jump(DISPATCH)

10              C      nil                      <assume trail stack>
```

```
                E       R = T1
                P       top of trail = R
                        TR = TR - 1              :(DISPATCH)
```

get_list m

This instruction marks the beginning of a list without substructures occurring as a head argument. The instruction gets the value of register **Am** and dereferences it. If the result is a reference to a variable, then that variable is bound to a new list pointer pointing at the top of the heap and execution proceeds in "write" mode. Otherwise, if the result is a list, then the pointer **S** is set to point to the arguments of the list and execution proceeds in "read" mode. Otherwise, backtracking occurs.

The following code assumes a **T0** register.

```
1               C       T = Am
                        T1 = Am
                        mode = read
                        list = islist(Am)
                        jump = not isref(Am)     :jump(NOTREF)

2       LOOP:   C       mode = write
                        T0 = B
                E       MAR = T,T1
                        trailB = T>T0

3               C       T0 = HB
                E       trailHB = T>T0
                        jump = isvar(T1)         :jump(VAR)

4               C       jump = isref(T1)
                        list = islist(T1)
                E       R = T1
                P       T = R                    :jump(LOOP)

5       NOTREF: C       mode = read
                        jump = list
                E       R = T1
                P       S = R                    :jump(DISPATCH)

6               C       nil                      :(FAIL)

7       VAR:    C       T = H
                E       jump = not trailHB and not trailB
                        MAR = T1
                        MDR = T                  :jump(DISPATCH)
```

```
8                 C       nil                          <assume trail stack>
                  E       R = T1
                  P       top of trail = R
                          TR = TR - 1                  :(DISPATCH)
```

The following code assumes no T0 register.

```
1                 C       T = Am
                          T1 = Am
                          mode = read
                          list = islist(Am)
                          jump = not isref(Am)         :jump(NOTREF)

2       LOOP:     C       nil
                  E       MAR = T,T1

3                 C       jump = isvar(T1)             :jump(VAR)

4                 C       jump = isref(T1)
                          list = islist(T1)
                  E       R = T1
                  P       T = R                        :jump(LOOP)

5       NOTREF:   C       jump = list
                  E       R = T1
                  P       S = R                        :jump(DISPATCH)

6                 C       nil                          :(FAIL)

7       VAR:      C       T = B
                          mode = write
                  E       notrailB = T>T1

8                 C       T = HB
                  E       notrailHB = T>T1

9                 C       T = H
                  E       jump = notrailHB and notrailB
                          MAR = T1
                          MDR = T                      :jump(DISPATCH)

10                C       nil                          <assume trail stack>
                  E       R = T1
                  P       top of trail = R
                          TR = TR - 1                  :(DISPATCH)
```

Unify Instructions

`unify_var_V n`

This instruction represents a head structure argument that is an unbound variable. It gets the next argument from **S** and stores it in register **Vn**.

```
1        C      T1 = S
                S = S + 1
         E      MAR = T1,Vn              :(DISPATCH)
```

`copy_var_V n`

This instruction represents a head structure argument that is an unbound variable. It pushes a new unbound variable onto the heap, and stores a reference to it in register **Vn**.

```
1        C      T = H
                H = H + 1
         E      MAR = T
                MDR = T||unbound tag
                R = T||reference tag
         P      Vn = R                   :(DISPATCH)
```

`unify_val_X n`

This instruction represents a head structure argument that is a variable bound to some global value. It gets the next argument from **S**, and *unifies* it with the value in register **Xn**, leaving the result in register **Xn**.

```
1        C      T = Xn
                T1 = S
                S = S + 1
         E      MAR = T1,T1              :call(UNIFY)

         C      nil
         E      R = T
         P      Xn = R                   :(DISPATCH)
```

`unify_val_Y n`

This instruction represents a head structure argument that is a variable bound to some global value. It gets the next argument from **S**, and *unifies* it with the value in register **Yn**.

```
1              C      T = Yn
                      T1 = S
                      S = S + 1
               E      MAR = T1,T1             :call(UNIFY)

               C      nil                     :(DISPATCH)
```

`unify_local_val_V n`

See code for `unify_val_V n`.

`copy_local_val_V n`

This instruction represents a head structure argument that is a variable bound to a value that is not necessarily global. It dereferences the value of register **Vn**. If the result is *not* a reference to a variable on the stack, then it pushes the result onto the heap. If the result *is* a reference to a variable on the stack, a new unbound variable is pushed onto the heap, the variable on the stack is bound to a reference to the new variable, the binding is trailed if necessary, and register **Vn** is set to point to the new variable.

```
1              C      T1 = Vn
                      T = H
                      H = H + 1
                      ref = isref(Vn)
                      var = isvar(Vn)
                      jump = var or ref       :jump(LOOP)

2              C      nil                     <simplest case>
               E      MAR = T
                      MDR = T1                :(DISPATCH)

3      LOOP:   C      jump = var              :jump(VAR)

4              C      nil
               E      MAR = T1,T1

5              C      var = isvar(T1)
                      jump = isref(T1)        :jump(LOOP)

6      VAR:    C      nil
               E      local = T1>T

7              C      nil
               E      jump = local & var      :jump(LOCAL)
```

```
8                C     nil
                 E     MAR = T
                       MDR = T1                 :(DISPATCH)

9       LOCAL:   C     nil                      <create unbound heap object>
                 E     MAR = T
                       MDR = T||unbound tag
                 P     Vn = T

10               C     nil                      <bind local variable to object>
                 E     MAR = T1
                       MDR = T

11               C     T = B
                 E     local = T1<T

12               C     nil
                 E     jump = local             :jump(DISPATCH)

13               C     nil                      <assuming a trail stack>
                 E     R = T1
                 P     top of trail = R
                       TR = TR - 1              :(DISPATCH)
```

copy_val_V n

Push value of variable **Vn** onto the heap.

```
1                C     T = Vn
                       T1 = H
                       H = H + 1
                 E     MAR = T1
                       MDR = T                  :(DISPATCH)
```

II. Appendix: Determinate Concatenate Trace

The determinate concatenate μcode trace follows. The second **get_list** switches to write mode. The dereferencing of the argument requires one indirection because, in the steady state, the argument is a reference to a list whose first element is a variable. The **prefetch** and **jump** instructions are executed in I-Unit. No **T0** register was assumed.

```
1             C      T = A1                   <get_list A1>
                     T1 = A1
                     mode = read
                     list = islist(A1)
                     jump = not isref(A1)     :jump(NOTREF)

2   NOTREF:   C      jump = list
              E      R = T1
              P      S = R                    :jump(DISPATCH)
```

```
3             C      T1 = S                   <unify_var X4>
                     S = S + 1
              E      MAR = T1,X4              :(DISPATCH)
```

```
4             C      T1 = S                   <unify_var A1>
                     S = S + 1
              E      MAR = T1,A1              :(DISPATCH)
```

```
5             C      T = A3                   <get_list A3>
                     T1 = A3
                     mode = read
                     list = islist(A3)
                     jump = not isref(A3)     :jump(NOTREF)

6   LOOP:     C      nil
              E      MAR = T,T1

7             C      jump = isvar(T1)         :jump(VAR)

8   VAR:      C      T = B
                     mode = write
              E      notrailB = T>T1

9             C      T = HB
              E      notrailHB = T>T1

10            C      T = H
```

	E	jump = notrailHB and notrailB	
		MAR = T1	
		MDR = T	:jump(DISPATCH)
11	C	T = Vn	<copy_val X4>
		T1 = H	
		H = H + 1	
	E	MAR = T1	
		MDR = T	:(DISPATCH)
12	C	T = H	<copy_var A3>
		H = H + 1	
	E	MAR = T	
		MDR = T||unbound tag	
		R = T||reference tag	
	P	A3 = R	:(DISPATCH)

III. Appendix: Nondeterminate Concatenate Trace

The nondeterminate concatenate μcode trace follows. No T0 register was assumed.

```
1            C     T = B                  <try_me_else C1>
                   T1 = A
                   i = 7
             E     R = T1 + (n+7)
             P     B = R
                   A = R

2            C     ?                      compare A to Z,
                                          if too close, copy-back.

3   LOOP:    C     j = i
                   i = i + 1
             E     test = j<(n+7)

4            C     T1 = Aj
             E     R = T1
                   jump = test
             P     Bj = R                 :jump(LOOP)

5   LOOP:    C     j = i
                   i = i + 1
             E     test = j<(n+7)

6            C     T1 = Aj
             E     R = T1
                   jump = test
             P     Bj = R                 :jump(LOOP)

7   LOOP:    C     j = i
                   i = i + 1
             E     test = j<(n+7)

8            C     T1 = Aj
             E     R = T1
                   jump = test
             P     Bj = R                 :jump(LOOP)

9            C     T1 = H
             E     R = T1
             P     B1 = R
                   HB = R
```

```
10            C      nil
              E      R = T
              P      B2 = R

11            C      T1 = TR
              E      R = T1
              P      B3 = R

12            C      T = E
              E      R = T
              P      B4 = R

13            C      T = CP
              E      R = T
              P      B5 = R

14            C      T = P
              E      R = T
              P      B6 = R                     :(DISPATCH)
```

```
15            C      T = A1                     <get_nil A1>
                     listT = islist(A1)
                     jump = not isref(A1)       :jump(NOTREF)

16     LOOP:  C      nil
              E      MAR = T,T1

17            C      jump = isvar(T1)           :jump(VAR)

18     VAR:   C      T = B
              E      notrailB = T>T1

19            C      T = HB
              E      notrailHB = T>T1

20            C      T = nil
              E      jump = notrailHB and notrailB
                     MAR = T1
                     MDR = T                    :jump(DISPATCH)

21            C      nil                        <assume trail stack>
              E      R = T1
              P      top of trail = R
                     TR = TR - 1                :(DISPATCH)
```

```
22            C      T1 = A2                    <get_val A2,A3>
```

```
23              C       T = A3                  :call(UNIFY)

24    UNIFY:    C       nil                     :rom

25    TREF:     C       nil
                E       MAR = T.T

26              C       jump = isref(T)         :jump(TREF)

27              C       nil                     :rom

28    TVAR:     C       nil
                E       MAR = T
                        MDR = T1

29              C       T1 = H
                E       local = T>T1

30              C       T1 = HB
                E       trailHB = T>T1
                        jump = local            :jump(LOCAL1)

31              C       jump = trailHB          :jump(RETURN)

32              C       nil
                E       R = T
                P       top of trail = R
                        TR = TR - 1             :(RETURN)

33              C       nil
                E       R = T1
                P       A2 = R                  :(DISPATCH)
```

```
 1              C       T = A1                  <get_list A1>
                        T1 = A1
                        mode = read
                        list = islist(A1)
                        jump = not isref(A1)    :jump(NOTREF)

 2    NOTREF:   C       jump = list
                E       R = T1
                P       S = R                   :jump(DISPATCH)
```

```
 3              C       T1 = S                  <unify_var X4>
                        S = S + 1
                E       MAR = T1,X4             :(DISPATCH)
```

```
4               C       T1 = S                  <unify_var A1>
                        S = S + 1
                E       MAR = T1,A1             :(DISPATCH)
--------------------------------------------------------------------------
5               C       T = A3                  <get_list A3>
                        T1 = A3
                        mode = read
                        list = islist(A3)
                        jump = not isref(A3)    :jump(NOTREF)

6      NOTREF:  C       jump = list
                E       R = T1
                P       S = R                   :jump(DISPATCH)
--------------------------------------------------------------------------
7               C       T = X4                  <unify_val X4>
                        T1 = S
                        S = S + 1
                E       MAR = T1,T1             :call(UNIFY)

8      UNIFY:   C       nil                     :rom

9      EQUAL:   C       jump = T=T1             :jump(RETURN)

10              C       nil                     :(FAIL)
--------------------------------------------------------------------------
       FAIL:    C       ?                       check if choice point is
                                                in buffer
                .
                .
                .

1      HIT:     C       T = B                   <pop stack to B>
                E       R = T
                P       A = R

2               C       T1 = B0                 <get old program pointer>
                E       R = T1
                P       P = R

3               C       T1 = B1                 <get old heap pointer>
                E       R = T1
                P       R

4               C       T = B2                  <get old trail pointer>
```

```
 5      LOOP:   C       T1 = TR                     <detrail bindings>
                E       done = T1 < T

 6              C       nil
                E       jump = done                 :jump(NEXT)

 7              C       T1 = top of trail
                        TR = TR + 1
                E       MAR = T1
                        MDR = T1||unbound tag       :(LOOP)

 8      LOOP:   C       T1 = TR                     <detrail bindings>
                E       done = T1 < T

 9              C       nil
                E       jump = done                 :jump(NEXT)

10              C       T1 = top of trail
                        TR = TR + 1
                E       MAR = T1
                        MDR = T1||unbound tag       :(LOOP)

11      LOOP:   C       T1 = TR                     <detrail bindings>
                E       done = T1 < T

12              C       nil
                E       jump = done                 :jump(NEXT)

13      NEXT:   C       T1 = B3                     <get old backtrack pointer>
                E       R = T1
                P       B = R

14              C       T1 = B4                     <get old continuation pointer>
                E       R = T1
                P       CP = R

15              C       T1 = B5                     <get old environment pointer>
                E       R = T1
                P       E = R

16              C       T = B6                      <get # of goal arguments>
                        i = 7

17      LOOP:   C       j = i
                        i = i + 1
                E       test = j<(n+7)

18              C       T1 = Bj                     <restore A registers>
                E       R = T1
```

```
                   jump = test
             P     Aj = R                    :jump(LOOP)

19   LOOP:   C     j = i
                   i = i + 1
             E     test = j<(n+7)

20           C     T1 = Bj                   <restore A registers>
             E     R = T1
                   jump = test
             P     Aj = R                    :jump(LOOP)

21   LOOP:   C     j = i
                   i = i + 1
             E     test = j<(n+7)

22           C     T1 = Bj                   <restore A registers>
             E     R = T1
                   jump = test
             P     Aj = R                    :jump(LOOP)

23           C     nil                       :(DISPATCH)
```

```
 1           C     T = A1                    <get_list A1>
                   T1 = A1
                   mode = read
                   list = islist(A1)
                   jump = not isref(A1)      :jump(NOTREF)

 2   LOOP:   C     nil
             E     MAR = T,T1

 3           C     jump = isvar(T1)          :jump(VAR)

 4   VAR:    C     T = B
                   mode = write
             E     notrailB = T>T1

 5           C     T = HB
             E     notrailHB = T>T1

 6           C     T = H
             E     jump = notrailHB and notrailB
                   MAR = T1
                   MDR = T                   :jump(DISPATCH)

 7           C     nil                       <assume trail stack>
             E     R = T1
```

```
                  P       top of trail = R
                          TR = TR - 1                   :(DISPATCH)
------------------------------------------------------------------------
 8                C       T = H                         <copy_var X4>
                          H = H + 1
                  E       MAR = T
                          MDR = T||unbound tag
                          R = T||reference tag
                  P       X1 = R                        :(DISPATCH)
------------------------------------------------------------------------
 9                C       T = H                         <copy_var X1>
                          H = H + 1
                  E       MAR = T
                          MDR = T||unbound tag
                          R = T||reference tag
                  P       X4 = R                        :(DISPATCH)
------------------------------------------------------------------------
10                C       T = A3                        <get_list A3>
                          T1 = A3
                          mode = read
                          list = islist(A3)
                          jump = not isref(A3)          :jump(NOTREF)

11       NOTREF:  C       jump = list
                  E       R = T
                  P       S = R                         :jump(DISPATCH)
------------------------------------------------------------------------
12                C       T = X4                        <unify_val X4>
                          T1 = S
                          S = S + 1
                  E       MAR = T1,T1                   :call(UNIFY)

13       UNIFY:   C       nil                           :rom

14       TREF:    C       nil
                  E       MAR = T,T

15                C       jump = isref(T)               :jump(TREF)

16                C       nil                           :rom

17       TVAR:    C       nil
                  E       MAR = T
                          MDR = T1
```

```
18                C      T1 = H
                  E      local = T>T1

19                C      T1 = HB
                  E      trailHB = T>T1
                         jump = local              :jump(LOCAL1)

20      LOCAL1:   C      T1 = B
                  E      trailB = T>T1

21                C      nil
                  E      jump = trailB             :jump(RETURN)

22                C      nil
                  E      R = T
                  P      X4 = R                    :(DISPATCH)
--------------------------------------------------------------------
23                C      T1 = S                    <unify_var A3>
                         S = S + 1
                  E      MAR = T1,A3               :(DISPATCH)
```

Acknowledgements

This work was supported by a Digital Equipment Corporation external research grant. I thank David Warren for his patience and invaluable assistance.

References

1. P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

2. T. Moto-Oka (Ed.). *Fifth Generation Computer Systems.* North Holland, 1982.

3. H. Nishikawa, M. Yokota, A. Yamamoto, K. Taki and S. Uchida. The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture. Logic Programming Workshop '83, Universidade Nova de Lisboa, June, 1983, pp. 53-73.

4. G. Radin. "The 801 Minicomputer." *IBM Journal of Research and Development 27* (May 1983), 237-246.

5. . Symbolics 3600 Technical Summary. Symbolics Inc., Cambridge, Massachusetts, 1983.

6. D. H. D. Warren. *Applied Logic -- Its Use and Implementation as Programming Tool.* Ph.D. Th., University of Edinburgh, Scotland, 1977. Available as Technical Note 290, Artificial Intelligence Center, SRI International.

7. D. H. D. Warren. An Abstract Prolog Instruction Set. Artificial Intelligence Center, SRI International, Menlo Park, California 94025, 1983.

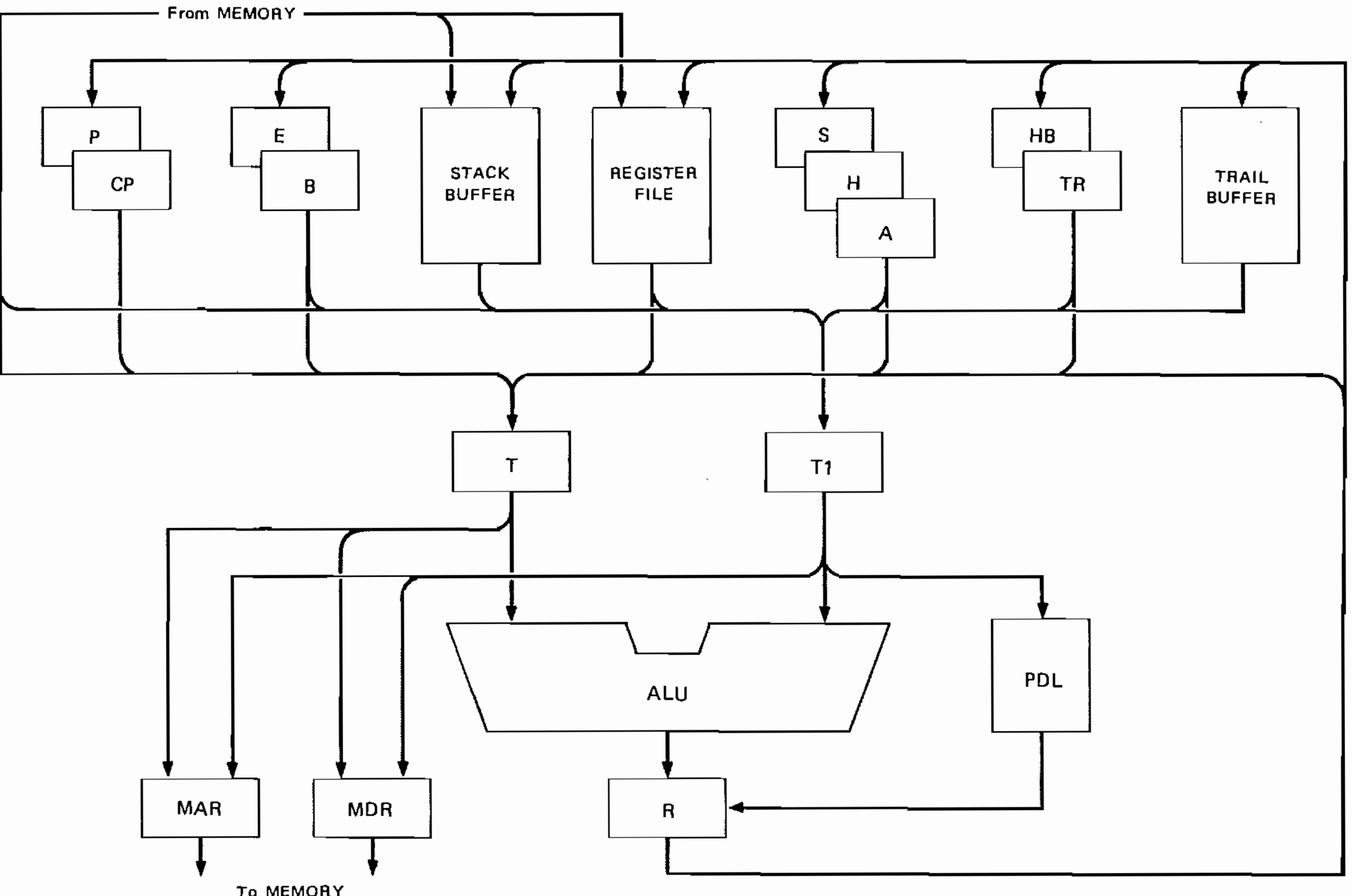


FIGURE 1 BASIC DATAPATHS OF E-UNIT

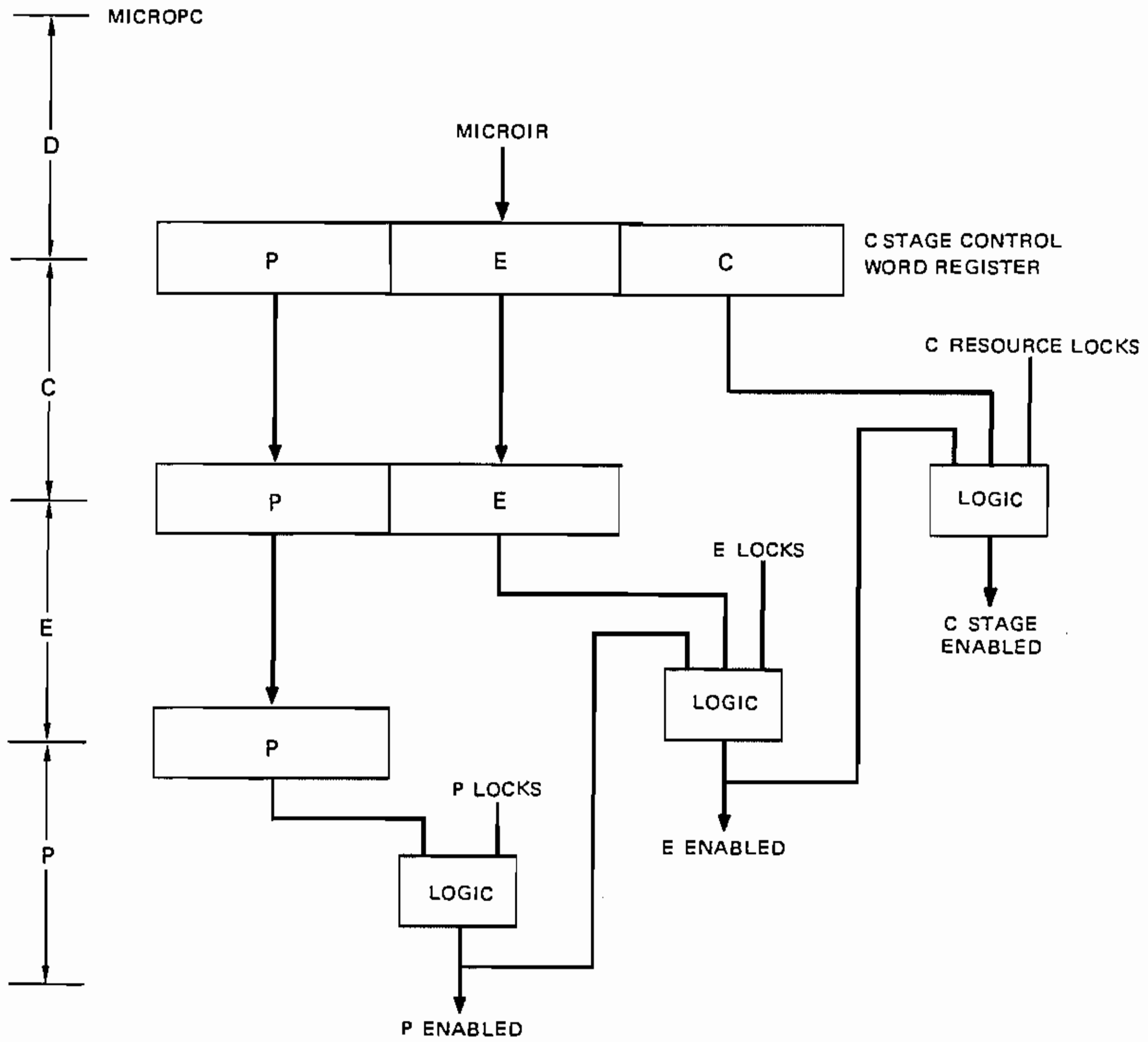


FIGURE 2 HIGH-LEVEL VIEW OF MICROCONTROL

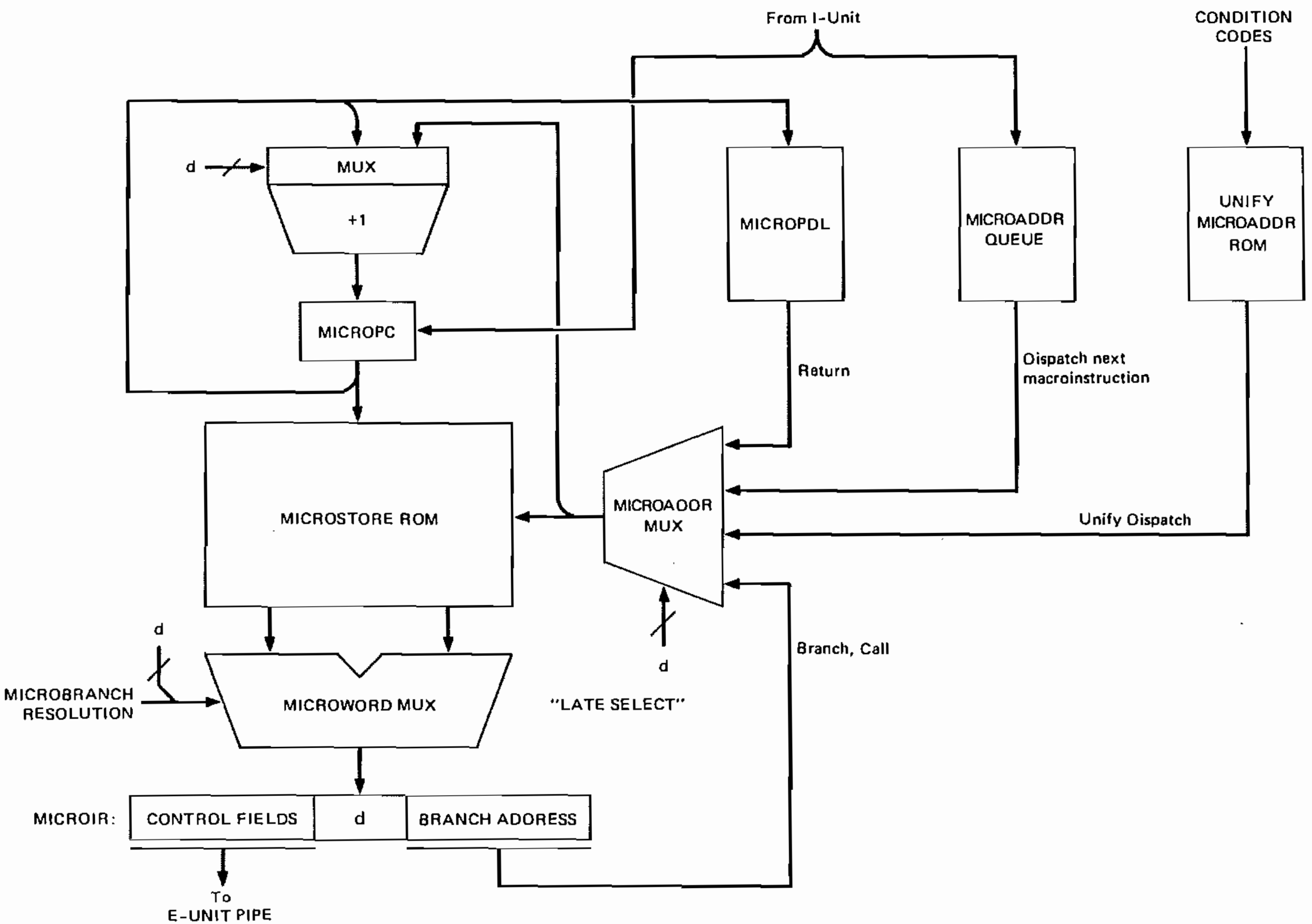


FIGURE 3 MICROCONTROLLER DATAPATHS

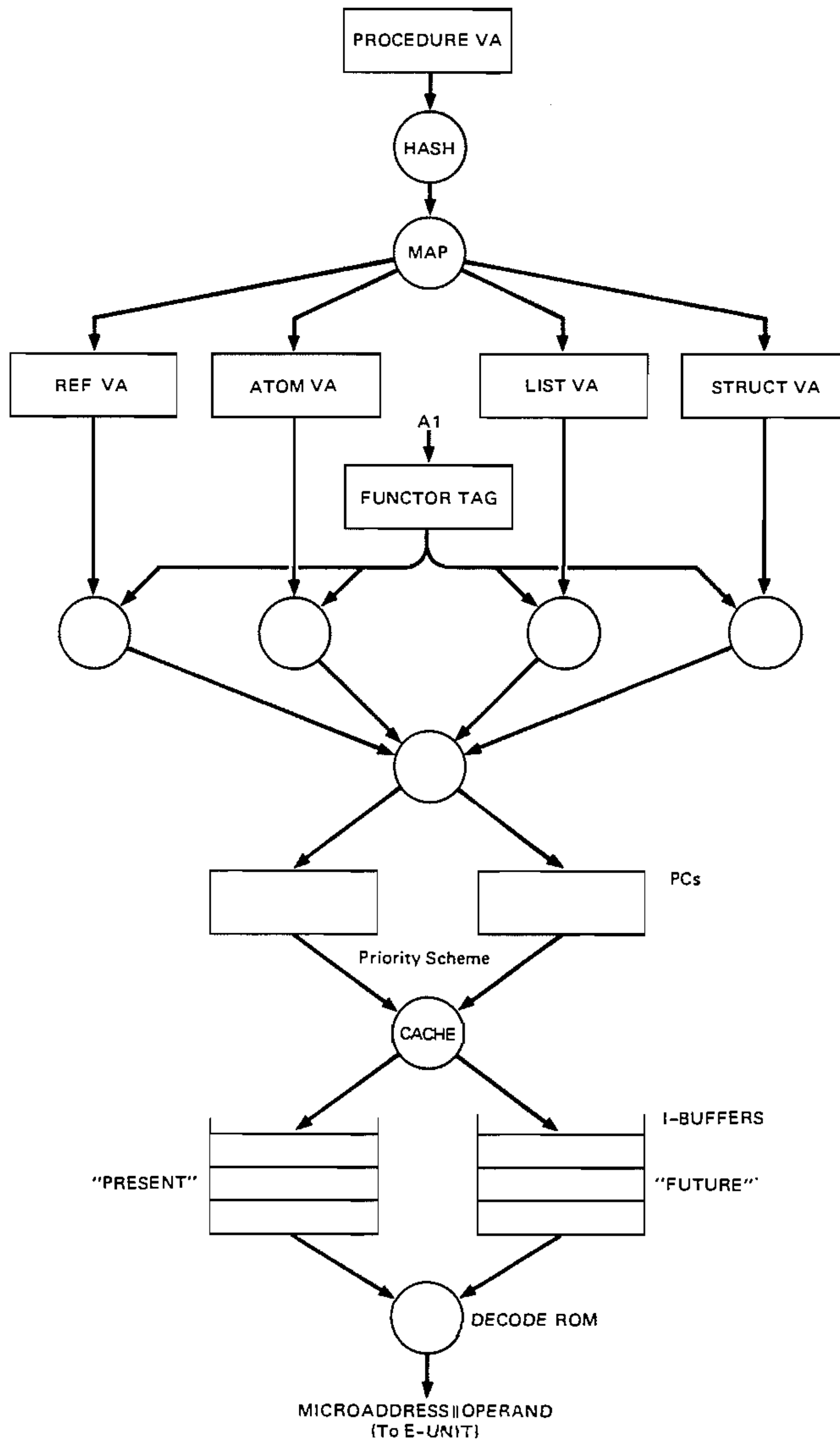


FIGURE 4 I-UNIT DATAFLOW