中 国 科 学
1989 年 3 月　　A 辑　　第 3 期

# 一种支持 Prolog 数据库操作和数值计算的顺序推理机系统结构的设计*

张晨曦　　慈云桂
（长沙工学院计算机系）

**摘　　要**

为了高效实现 Prolog，人们研制了面向 Prolog 的专用处理机或处理器片子[1]。然而，已有的这类机器大多或者不能支持 Prolog 数据库操作的实现，或者不能进行快速的数值计算。本文论述了我们设计的一种基于 WAM-PLUS 模型[2] 的顺序推理机 GKD-PLM 的系统结构。该结构不仅能高速执行 Prolog 代码，而且支持数据库操作和数值计算。文中在论述 GKD-PLM 的特点之后，介绍了 Prolog 处理机 SPP 的指令系统、数据表示、执行部件的硬件结构、微程序控制器的结构等。并对其性能进行了评估。

**关键词：顺序推理机，指令系统，数据表示，执行部件，微程序控制器**

## 一、引　　言

Prolog 的高效实现一直是逻辑程序设计领域的一个重要研究课题。近年来，国际上已经提出和实现了一些基于编译技术、面向 Prolog 的顺序推理机。例如，日本的 HPM[3]，IPP[4]，美国加州大学 Berkeley 分校的 PLM[5]，美国 Tick 提出的流水 Prolog 处理机[6]，等等。这些机器达到了较高的推理速度(几百 KLIPS)。然而，它们大多存在以下两个问题（或其中的一个）：

1）不能实现 Prolog 数据库操作；

2）数值计算能力较差（IPP 例外）。

内部谓词是 Prolog 的一个重要组成部分。我们认为，这些内部谓词，特别是数据库操作内部谓词，对于实用的 Prolog 系统来说，是必不可少的。因此，在设计顺序推理机时，必须从执行模型和系统结构等方面对数据库操作的实现提供支持。

实际应用中的计算问题可以分为三类：第一类以数值计算为主，第二类以符号运算为主，第三类则是两种计算量相当。对于第一类计算问题，传统机已能满足要求。对于第二类，已有的 Prolog 专用机也具有较高的性能。但是，对于第三类，上述机器都不能很好地适应。在现

本文 1988 年 3 月 25 日收到，1988 年 10 月 10 日收到修改稿。
* 国家自然科学基金资助项目。

有许多专家系统(例如,仿真专家系统)的应用中,要求能交替地进行数值计算和符号运算. 只单方面地提高其中一种计算的速度并不能有效地提高系统的性能. 本文论述我们设计的一种既能高效执行 Prolog 程序,又能快速进行数值计算,而且支持数据库操作的顺序推理机系统结构.

## 二、Prolog 执行模型——WAM-PLUS

Warren 抽象机 (WAM)[7]是 Warren 于 1983 年提出的一个非常高效的 Prolog 执行模型. 许多高性能 Prolog 系统 (例如文献 [3—9]) 都是以该模型为基础. 但是, WAM 对于 Prolog 中非逻辑成分的实现几乎没有提供支持,不能实现数据库修改等操作. 因此,还需进一步扩充和完善.

我们在文献[2]和张晨曦文中[1]提出了一个功能强、效率高、实用的 Prolog 执行模型——WAM-PLUS. 它由扩充了的 WAM 和非逻辑成分执行机制 NLEM 组成. 该模型实现了 Prolog 代码的动态修改,解决了原 WAM 不能实现数据库操作的问题. 该模型还采用了代码分类、执行驱动编译策略等优化实现技术. 本文论述的顺序推理机系统结构是在该模型的基础上设计的.

本文要求读者熟悉 WAM. 可参阅文献[7].

## 三、顺序推理机 GKD-PLM 总体结构的特点

GKD-PLM 的系统构成如图 1 所示. 它有以下特点:

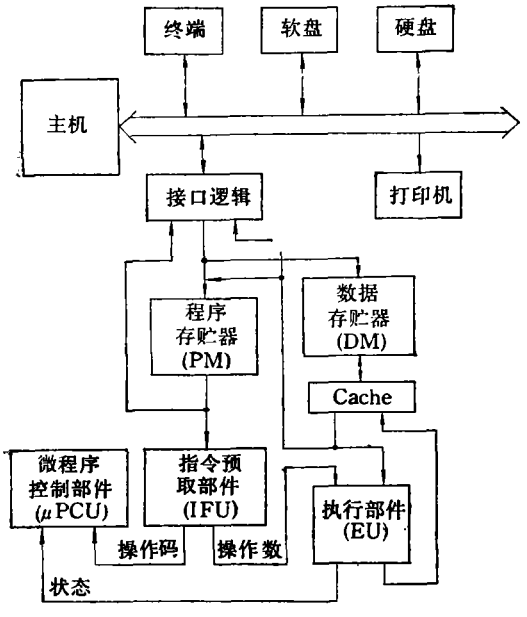


图 1 GKD-PLM 的结构框图

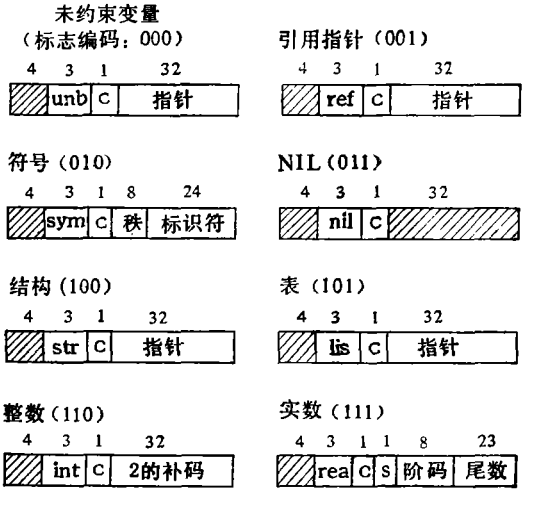


图 2 SPP 的数据类型
(C——cdr 位, S——符号位)

1) 张晨曦,长沙工学院计算机系博士学位论文,1987 年 12 月.

1. 采用主-辅机结构. 主机是一个微型机或超级微型机. Prolog 处理机 SPP 挂接在主机的总线上.

2. Prolog 处理机的指令系统包括一组面向 Prolog 的指令和一组面向数值计算的指令. 其数据通路也兼顾了数值计算的特点. 因此,它不仅能高效执行 Prolog 程序,而且为传统语言的实现提供了支持.

3. 采用带标志的数据表示.

4. 具有独立的程序存贮器和数据存贮器.

5. 采用微程序控制. 相对于 RISC 或其它传统机器的指令系统而言, Warren 代码是级别较高的代码,其指令具有较强的功能. 采用微程序控制,具有易于实现、修改,灵活性大, 可扩充性好等优点.

6. 当前微指令的执行与下一条微指令的预取重叠进行.

7. 执行部件的核心部分采用位片器件实现.

8. 对数据设置了高速缓存 Cache.

9. 对指令设置了预取部件.

## 四、SPP 的指令系统和数据表示

### 1. 指令系统

**1) 面向 Prolog 的指令** 这组指令共有 60 多条(不包括全部内部谓词指令). 其指令格式及其与符号指令(即汇编语言)的对应关系如表 1 所示. 符号指令到 SPP 指令的变换由汇编/装入程序完成. 下面对 SPP 中与 Warren 代码不同的指令作一简单介绍.

(1) unify_long_integer, unify_short_integer, unify_real, unify_constant_nil 和 unify_atom 它们是 unify_constant 指令的进一步加细. 分别用于当常数 $c$ 为 4 字节长整数、2 字节短整数、实数、nil 以及除 nil 之外的原子等情况. 对于 get_constant 和 put_constant 指令,也分别有类似的 4 条指令.

(2) unify_cdr_x 和 unify_cdr_y 由于 SPP 采用 cdr 编码法[10]表示结构数据,故需要有专门的指令来实现表尾的一致化. 表尾为 nil 的情况可由 unify_nil 指令处理. unify_cdr 的作用是处理表尾为变量的情况.

(3) call_direct, execute_direct, call_indirect 和 execute_indirect 这些是对应于含变元子目标的过程调用指令. 根据 WAM-PLUS 的要求, SPP 必须具备动态链接/装入的功能. 由此引起的问题是如何解决对非静态过程的调用的重定位. 传统的解决方法有两种. 一种是修改代码库中所有对该过程的调用. 另一种是要求所有调用都是间接的,这样,在重定位时,只需修改间接单元的内容. 前一种方法开销太大, 不可取. 我们采用了后一种方法. 但是,间接调用比直接调用的开销大得多,因此我们对于静态代码采用直接调用与间接调用相结合的方法.

我们把过程调用分为两类: 定义前调用和定义后调用. 若一个过程调用所调用的过程为静态过程,而且此过程的装入先于该过程调用的装入,则称该过程调用为定义后调用. 否则称之为定义前调用. 用直接调用实现定义后调用,用间接调用实现定义前调用.

call_direct 和 execute_direct 是直接调用指令, 其操作数为直接转移地址. call_indirect

表 1 内部指令的格式及其与符号指令的对应关系*

| 符号指令 | 内部指令 | | | |
|---|---|---|---|---|
| | 操作码 | | 操作数 | 指令长度(字节数) |
| | 符号名 | 十进制值 | | |
| | nop | 0 | A | 4 |
| unify_void *N* | unify_void | 1 | O | 2 |
| unify_nil | unify_nil | 2 | | 1 |
| unify_constant *c* | unify_long_integer | 3 | L | 5 |
| | unify_short_integer | 4 | I | 3 |
| | unify_real | 5 | R | 5 |
| | unify_atom | 6 | S | 4 |
| | unify_constant_nil | 7 | | 1 |
| unify_variable $X_i$ | unify_variable_x | 8 | V | 2 |
| unify_variable $Y_i$ | unify_variable_y | 9 | V | 2 |
| unify_value $X_i$ | unify_value_x | 10 | V | 2 |
| unify_value $Y_i$ | unify_value_y | 11 | V | 2 |
| unify_local_value $X_i$ | unify_local_value_x | 12 | V | 2 |
| unify_local_value $Y_i$ | unify_local_value_y | 13 | V | 2 |
| unify_cdr $X_i$ | unify_cdr_x | 14 | V | 2 |
| unify_cdr $Y_i$ | unify_cdr_y | 15 | V | 2 |
| get_nil $A_i$ | get_nil | 16 | V | 2 |
| get_constant *c*, $A_i$ | get_long_integer | 17 | VL | 6 |
| | get_short_integer | 18 | VI | 4 |
| | get_real | 19 | VR | 6 |
| | get_atom | 20 | VS | 5 |
| get_list $A_i$ | get_list | 21 | V | 2 |
| get_structure $f/a,A_i$ | get_structure | 22 | VOS | 6 |
| get_value $X_i$ $A_j$ | get_value_x | 23 | VV | 3 |
| get_value $Y_i,A_j$ | get_value_y | 24 | VV | 3 |
| get_variable $X_i,A_j$ | get_variable_x | 25 | VV | 3 |
| get_variable $Y_i,A_j$ | get_variable_y | 26 | VV | 3 |
| put_nil $A_i$ | put_nil | 27 | V | 2 |
| put_constant *c*, $A_i$ | put_long_integer | 28 | VL | 6 |
| | put_short_integer | 29 | VI | 4 |
| | put_real | 30 | VR | 6 |
| | put_atom | 31 | VS | 5 |
| put_list $A_i$ | put_list | 32 | V | 2 |
| put_structure $f/a,A_i$ | put_structure | 33 | VOS | 6 |
| put_unsafe_value $Y_i,A_j$ | put_unsafe_value | 34 | VV | 3 |
| put_value $X_i,A_j$ | put_value_x | 35 | VV | 3 |
| put_value $Y_i,A_j$ | put_value_y | 36 | VV | 3 |
| put_variable $X_i,A_j$ | put_variable_x | 37 | VV | 3 |
| put_variable $Y_i,A_j$ | put_variable_y | 38 | VV | 3 |
| | compile | 39 | M | 3 |
| prefetch_cont | prefetch_cont | 40 | | 1 |
| allocate | allocate | 41 | | 1 |
| deallocate | deallocate | 42 | | 1 |
| call $p/a,N$ | call_indirect | 43 | OM | 4 |

表 1(续)

| 符号指令 | 内部指令 | | | |
|---|---|---|---|---|
| | 操作码 | | 操作数 | 指令长度(字节数) |
| | 符号名 | 十进制值 | | |
| | call_direct | 44 | OA | 5 |
| execute *p/a* | execute_indirect | 45 | M | 3 |
| | execute_direct | 46 | A | 4 |
| ncall *p*,*N* | ncall_indirect | 47 | OM | 4 |
| | ncall_direct | 48 | OA | 5 |
| nexecute *p* | nexecute_indirect | 49 | M | 3 |
| | nexecute_direct | 50 | A | 4 |
| | jump | 51 | A | 4 |
| call_convention | call_convention | 52 | A | 4 |
| escape | escape | 53 | O | 2 |
| cut | cut | 54 | | 1 |
| cutd $Y_i$ | cutd | 55 | O | 2 |
| storeb $Y_i$ | storeb | 56 | O | 2 |
| try *L* | try | 57 | A | 4 |
| retry *L* | retry | 58 | A | 4 |
| trust *L* | trust | 59 | A | 4 |
| try_me_else *L* | try_me_else | 60 | A | 4 |
| | ctry_me_else | 61 | AA | 7 |
| retry_me_else *L* | retry_me_else<br>ctry_me_else | 62 | A | 4 |
| trust_me_else fail | trust_me_else fail<br>ctry_me_else | 63 | A | 4 |
| switch_on_constant *N* | switch_on_constant | 64 | M | 3 |
| switch_on_structure *N* | switch_on_structure | 65 | M | 3 |
| switch_on_term VLbl,<br>CLbl, LLbl, SLbl | switch_on_term | 66 | AAAA | 13 |
| | trust_fail | 67 | | 1 |
| | fail | 68 | | 1 |

* V——单字节．表示永久变量、暂时变量序号，或寄存器序号；O——单字节正整数，表示变元数等；I——2 字节整数；M——2 字节正整数，表示过程表序号、hash 表大小等；A——3 字节正整数，通常用作代码区的地址；S——3 字节．表示符号表地址；L——4 字节整数；R——4 字节浮点数．

和 execute_indirect 为间接调用指令，其操作数为被调用过程的过程表项的序号．过程表项中存放有过程的入口地址．

(4) ncall_direct, nexecute_direct, ncall_indirect 和 nexecute_indirect　它们是对应于不含变元的子目标的过程调用指令．在功能上它们与(3)中对应的指令类似．设置这组指令是为了实现 switch_on_term 指令的超前执行．

(5) ctry_me_else 和 trust_fail　这是新增加的选择点指令，用于实现动态 Prolog 代码．

(6) escape, cut, cutd 和 storeb　cut, cutd 和 stored 用于实现"!"内部谓词，escape 则是内部谓词公共调用指令．

(7) prefetch_cont　该指令用于支持继续指令（continuation）的预取．

(8) compile 其操作数为过程表项序号. 该指令的功能是调用编译器对过程进行编译并装入代码库.

(9) call_convention 对传统程序的调用指令. 有关上述指令的进一步论述见另文[1].

**2) 面向传统语言、数值计算的指令** 这组指令的数目与上一组相当. 其操作码的最高位为"1". 这些指令类似于传统机器的指令, 包括数据传输、算逻运算、程序控制、输入输出(到主机)等几类. 指令字长采用 24 位和 48 位两种. 指令格式参考了 IBM 370 的指令格式(这里的寄存器地址为 8 位),设有寄存器-寄存器型、变址型、寄存器-存贮器型、寄存器-直接数型等指令. 设置这组指令能大大提高 Prolog 的数值计算能力. 我们可以将 Prolog 程序中的表达式计算编译到该组指令.

### 2. SPP 的数据表示

为了支持 Prolog 的实现, SPP 采用了带标志的数据表示. 字长为 40 位. 每个数据字由标志域和值域两部分组成. SPP 的数据类型如图 2 所示. cdr 位是为实现用 cdr 编码法表示结构数据而设置的.

## 五、存贮器的设置

与传统语言相比, Prolog 程序的访存频率较高. 因此,存贮器的吞吐率是影响 Prolog 处理机性能的一个重要因素. 根据不同的用途设置分散存贮器能降低对吞吐率的要求. PEK[11] 中设置了 5 个存贮器. 但是,存贮器个数太多不仅会增加数据通路和控制电路的复杂性,而且还会降低存贮空间的利用率. 根据存取数据和存取指令的不同特点,我们设置了两个存贮器: 数据存贮器 DM 和程序存贮器 PM.

程序存贮器按字节寻址,具有一次读出多个字节(4 个或 8 个)的功能. 其容量为 1M 字节(最大可扩充至 16M 字节). 访问时间为 360ns,可采用动态 MOS 存贮片子实现.

数据存贮器的字长为 40 位,容量为 16M 字 (其寻址空间允许扩充至 4G 字). 要求它是一个多体交叉并行存贮器. 数据存贮器的空间按作用不同划分为几个区,包括过程表区、符号表区、源项区、全局栈、局部栈、尾栈以及传统程序数据区等. 各个区的边界由边界寄存器 (在图 3 的 BCHK 中)确定.

符号处理的特点之一是需要大量的存贮空间. Prolog 处理机中需设置大容量的数据存贮器. 但大容量存贮器的速度往往较低. SPP 中 DM 的访问时间为 360ns, 需三个微指令周期. 解决这个问题的办法之一是采用 Cache-主存存贮层次. 尽管在 Prolog 程序的执行过程中,不少访存(例如,由 dereference 操作引起的访存)的地址随机性较大, 但大多数访存仍具有较好的局部性. 而且传统程序访存的局部性是较好的. 因此,预计设置 Cache 能有效地减少访问时间. SPP 中 Cache 的访问时间为一个微周期.

## 六、指令预取部件和执行部件 EU

### 1. 预取部件 IFU

IFU 的主要作用是向执行部件提供加工好的指令,实现取指令和执行指令的重叠. IFU

1) 见 328 页脚注.

还具有超前执行部分指令的功能．我们将在另外的文章中介绍 IFU 的结构及指令的超前执行．

**2. 执行部件的硬件结构**

EU 的硬件结构如图 3 所示．其主要特点为

1. 用速度快、集成度高的 Am29300 系列位片器件[12]实现主要功能．

2. 采用了三总线结构．三条总线：BUS-A, BUS-B, BUS-C 使得参加运算的两个操作数和运算结果能并行传输，缩短了传输延迟．因而使 EU 在不采用流水技术的情况下仍然能实现较小的微周期．

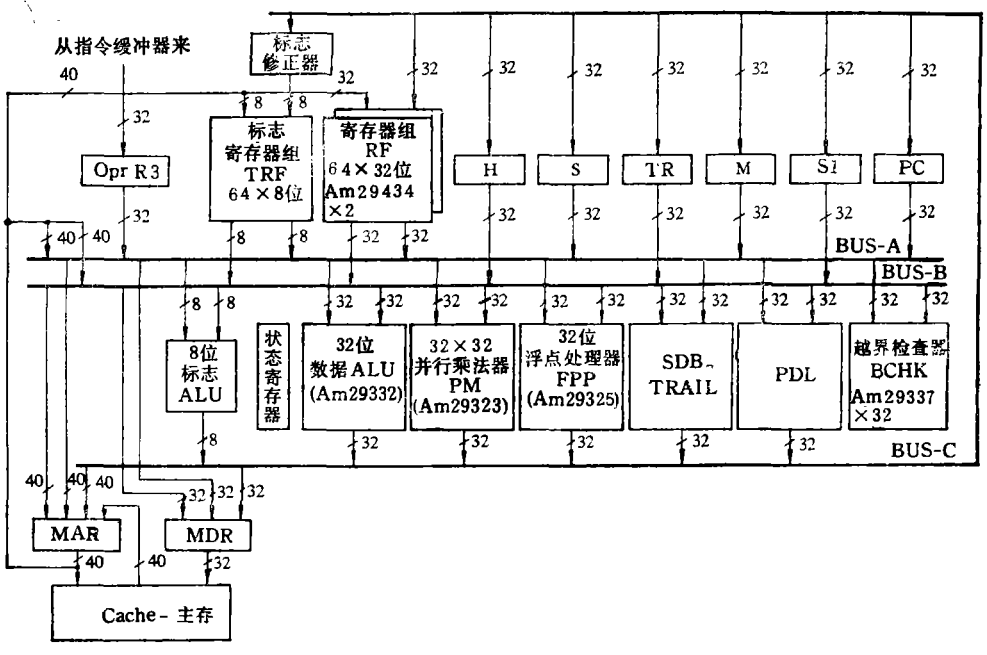


图 3　执行部件 EU 的硬件结构
(图中 Am29434 应为 Am29334)

3. 设置了硬件 PDL 和越界检查器 BCHK．

4. 设置了专门处理 tag 的硬件．Am29300 系列位片中的运算器是 32 位字长，而 SPP 数据字的长度却为 40 位．我们用 Am29300 系列位片处理 SPP 数据字的值域．对于标志域则另外设置了寄存器组 TRF 和标志运算器．

5. 设置了外部状态寄存器．虽然 Am 29332 片内设有状态寄存器，但其读出时间较长 (24ns)．为了使 μPCU 尽早取得状态码，同时考虑到有些状态位需反映标志运算器的运算结果，我们在 SPP 中设置了一个外部状态寄存器．

6. 设置了一个 64 字×32 位的数据寄存器组 RF 和一些指针计数器．RF 按作用不同划分为三个区：

1) Prolog 程序计算状态区：RF[0] ～ RF[23]．存放 Prolog 程序的计算状态．分配如表 2．

表 2 Prolog 程序状态区的分配情况

| 寄存器号 | RF[0]~RF[13] | | RF[14] | RF[15] | RF[16] | RF[17] |
|---|---|---|---|---|---|---|
| 作 用 | AX[0]~AX[13] | | T1 | T2 | CP | E |
| 寄存器号 | RF[18] | RF[19] | RF[20] | RF[21] | RF[22] | RF[23] |
| 作 用 | B | HB | NA | NY | CUTB | SCLB_PTR |

表 2 中 T1, T2 为工作单元, NA 为调用目标的变元个数, NY 为当前环境中永久变量的个数. CUTB 和 SCLB_PTR 分别用于实现"!"内部谓词和源程序库的搜索.

2) 传统程序计算状态区: RF[24] ~ RF[47]. 其中

RF[24] ~ RF[31]——传统程序专用寄存器;

RF[32] ~ RF[47]——传统程序通用寄存器组.

3) 全局变量区: RF[48] ~ RF[63].

为 Prolog 程序和传统程序分配不同的状态寄存器组,是为了在它们之间互相调用时,避免进行计算状态的保存与恢复.

图 3 中的 S1, S, H, TR 为指针计数器, PC 是传统程序的指令计数器, M 是剩余变元个数计数器. S1, S, M 是进行两个结构项的一致化操作时所需的三个基本量.

7. 设置了支持源程序库管理的专用硬件: SCLB_PTR 寄存器和 SDB_TRAIL 硬堆栈.

8. 除具有支持 Prolog 的专用硬件外,还设有支持数值计算的硬件. 例如,32 位浮点处理器和32位并行乘法器等.

9. 没采用流水技术. Tick 提出的 Prolog 处理机[6]以及 Berkeley 实现的 PLM[5]的执行部件均采用了三级流水. 一般说来,流水技术能提高吞吐率, 但会使控制电路变得更加复杂. 由于 SPP 执行部件中的高速器件 (Am29332 等片子支持直通 (flow-through) 结构) 和多总线结构使得数据通路的最大延迟(不包括浮点的情况)已经和微程序控制部件中的最大传输延迟相当(约 100ns), 因此, 可以不对执行部件的数据通路进行分级流水. 对于浮点操作来说,需要的话,可用 Am29325 内部的寄存器实现流水.

**3. 对 SPP 推理速度的评估**

根据估算结果, $\mu$PCU 中控制通路和 EU 中数据通路的最大传输延迟为 90—100ns.取 20%的余量,我们初步确定 SPP 的微周期为 120ns.

手算模拟结果表明, 当 Cache 的命中率为 100% 时, 对于 concatenate 过程的确定 (determinate) 执行, SPP 的推理速度约为 380KLIPS[1].

## 七、微程序控制器 $\mu$PCU

$\mu$PCU 采用二级重叠并行微程序控制. 其结构框图如图 4 所示. Am29331 是 16 位高速单片微程序时序控制器. 它具有一套功能很强的指令(指令码为 6 位), 能方便地实现各类转移、循环和子程序调用等. Am29331 有两个独立的转移地址输入端口: D和 A, 并且有四组

1) 见 328 页脚注.

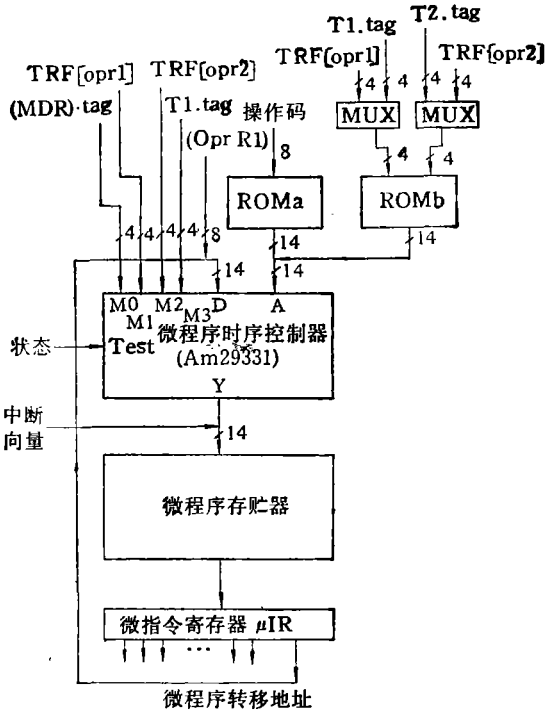


图 4　微程序控制部件 μPCU 的结构框图

16 路转移输入端口：M0 ~ M3。它还具有处理微程序中断的功能。Am29331 内部设有 33 字堆栈、计数器、条件码测试逻辑以及断点逻辑等。因此它很适合用作 SPP 的微程序控制器。

ROMa 和 ROMb 都是 EPROM 存贮器。ROMa 的容量为 256 字×14 位。它根据宏指令的操作码产生其微程序入口地址。ROMb 是支持一致化基本操作的专用硬件。在一致化操作中，经常需要根据两个项的类型进行多分支转移。这可以利用 ROMb 在一个微周期内实现。ROMb 的容量为 256 字× 28 位。它以两个项的 tag 作为其地址输入，其输出的高 14 位和低 14 位分别用作 unify 和 unify_sdb (用于源程序库的管理)基本操作的转移地址。

μPCU 中下一条微指令地址的来源有下述 8 种：

1. Am29331 中微程序计数器的内容。实现微程序的顺序执行。

2. Am29331 中栈顶的内容。实现微循环和微子程序的返回。

3. Am29331 中中断返回地址寄存器的内容。

4. 外来中断向量。实现微中断。

5. 当前微指令中的直接转移地址。实现无条件转移或 2 分支条件转移。

6. 多分支转移地址。在一些指令和基本操作的微程序中，经常需要测试 (MDR). tag, TRF [opr1], TRF [opr2] 和 T1. tag (opr1 和 opr2 分别表示指令的第一、第二操作数)。为了在一个微周期内实现这种测试转移，我们设置专门的通路把它们直接送入 μPCU 中 Am 29331 的 M0 ~ M3。

7. ROMa 的输出。

8. ROMb 的输出.

## 八、结 束 语

基于编译技术的顺序推理机是新一代计算机研究中的一个重要课题. 本文论述了我们在 Prolog 抽象执行模型 WAM-PLUS 的基础上设计的顺序推理机 GKD-PLM 的系统结构.其中的隶属 Prolog 处理机 SPP 不仅能高效执行类 Warren 代码, 而且能支持数据库操作和数值计算. 这是 SPP 不同于其它 Prolog 处理机的主要特点之一. 由于执行部件的核心部分采用高速的 VLSI 位片器件实现,所以虽然我们为了简化控制而未采用流水技术, 但 SPP 仍具有较高的性能. 估算结果表明,对于 concatenate 过程的确定执行, SPP 的速度约为 380 KLIPS.

## 参 考 文 献


[1] Srini, V. P. et al., in *Proceedings of the International Conference on Computer Design*, New York, Oct., 1987.

[2] 张晨曦,慈云桂,中国科学A辑, 1988, 12: 1323.

[3] Nakazaki, R. et al., in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June, 1985, 191—197.

[4] Abe, S. et al., in *Proceedings of the 14th Internationaal Symposium on Computer Architecture*, 1987, 100—107.

[5] Dobry, T. P. et al., in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June, 1985, 180—190.

[6] Tick, E. and Warren, D. H. D., in *New Generation Computing*, OHMSHA. LTD. and Springer-Verlag, 1984, 2: 323—345.

[7] Warren, D. H. D., *Technical Note* 309, *SRI International, Oct.*, 1983.

[8] Lindholm, T. G. and O'keefe, R. A., in *Proceedings of the 4th International Conference on Logic Programming*, 1987.

[9] Hermenegildo, M. V., in *Proceedings of the 3rd International Conference on Logic Programming*, July, 1986, 25—39.

[10] Dobry, T. P. et al., in *Proceedings of the MICRO* 17, Oct., 1984.

[11] Tamura, N. et al., in *Proc. of the International Conference of FGCS*, 1984, 542—550.

[12] *Bopolar Microprocessor Logic and Interface Data Book*, Advance Micro Device, Inc., 1985.